André Vasconcelos Carrusca

Licenciatura em Engenharia Informática

Gestão de micro-serviços na Cloud e Edge

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática

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

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

Júri

Presidente: Prof. Dr. Joaquim Francisco Ferreira Silva
Arguente: Prof. Dr. João Coelho Garcia
Vogal: Prof. Dra. Maria Cecília Farias Lorga Gomes

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Setembro, 2018

Gestão de micro-serviços na Cloud e Edge

Copyright © André Vasconcelos Carrusca, Faculdade de Ciências e Tecnologia, Universidade NOVA de Lisboa.
A Faculdade de Ciências e Tecnologia e a Universidade NOVA de Lisboa têm o direito, perpétuo e sem limites geográficos, de arquivar e publicar esta dissertação através de exemplares impressos reproduzidos em papel ou de forma digital, ou por qualquer outro meio conhecido ou que venha a ser inventado, e de a divulgar através de repositórios científicos e de admitir a sua cópia e distribuição com objetivos educacionais ou de investigação, não comerciais, desde que seja dado crédito ao autor e editor.

Este documento foi gerado utilizando o processador (pdf)LaTeX, com base no template "novathesis" [1] desenvolvido no Dep. Informática da FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

Agradecimentos

Gostaria de começar por agradecer à minha orientadora, a professora Cecília Gomes, pelos conselhos, confiança, apoio e disponibilidade prestados a esta dissertação. Um agradecimento ao meu coorientador, professor João Leitão, pela sua disponibilidade e sugestões que contribuíram para o desenvolvimento do trabalho realizado.

Agradeço à Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa, em especial ao departamento de Informática e aos seus docentes por todo o conhecimento transmitido.

Um grande obrigado a todos os meus amigos que, ao longo destes anos, me ajudaram a ultrapassar os mais diversos desafios e cujo apoio foi essencial para a conclusão desta dissertação.

Um obrigado especial à minha família! Aos meus pais, avós, irmãos, tios e primos que sempre me apoiaram ao longo da minha vida e que sem eles este percurso teria sido muito mais difícil.

Resumo

O aumento do número de dispositivos móveis nos últimos anos tem elevado o número de pedidos realizados aos serviços de *backend* da *cloud*, bem como a quantidade de dados produzida. Este facto tem levado à utilização de novas arquiteturas no desenvolvimento dos sistemas e à necessidade de novas estratégias para garantir a qualidade dos serviços.

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

A heterogeneidade dos locais de *deployment* considerados, ou seja, a *cloud* e a *edge*, torna complexa a gestão dos micro-serviços, em particular a migração/replicação dos serviços. É necessário decidir quando se processa a migração/replicação de um dado serviço, e para que local, sendo depois também necessário decidir como se processa essa migração/replicação. Ao existirem vários micro-serviços, em que pode haver dependências entre eles, a sua gestão é mais complexa, bem como a decisão sobre as suas dependências.

A solução consiste num protótipo aplicacional com mecanismos automáticos de migração e replicação de micro-serviços na *cloud* e na *edge*, que permite uma diminuição no tempo de acesso a esses serviços, resultando num melhor desempenho aplicacional. Estes mecanismos possibilitam o *deployment* de micro-serviços automaticamente na *cloud* e *edge* consoante certas regras e métricas configuráveis (ex.: latência, número de acessos).

A avaliação realizada permitiu comprovar que a utilização da *cloud* e da *edge* para a execução dos serviços permitiu uma diminuição dos tempos de acessos aos mesmos, em comparação à utilização apenas da *cloud*.

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

Abstract

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

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

The heterogeneity of the deployment sites considered, i.e., cloud and edge, makes the management of microservices complex, in particular, the migration/replication of services. It is necessary to decide when to migrate/replicate a given service and to where, and then it is necessary to decide how this migration/replication is processed. When several distinct microservices are interdependent, the decision on if and how to migrate/replicate a service may implicate the migration/replication of other services.

This work consists of an application prototype with automatic mechanisms for microservices migration and replication on the cloud and edge. The implementation aims to reduce the access time to these services leading in better application performance. These mechanisms allow the deployment of microservices automatically in cloud and edge according to certain configurable rules and metrics (e.g., response times, accesses).

With the evaluation, it was possible to prove that by executing microservices on the cloud and on the edge resulted in reduction on this services access time, compared with the execution of microservices only on the cloud.

Keywords: Microservices, cloud, edge, microservice deployment

Índice

Lista de Figuras xv

Lista de Tabelas xix

Listagens xxi

Glossário xxiii

Siglas xxv

1 Introdução 1

1.1 Contexto . . . 1
1.2 Problema . . . 3
1.3 Solução proposta . . . 4
1.4 Contribuições . . . 5
1.5 Estrutura do documento . . . 6

2 Estado da arte 7

2.1 Computação orientada a serviços - SOC . . . 7
2.2 Micro-serviços . . . 10
2.2.1 SOA e micro-serviços . . . 11
2.2.2 Vantagens e desvantagens da arquitetura de micro-serviços . . . 12
2.2.3 Arquitetura de micro-serviços . . . 14
2.2.4 Transição de uma aplicação monolítica para micro-serviços . . . 24
2.2.5 Utilização da arquitetura de micro-serviços na prática . . . 26
2.3 *Cloud Computing* . . . 28
2.3.1 Modelo de *Deployment* . . . 30
2.3.2 Virtualização de recursos . . . 30
2.3.3 Gestão de recursos e agendamento . . . 31
2.3.4 Infraestruturas de *cloud* . . . 33
2.4 *Edge Computing* . . . 35
2.4.1 Benefícios da *Edge Computing* . . . 35
2.4.2 Desafios da *Edge Computing* . . . 36
2.5 Computação na *Cloud* e *Edge* . . . 36

2.5.1 Monitorização em arquiteturas heterogéneas 37
2.5.2 Orquestração de micro-serviços e controlo da elasticidade 37
2.5.3 Migração de serviços 38
2.6 *Containers* 39
2.6.1 *Containers* e micro-serviços 40
2.6.2 Orquestração de *containers* 40
2.7 Trabalho relacionado 41
2.7.1 CAUS - Custom AutoScaler 42
2.7.2 ENORM - Edge Node Resource Management 43
2.8 Sumário 47

3 Solução proposta **49**
3.1 Arquitetura de um sistema de gestão de micro-serviços com propriedades autonómicas 49
3.2 Arquitetura do componente de gestão de micro-serviços 51
3.2.1 Informações necessárias à execução de serviços e nós 53
3.2.2 Funcionalidades principais oferecidas pelo componente de gestão 54
3.2.3 Subcomponentes necessários ao sistema de gestão 55
3.3 Soluções de escalabilidade 58
3.4 Interligação dos subcomponentes e funcionalidades: processo de reconfiguração 59
3.4.1 Monitorização 61
3.4.2 Análise 62
3.4.3 Planeamento 63
3.4.4 Execução 64
3.5 Restrições e limitações 65
3.5.1 Infraestrutura 65
3.5.2 Serviços 66
3.6 Análise comparativa entre a solução e os trabalhos relacionados 66
3.7 Cenários de migração e replicação 67
3.7.1 Fatores que influenciam a replicação e migração 67
3.7.2 Cenários considerados 69

4 Implementação **75**
4.1 Componente de gestão de micro-serviços 75
4.1.1 Interface de utilizador (UI) 75
4.1.2 APIs 79
4.1.3 Base de dados 79
4.1.4 Gestão de *containers* e nós do *cluster* - Docker 80
4.1.5 Obtenção de métricas 81
4.1.6 Serviço de regras (*Rules service*) 82

4.1.7 Gestão de nós 83
4.1.8 Processo de reconfiguração: nós 85
4.1.9 Processo de reconfiguração: *containers* 86
4.2 Subcomponentes de sistema 86
4.2.1 Componente de registo de serviços (*service registry*) 86
4.2.2 Componente para registar e descobrir serviços 86
4.2.3 Componente de balanceamento de carga 87
4.2.4 Componente para monitorização de pedidos 89
4.2.5 Componente para monitorização dos nós 89

5 Validação e Avaliação experimental **91**
5.1 Caso de estudo 91
5.2 Adaptação do caso de estudo à arquitetura da solução 92
5.3 Avaliação experimental 94
5.3.1 Testes de carga aos micro-serviços da aplicação *Sock Shop* 94
5.4 Discussão 123

6 Conclusões e trabalho futuro **125**
6.1 Conclusões 125
6.2 Trabalho futuro 127

Bibliografia **129**

Lista de Figuras

1.1 Visão simplificada da solução. 5

2.1 Diagrama de uma aplicação monolítica. Adaptado de [53]. 10
2.2 Diagrama de uma aplicação em micro-serviços. Adaptado de [52]. 11
2.3 Taxonomia da arquitetura de micro-serviços. Adaptado de [24]. 15
2.4 Utilização de uma API Gateway com micro-serviços. Adaptado de [60]. . . . 17
2.5 Utilização de *event sourcing*. Adaptado de [60]. 21
2.6 Localização da computação na *Edge*. Adaptado de [10]. 35
2.7 Máquinas virtuais vs. *containers*. Adaptado de [46]. 40

3.1 Visão do sistema de gestão de micro-serviços proposto e componentes associados. 52
3.2 Comunicação de um serviço com outro serviço, com recurso ao componente usado para registar e descobrir serviços. 56
3.3 Funcionamento do componente de balanceamento de carga. 58
3.4 Parte do processo de reconfiguração dinâmica referente aos serviços. 60
3.5 Parte do processo de reconfiguração dinâmica referente aos nós. 61
3.6 Decisão de replicação de um serviço sem dependências. 71
3.7 Decisão de migração de um serviço sem dependências. 71

4.1 Visão detalhada do componente de gestão de micro-serviços. 76
4.2 Página das aplicações. 77
4.3 Página dos serviços. 77
4.4 Página de nós da *edge*. 77
4.5 Página das regiões. 77
4.6 Página dos *containers*. 78
4.7 Página dos nós. 78
4.8 Página de inicialização de servidores Eureka. 78
4.9 Página de inicialização de *load balancers*. 78
4.10 Página inicial de gestão das regras. 79
4.11 Página inicial de gestão das métricas simuladas. 79
4.12 Página para adicionar uma nova condição. 83
4.13 Página para adicionar uma nova regra. 83

4.14 Utilização do Prometheus pelo componente de gestão de micro-serviços para a obtenção das métricas de um nó. 90

5.1 Arquitetura da aplicação *Sock Shop*. 92
5.2 Cenário dos testes de carga sem replicação e com replicação na *cloud*. 95
5.3 Cenário dos testes de carga com replicação na *cloud* e na *edge*. 95
5.4 Teste de carga ao acesso do catálogo do produtos, sem replicação. 97
5.5 Métricas do serviço Frontend, durante o teste de carga ao acesso do catálogo dos produtos, sem replicação. 98
5.6 Métricas do serviço Catalogue, durante o teste de carga ao acesso do catálogo dos produtos, sem replicação. 98
5.7 Teste de carga ao acesso do catálogo dos produtos, com replicação na *cloud*. . 100
5.8 Métricas da 1ª réplica do serviço Frontend, durante o teste de carga ao acesso do catálogo dos produtos, com replicação na *cloud*. 102
5.9 Métricas da 2ª réplica do serviço Frontend, durante o teste de carga ao acesso do catálogo dos produtos, com replicação na *cloud*. 102
5.10 Métricas da 3ª réplica do serviço Frontend, durante o teste de carga ao acesso do catálogo dos produtos, com replicação na *cloud*. 102
5.11 Métricas da 4ª réplica do serviço Frontend, durante o teste de carga ao acesso do catálogo dos produtos, com replicação na *cloud*. 102
5.12 Métricas da 5ª réplica do serviço Frontend, durante o teste de carga ao acesso do catálogo dos produtos, com replicação na *cloud*. 103
5.13 Métricas da 1ª réplica do serviço Catalogue, durante o teste de carga ao acesso do catálogo dos produtos, com replicação na *cloud*. 103
5.14 Métricas da 2ª réplica do serviço Catalogue, durante o teste de carga ao acesso do catálogo dos produtos, com replicação na *cloud*. 103
5.15 Métricas da 3ª réplica do serviço Catalogue, durante o teste de carga ao acesso do catálogo dos produtos, com replicação na *cloud*. 104
5.16 Métricas da 4ª réplica do serviço Catalogue, durante o teste de carga ao acesso do catálogo dos produtos, com replicação na *cloud*. 104
5.17 Teste de carga ao acesso do catálogo dos produtos, com replicação na *cloud* e na *edge* (média). 105
5.18 Teste de carga ao acesso do catálogo dos produtos, com replicação na *cloud* e na *edge* (Portland). 106
5.19 Teste de carga ao acesso do catálogo dos produtos, com replicação na *cloud* e na *edge* (Londres). 106
5.20 Variação das réplicas no teste de carga ao acesso do catálogo dos produtos, com replicação na *cloud* e na *edge*. 107
5.21 Métricas da 1ª réplica do serviço Frontend, durante o teste de carga ao acesso do catálogo dos produtos, com replicação na *cloud* e na *edge*. 108

5.22 Métricas da 2ª réplica do serviço Frontend, durante o teste de carga ao acesso do catálogo dos produtos, com replicação na *cloud* e na *edge*. 108
5.23 Métricas da 3ª réplica do serviço Frontend, durante o teste de carga ao acesso do catálogo dos produtos, com replicação na *cloud* e na *edge*. 108
5.24 Métricas da 4ª réplica do serviço Frontend, durante o teste de carga ao acesso do catálogo dos produtos, com replicação na *cloud* e na *edge*. 108
5.25 Métricas da 5ª réplica do serviço Frontend, durante o teste de carga ao acesso do catálogo dos produtos, com replicação na *cloud* e na *edge*. 109
5.26 Métricas da 1ª réplica do serviço Catalogue, durante o teste de carga ao acesso do catálogo dos produtos, com replicação na *cloud* e na *edge*. 109
5.27 Métricas da 2ª réplica do serviço Catalogue, durante o teste de carga ao acesso do catálogo dos produtos, com replicação na *cloud* e na *edge*. 109
5.28 Métricas da 3ª réplica do serviço Catalogue, durante o teste de carga ao acesso do catálogo dos produtos, com replicação na *cloud* e na *edge*. 110
5.29 Métricas da 4ª réplica do serviço Catalogue, durante o teste de carga ao acesso do catálogo dos produtos, com replicação na *cloud* e na *edge*. 110
5.30 Métricas da 5ª réplica do serviço Catalogue, durante o teste de carga ao acesso do catálogo dos produtos, com replicação na *cloud* e na *edge*. 110
5.31 Métricas da 6ª réplica do serviço Catalogue, durante o teste de carga ao acesso do catálogo dos produtos, com replicação na *cloud* e na *edge*. 110
5.32 Teste de carga ao efetuar logins e registos de utilizadores, sem replicação. . . 111
5.33 Métricas do serviço Frontend, durante o teste de carga ao efetuar logins e registos de utilizadores, sem replicação. 112
5.34 Métricas do serviço User, durante o teste de carga ao efetuar logins e registos de utilizadores, sem replicação. 113
5.35 Teste de carga ao efetuar logins e registos de utilizadores, com replicação na *cloud*. 114
5.36 Métricas da 1ª réplica do serviço Frontend, durante o teste de carga ao efetuar logins e registos de utilizadores, com replicação na *cloud*. 116
5.37 Métricas da 2ª réplica do serviço Frontend, durante o teste de carga ao efetuar logins e registos de utilizadores, com replicação na *cloud*. 116
5.38 Métricas da 3ª réplica do serviço Frontend, durante o teste de carga ao efetuar logins e registos de utilizadores, com replicação na *cloud*. 116
5.39 Métricas da 1ª réplica do serviço User, durante o teste de carga ao efetuar logins e registos de utilizadores, com replicação na *cloud*. 117
5.40 Métricas da 2ª réplica do serviço User, durante o teste de carga ao efetuar logins e registos de utilizadores, com replicação na *cloud*. 117
5.41 Métricas da 3ª réplica do serviço User, durante o teste de carga ao efetuar logins e registos de utilizadores, com replicação na *cloud*. 117
5.42 Teste de carga ao efetuar logins e registos de utilizadores, com replicação na *cloud* e na *edge* (média). 119

5.43 Teste de carga ao efetuar logins e registos de utilizadores, com replicação na *cloud* e na *edge* (Portland). 119
5.44 Teste de carga ao efetuar logins e registos de utilizadores, com replicação na *cloud* e na *edge* (Londres). 120
5.45 Variação de réplicas no teste de carga ao efetuar logins e registos de utilizadores, com replicação na *cloud* e na *edge*. 121
5.46 Métricas da 1ª réplica do serviço Frontend, durante o teste de carga ao efetuar logins e registos de utilizadores, com replicação na *cloud* e na *edge*. 121
5.47 Métricas da 2ª réplica do serviço Frontend, durante o teste de carga ao efetuar logins e registos de utilizadores, com replicação na *cloud* e na *edge*. 121
5.48 Métricas da 3ª réplica do serviço Frontend, durante o teste de carga ao efetuar logins e registos de utilizadores, com replicação na *cloud* e na *edge*. 122
5.49 Métricas da 1ª réplica do serviço User, durante o teste de carga ao efetuar logins e registos de utilizadores, com replicação na *cloud* e na *edge*. 122
5.50 Métricas da 2ª réplica do serviço User, durante o teste de carga ao efetuar logins e registos de utilizadores, com replicação na *cloud* e na *edge*. 122
5.51 Métricas da 3ª réplica do serviço User, durante o teste de carga ao efetuar logins e registos de utilizadores, com replicação na *cloud* e na *edge*. 123

Lista de Tabelas

2.1 Prós e contras das *frameworks* Dropwizard, Light Rest 4J e Spring Boot. . . . 16

2.2 Estilos de IPC. Adaptado de [60]. 18

3.1 Exemplo de um agendamento de evento. 55

3.2 Composição da linha de uma métrica dos serviços. 61

3.3 Composição da linha de uma métrica dos nós. 62

3.4 Comparação entre as soluções. 66

4.1 APIs para registar e descobrir serviços. 87

4.2 APIs da aplicação de reconfiguração do *load balancer*. 88

4.3 Estrutura dos dados recolhidos na monitorização dos pedidos. 89

4.4 APIs para consultar e adicionar informações sobre os pedidos aos serviços. . 90

5.1 Valores do acesso ao catálogo sem replicação. 96

5.2 Valores do acesso ao catálogo, com replicação na *cloud*. 99

5.3 Diferença dos valores do acesso entre os testes do catálogo com replicação na *cloud* e sem replicação. 99

5.4 Custos da replicação dos micro-serviços Frontend, Catalogue e Catalogue DB. 100

5.5 Valores do acesso ao catálogo, com replicação na *cloud* e na *edge* (média). . . 107

5.6 Valores do acesso ao catálogo, com replicação na *cloud* e na *edge* (Portland). . 107

5.7 Valores do acesso ao catálogo, com replicação na *cloud* e na *edge* (Londres). . 107

5.8 Diferença dos valores do acesso entre os testes do catálogo com replicação apenas na *cloud* e com replicação na *cloud* e na *edge* (Londres). 107

5.9 Valores dos logins e registos de utilizadores, sem replicação. 112

5.10 Valores dos logins e registos de utilizadores, com replicação na *cloud*. 114

5.11 Diferença dos valores do acesso entre os testes dos logins e registos de utilizadores com replicação na *cloud* e sem replicação. 114

5.12 Custos da replicação dos micro-serviços User e User DB. 115

5.13 Valores dos logins e registos de utilizadores, com replicação na *cloud* e na *edge* (média). 120

5.14 Valores dos logins e registos de utilizadores, com replicação na *cloud* e na *edge* (Portland). 120

5.15 Valores dos logins e registos de utilizadores, com replicação na *cloud* e na *edge* (Londres). 120
5.16 Diferença dos valores do acesso entre os testes dos logins e registos de utilizadores com replicação apenas na *cloud* e com replicação na *cloud* e na *edge* (Londres). 120

Listagens

4.1 Adicionar um novo nó ao *cluster*. 80
4.2 Procura de um nó disponível para a execução de um *container*. 84
4.3 Inicialização do `Timer` do processo de reconfiguração dos nós. 85
5.1 Versão original da obtenção do *endpoint* do serviço Shipping no serviço Orders. 93
5.2 Versão modificada da obtenção do *endpoint* do serviço Shipping no serviço Orders. 93

Glossário

consistência eventual
: É um tipo de consistência em que pode existir conflitos, isto é, para uma dada consulta podem ser apresentados diferentes resultados, mas existe uma comunicação entre os processos para tentar resolvê-los, para que a uma dada altura seja acordado o valor definitivo. Assim, se durante um período de tempo, não existirem mais alterações nos dados, todos os processos irão manter o mesmo valor para os dados, para quando existir uma consulta, eventualmente (*eventually*), seja apresentado o mesmo valor para todos os que consultarem [70].

consistência forte
: É um tipo de consistência que assegura que apenas um estado consistente pode ser visto. Todos os processos do sistema mantêm um estado semelhante e devem retornar o mesmo valor para uma dada consulta [70].

Siglas

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

Capítulo 1

Introdução

O presente capítulo apresenta o contexto geral no qual se enquadra o trabalho, indica o problema encontrado e fornece uma breve explicação sobre a solução proposta. A seguir são apresentadas as contribuições resultantes da solução. A finalizar, encontra-se a estrutura geral do restante documento.

1.1 Contexto

A evolução da tecnologia nos últimos anos e a exigência, por parte das pessoas, pela inovação contínua, obrigou as empresas a ter que repensar a gestão das suas aplicações. Além de ser necessário lançar atualizações para corrigir erros, as empresas têm que estar em constante progressão de forma a fornecer novas funcionalidades aos seus utilizadores e com qualidades de serviço adequadas, pois sem isso podem perder clientes e, consequentemente, receitas, ao longo do tempo [17, 22]. Outro aspeto que tem forçado a evolução dos sistemas, é o aumento do número de dispositivos móveis que se tem observado nos últimos tempos [3]. Isto significa que, por parte destes dispositivos, existe um acréscimo na quantidade de dados produzidos e também no número de pedidos realizados aos serviços de *backend*. É portanto necessário encontrar estratégias de forma a garantir a Qualidade de Serviço (QoS), pois se esta se degradar, por exemplo, se os tempos de resposta se tornarem elevados, os utilizadores tendem a procurar outros serviços.

As soluções arquiteturais mais tradicionais de desenvolvimento e *deployment* de aplicações não são as mais indicadas para este ritmo de inovação e melhorias. A adição de novas funcionalidades e o *deployment* de uma nova versão de uma aplicação que apresente uma arquitetura monolítica, não é possível de realizar da forma mais eficiente. Esta ineficiência está relacionada com os custos associados ao *deploy* de uma nova versão integral da aplicação, ao invés do *deploy* apenas do componente modificado. Para mais, quando

há necessidade de responder a um incremento no número de acessos dos clientes, é necessário escalar a totalidade da operação, ao invés de escalar apenas certos componentes de forma mais eficiente [16, 21].

Para tentar solucionar estes problemas surgiu o conceito de micro-serviços [16, 21, 24], uma nova arquitetura aplicacional que pretende capitalizar as vantagens da arquitetura e computação orientada a serviços (SOA/SOC, secção 2.1).

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

A computação na *Cloud* é um modelo de computação que permite o acesso partilhado a um conjunto de recursos como, por exemplo, armazenamento e capacidade de processamento, fornecidos como serviços [37]. Tem ainda como vantagem o aumento e a diminuição dos recursos alocados a uma aplicação de forma dinâmica, com base em métricas ou regras (por exemplo, a taxa média de utilização do CPU superior a 50%).

Devido ao tamanho reduzido de cada micro-serviço, é também possível aproveitar outros tipos de locais para realizar a computação, como a *edge*. A computação na *Edge* é um paradigma que permite que a computação seja realizada em dispositivos mais próximos do utilizador, libertando assim a infraestrutura da *cloud* de certos processamentos [63]. Estes dispositivos, por exemplo, *routers* e *switches*, quando comparados com os da *cloud*, são mais limitados a nível de recursos, existem em maior número e são heterogéneos entre si. É então possível migrar certos serviços da *cloud* para este tipo de dispositivos na *edge*, permitindo uma diminuição no tempo de acesso a esses serviços, traduzindo-se num melhor desempenho aplicacional, podendo ainda proporcionar uma redução nos custos. Essa migração possibilita também uma redução do volume dos dados transferidos entre os utilizadores e a *cloud*, se os serviços na *edge* fizerem uma filtragem e agregação dos dados produzidos.

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

1.2 Problema

A grande complexidade na gestão de aplicações compostas por micro-serviços em execução na *cloud* depende do número de micro-serviços que as constituem, bem como do número de utilizadores dos diferentes serviços e QoS esperada. Tal implica alterações na escalabilidade dos diferentes serviços. Numa aplicação que exija tempos de latência baixos das operações realizadas pelos utilizadores, a possibilidade de ter serviços disponíveis na *edge* permite que a comunicação entre os dispositivos dos utilizadores e o serviço seja mais rápida, bem como a filtragem de dados, diminuindo o volume dos dados a enviar para a *cloud* [63, 68].

A migração dos serviços para a *edge* implica, no entanto, gerir um maior número de locais de *deployment* possíveis, bem como atender à sua heterogeneidade [63, 68]. As decisões têm de ter em atenção os recursos desses nós, a sua localização face aos pedidos de acesso, o volume de dados em trânsito, assim como a possível replicação dos dados de um serviço (ex.: entre um *data center* na *cloud* e um nó na *edge*). No contexto da gestão dos micro-serviços, este trabalho foca-se nos problemas relacionados com a migração/replicação de serviços na *cloud* e na *edge*, seguindo uma estratégia incremental:

1. Migração/replicação dinâmica (em tempo de execução) de micro-serviços individuais, em ambiente de *cloud*, com o intuito de validar as funcionalidades de migração/replicação dos micro-serviços. Considera-se que não existem dependências do micro-serviço em relação a outros micro-serviços. Avaliação de quais as métricas adequadas a essa decisão de migração/replicação, de forma a garantir uma qualidade de serviço previamente definida;

2. Migração/replicação dinâmica de micro-serviços individuais, considerando uma infraestrutura híbrida composta por nós na *cloud* e na *edge*, com o objetivo de analisar o local de execução dos serviços aquando uma migração/replicação. Avaliação de quais as métricas adequadas tendo em conta os resultados obtidos no ponto 1 e verificação das diferenças nos tempos de acesso aos serviços através da utilização dos nós na *edge*.

Para abordar os sub-problemas identificados, é necessário ter em conta o problema da gestão dos recursos, tanto na *cloud*, como nos nós da *edge*. A gestão dos recursos tem que ser [14, 75]: 1. eficiente, ou seja, utilizar ao máximo os recursos de cada nó até um certo limite, de forma a não os sobrecarregar; 2. adaptável, de forma a suportar os diferentes ambientes, de *cloud* e *edge* (e vários tipos de dispositivos da *edge*); 3. garanta a execução dos micro-serviços em locais que possam efetivamente melhorar a QoS das aplicações.

O objetivo geral da gestão automática da migração/replicação de micro-serviços é possibilitar o melhor desempenho de uma aplicação ao reduzir a comunicação com a *cloud*, melhorando os tempos de resposta através de componentes localizados na *edge*, e reduzindo o volume de dados a transferir. Este objetivo é bastante importante para aplicações

que requeiram uma baixa latência nas operações. Pode também permitir que os custos das infraestruturas na *cloud* sejam reduzidos, devido à diminuição da sua utilização.

1.3 Solução proposta

Esta dissertação apresenta mecanismos de gestão automática que permitem a migração e replicação, na *cloud* e na *edge*, de micro-serviços (sem estado) que compõem uma dada aplicação. As métricas utilizadas nas decisões de migração/replicação incluem a taxa de utilização dos recursos (CPU, RAM, etc.) e a proveniência dos acessos. A solução permite ainda que no futuro venham a ser incluídas métricas adicionais, tais como a latência e o número de acessos.

As métricas são definidas sob a forma de regras, tendo cada uma a sua decisão (migração, replicação) associada. As regras podem ser configuradas ao nível da aplicação e/ou ao nível dos micro-serviços, onde a decisão diz respeito ao serviço que despoletou determinada regra. Existe ainda a possibilidade das regras serem alteradas em tempo de execução.

Para permitir a migração e replicação dos micro-serviços nos ambientes de *cloud* e *edge*, foi também necessário desenvolver mecanismos para realizar a gestão dos nós e dos seus recursos, de uma forma transparente para os utilizadores. Os nós da *edge* podem ser configurados e se o sistema considerar que a sua utilização é benéfica para o desempenho das aplicações, serão utilizados para a execução de micro-serviços.

Existem também métricas, referentes aos nós (na *cloud* e na *edge*), que podem ser utilizadas para decidir previamente a necessidade de acrescentar um novo nó para a execução de micro-serviços, pois os nós presentes podem ter atingindo valores de recursos (CPU, RAM) que se considera estarem a ficar sobrecarregados. O mesmo se procede quando um determinado nó tem valores de recursos que indicam uma baixa utilização, podendo-se proceder à sua remoção para diminuir custos. As métricas dos nós são utilizadas, como nas aplicações/micro-serviços, através de regras, que podem ser configuradas para todos os nós em geral e/ou para um nó em particular.

A solução tem como base algumas contribuições na área, tal como a computação osmótica (secção 2.5), a qual permite uma gestão dinâmica dos serviços entre a *cloud* e *edge*, sem degradar o desempenho da aplicação, bem como a solução proposta em 2.7.1, que tem como objetivo conferir mecanismos que permitam dotar os micro-serviços com funcionalidades elásticas (replicação) em tempo de execução.

A Figura 1.1 apresenta uma visão simplificada da arquitetura da solução e seus componentes. O componente de gestão de micro-serviços (*µServices Management*), o componente principal da solução, tal como representado na figura, é responsável pela migração e replicação dos micro-serviços. Esta migração/replicação é efetuada com base nas informações fornecidas pelos componentes de Monitorização (*Monitoring*), responsável pela obtenção de informação sobre os serviços em execução na *cloud* e em nós da *edge*.

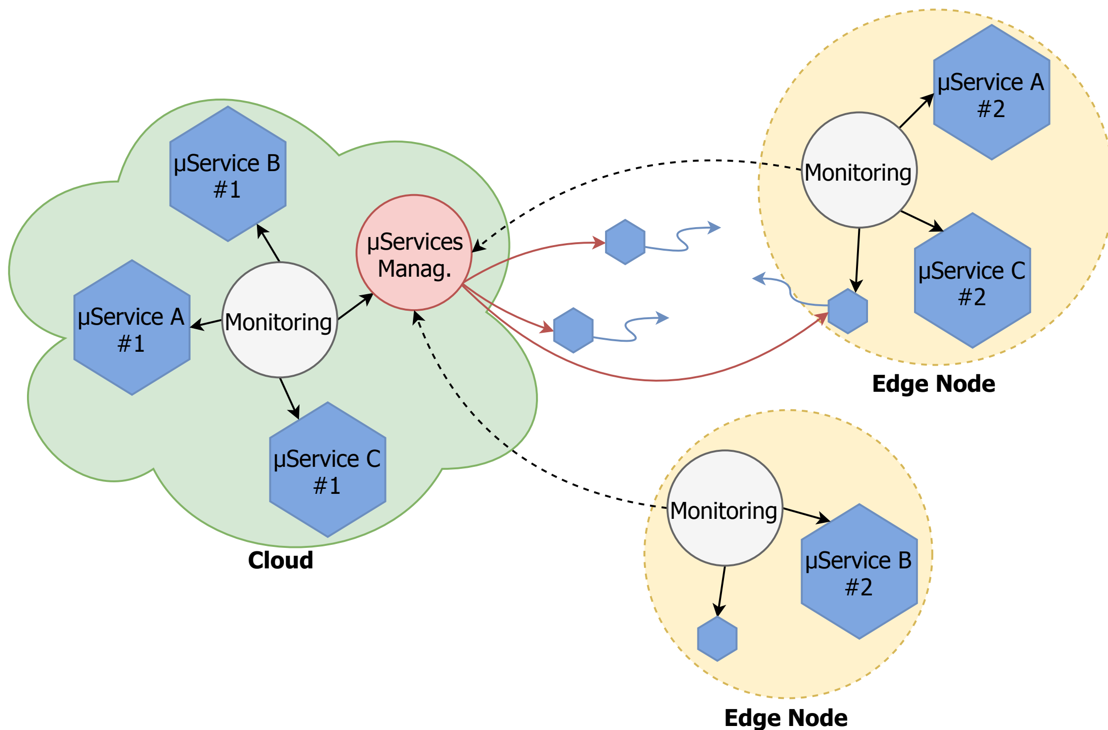


Figura 1.1: Visão simplificada da solução.

Tendo em vista a resolução dos sub-problemas identificados na secção 1.2, o trabalho começou por avaliar uma aplicação particular com uma arquitetura de micro-serviços, previamente desenvolvida, a *Sock Shop* [72], e que é um protótipo de uma aplicação web de comércio eletrónico de venda de meias. A *Sock Shop* foi escolhida tendo em conta que é *open-source* (uma outra possibilidade seria [11]), possui uma boa documentação e tem um nível de complexidade adequado para o problema que se pretende resolver.

1.4 Contribuições

A dissertação tem como objetivo desenvolver um sistema que incorpore mecanismos de gestão e coordenação automáticos de aplicações com arquitetura de micro-serviços, em concreto na migração/replicação dos serviços entre a *cloud* e a *edge*. As contribuições do trabalho desenvolvido são as seguintes:

1. Desenho de uma arquitetura modular que permita englobar mecanismos de gestão de aplicações de micro-serviços em ambientes heterogéneos, compostos pela *cloud* e *edge*;

2. Conceção de um protótipo aplicacional com as seguintes funcionalidades:

   a) Gestão e coordenação automáticas de aplicações de micro-serviços, no que diz respeito à migração/replicação dos serviços na *cloud* e na *edge*, com base em métricas e regras configuradas;

   b) Gestão dos ambientes de execução dos micro-serviços, isto é, a *cloud* e a *edge*, nomeadamente ao nível dos recursos, automaticamente para a execução dos micro-serviços;

   c) Suporte às aplicações de micro-serviços, a nível da comunicação entre serviços e balanceamento de carga dos micro-serviços.

3. Validação do protótipo, de forma a verificar a viabilidade dos mecanismos desenvolvidos, e os seus benefícios concretos.

1.5 Estrutura do documento

Seguindo o capítulo da Introdução, o capítulo 2 diz respeito ao estado da arte, onde são apresentados os conceitos relacionados com os que a solução aborda.

O capítulo 3 apresenta a arquitetura da solução de forma detalhada, explicando o funcionamento do sistema, os componentes necessários, para além do componente principal de gestão de micro-serviços, bem como as opções que foram tomadas em cada momento particular.

Em seguida, o capítulo 4 aprofunda os aspetos relevantes da implementação do sistema, detalhando também as tecnologias que foram utilizadas em cada um dos componentes desenvolvidos.

O capítulo 5, diz respeito à validação da solução, englobando uma breve descrição do caso de estudo utilizado, a aplicação *Sock Shop*, quais as alterações realizadas no micro-serviços da aplicação de forma a serem compatíveis com a arquitetura do sistema desenvolvido, e por fim a avaliação experimental.

Por fim, o capítulo 6 apresenta uma conclusão geral sobre os pontos mais relevantes desta dissertação, e algumas ideias que podem ser seguidas num trabalho futuro de forma a complementar e melhorar a solução.

Capítulo 2

Estado da arte

Este capítulo tem como objetivo a apresentação de conceitos, definições e abordagens relevantes para o desenvolvimento da solução proposta. A secção 2.2 aborda o conceito de micro-serviços, as vantagens e desvantagens da utilização, principais abordagens para o desenvolvimento de uma aplicação sob uma arquitetura de micro-serviços, como transitar de uma arquitetura mais clássica, como a monolítica, para esta arquitetura e alguns casos práticos. São ainda descritas algumas características de suporte à execução de micro-serviços no contexto de computação na *Cloud*, o qual é tratado com mais detalhe na secção seguinte. A secção 2.3 descreve as características principais da computação na *Cloud*, a gestão dos seus recursos, bem como alguns exemplos de infraestruturas públicas na *cloud*. A secção 2.4 refere-se à computação na *Edge*, explicando no que consiste, bem como as suas vantagens e os desafios que se levantam na sua utilização. A secção 2.5 aborda a problemática da computação numa infraestrutura heterogénea composta pela computação na *Cloud* e computação na *Edge*, e uma proposta recente neste domínio. A secção 2.6 analisa a tecnologia de *containers* e como esta pode ser utilizada para a execução de micro-serviços. A secção 2.7 apresenta alguns trabalhos relacionados, isto é, que incorporam certas funcionalidades e características semelhantes ao pretendido na solução.

2.1 Computação orientada a serviços - SOC

A computação orientada a serviços (*Service-Oriented Computing* - SOC) é um paradigma de programação, que tem como base as propostas de *design* seguidas pela arquitetura orientada a serviços (*Service-Oriented Architecture* - SOA). A sua visão é obter uma infraestrutura flexível e disponibilizar como serviços, funcionalidades aplicacionais independentes e reutilizáveis, separando as funcionalidades em três camadas: na inferior encontram-se

os serviços base, na camada intermédia a composição de serviços e no topo a gestão e monitorização de serviços [44].

Os serviços são componentes autónomos e *loosely coupled*, que são usados no paradigma SOC para possibilitar um desenvolvimento mais rápido e barato de aplicações distribuídas [44]. Este paradigma emergiu devido à necessidade de dar resposta à evolução de como as empresas pretendiam gerir os seus negócios - ao invés de existirem apenas aplicações ("serviços") desenvolvidas por uma única empresa. O paradigma de serviço permite que várias empresas desenvolvam as suas aplicações, e que estas comuniquem e interajam através de *Web services*. Um *Web service* é um tipo de serviço específico que é identificado por um *Uniform Resource Identifier* (URI) [43].

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

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

- **Coreografia**: os serviços trocam mensagens entre eles, sendo que são definidas regras para a sua interação.

Este tipo de serviços pode ser desenvolvido utilizando duas soluções principais: *Web services* SOAP e *Web services* RESTful [62].

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

A BPEL/BPEL4WS (*Business Process Execution Language for Web Services*) define uma linguagem para a composição de serviços numa forma de processos de negócio, em que cada composição é um processo de negócio ou um *workflow* que interage com um conjunto de *Web services* para atingir um determinado objetivo. A sua composição é denominada de processo e os serviços com o qual este interage são denominados de parceiros, sendo que um processo, como qualquer *Web service*, suporta um conjunto de interfaces WSDL que possibilita a troca de mensagens com os seus parceiros. A interação entre a composição BPEL e os seus parceiros é, no caso geral, *peer-to-peer*, em que cada parte invoca operações das interfaces públicas de cada um. Certas aplicações podem apenas utilizar um processo como um serviço sem fornecer qualquer funcionalidade, enquanto outros podem ser utilizados pelo processo como serviços [12].

Os *Web services* RESTful são uma tecnologia mais recente, que surgiu como uma alternativa mais simples e leve comparando com os *Web services* SOAP, e que foram idealizados

para facilitar o seu acesso e a composição de vários serviços [25]. Os serviços seguem a arquitetura *Representational State Transfer* (REST), a qual fornece um conjunto de restrições arquiteturais que se focam na escalabilidade dos componentes, no seu *deployment* independente e na implementação de interfaces genéricas [18]. Este tipo de serviços usa os métodos (PUT, POST, GET e DELETE) de invocação remota HTTP que são aplicáveis a qualquer recurso através do seu URI [25]. De acordo com [25], existem 4 propriedades que estes serviços apresentam:

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

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

1. Utilização de interfaces uniformes imutáveis, sendo que não existe o problema da quebra de acessos nos clientes [45]. 2. Protocolo suportado pela maior parte dos clientes, visto não ser necessário qualquer tipo de *middleware*, já que as invocações são feitas por HTTP [45]. 3. As mensagens SOAP são bastantes descritivas, produzindo maior tráfego na rede devido seu tamanho e causando uma maior latência quando comparado com *Web services* RESTful [40]. 4. A codificação e descodificação de mensagens SOAP consomem mais recursos, provocando *overheads* visíveis no seu desempenho [40].

Num teste de performance, realizado num ambiente de computação móvel que comparou *Web services* SOAP e *Web services* RESTful, foi possível concluir [40]:

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

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

[1] *Stateless* significa que todos os pedidos devem ter a informação necessária para o seu processamento, e não pode ser armazenado qualquer contexto no servidor, fazendo com que todo o estado da sessão seja mantido no cliente [18].

2.2 Micro-serviços

O termo micro-serviço refere-se a um pequeno processo independente que implementa uma dada funcionalidade de forma completa, podendo funcionar autonomamente sem recurso a outro tipo de serviço. O estilo arquitetural de micro-serviços baseia-se no desenho de aplicações em que um conjunto de micro-serviços independentes "cooperam" entre si de forma a fornecer certas funcionalidades, interagindo através de mensagens enviadas pela rede. Sendo que cada micro-serviço é independente, isto permite que cada um deles possa ser desenvolvido num tipo de tecnologia específica, não existindo por isso qualquer tipo de limitação à linguagem de programação, plataforma de desenvolvimento ou *framework* que é utilizada [16, 21].

Para melhor compreensão deste estilo arquitetural, bem como as vantagens e desvantagens inerentes, é relevante explicar um estilo arquitetural mais tradicional, o monolítico. Uma aplicação monolítica é desenvolvida como uma unidade única e é organizada em módulos dependentes da aplicação. Estes módulos não podem assim ser reutilizados por outra aplicação [16]. Um exemplo clássico da utilização deste tipo de arquitetura são as aplicações (Web) compostas por uma interface de utilizador (constituída por ficheiros HTML e Javascript), uma base de dados e a aplicação/ lógica do lado do servidor. Quando é necessário ser realizada qualquer alteração, tem de ser feito o *deploy* completo da aplicação, mesmo que seja uma alteração mínima num componente específico [21]. Para além deste inconveniente, à medida que a aplicação vai evoluindo e crescendo, torna-se difícil manter uma boa estrutura modular, dificultando as alterações de um dado módulo apenas dentro deste sem afetar os outros.

Estas são algumas das desvantagens que um sistema desenvolvido sob o estilo monolítico pode apresentar, e é aqui que começou a suscitar maior interesse o estilo arquitetural de micro-serviços, de forma a tentar ultrapassar este tipo de dificuldades. É possível avaliar os dois estilos arquiteturais, para uma mesma aplicação, nas figuras 2.1 e 2.2.

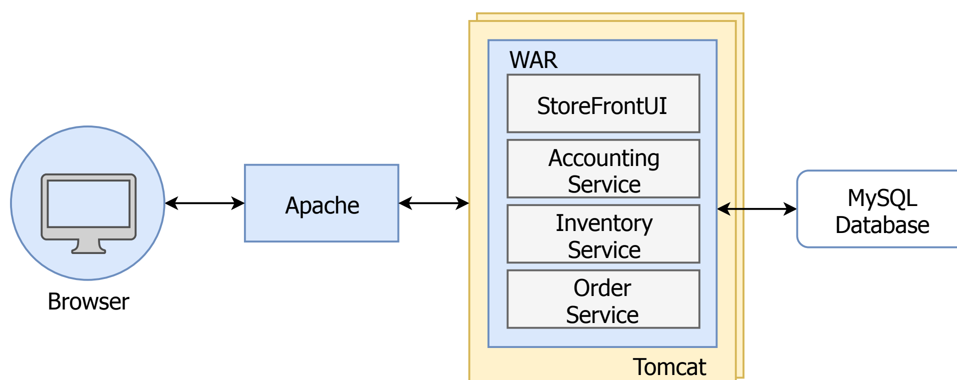


Figura 2.1: Diagrama de uma aplicação monolítica. Adaptado de [53].

A Figura 2.1 mostra quais os componentes de uma aplicação de comércio online que permite encomendas de clientes, verifica o inventário e o crédito disponível, e processa

as encomendas. A aplicação é composta por vários componentes: 'StoreFrontUI' que implementa a interface e os restantes componentes de *backend*. A aplicação é implementada através de uma arquitetura monolítica, num único ficheiro WAR que corre num *web container* Tomcat, estando também conectada a uma única base de dados [53].

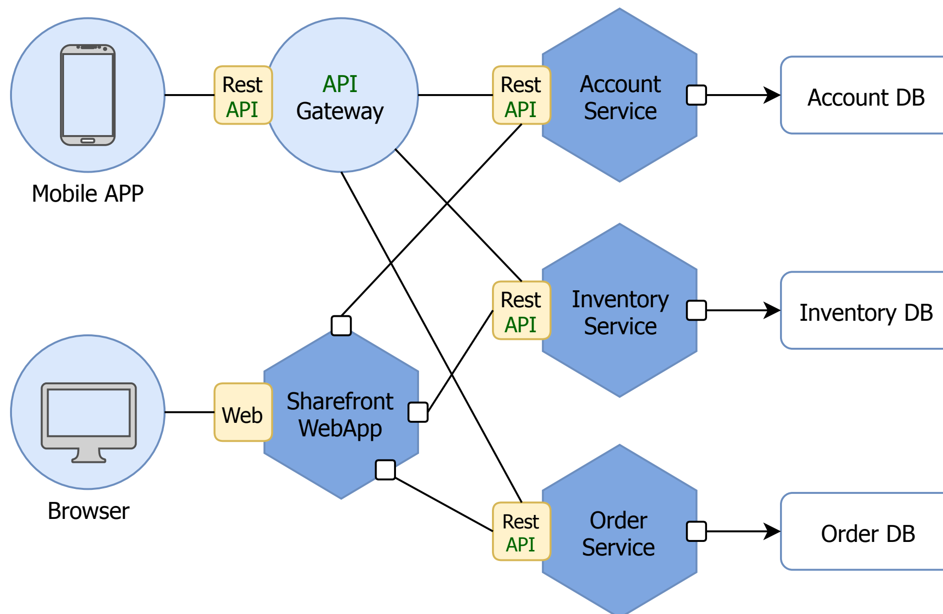


Figura 2.2: Diagrama de uma aplicação em micro-serviços. Adaptado de [52].

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

2.2.1 SOA e micro-serviços

Os micro-serviços podem ser vistos como uma evolução da arquitetura orientada a serviços (*Service-Oriented Architecture* - SOA) em que, apesar de diferirem, partilham alguns aspetos [24].

A SOA surgiu como uma forma de mitigar os problemas e desafios das grandes aplicações monolíticas [41]. A sua visão é obter uma infraestrutura flexível e disponibilizar como serviços funcionalidades aplicacionais independentes e reutilizáveis [44]. Tem também como objetivo facilitar a manutenção do software, visto que é possível substituir um serviço por outro, se a semântica do novo serviço não sofrer uma grande alteração [41].

Uma das grandes diferenças entre a arquitetura de micro-serviços e a SOA, é a granularidade de cada serviço. Nos micro-serviços, os serviços fornecem apenas uma única funcionalidade, tendo por isso uma granularidade fina, enquanto que na SOA estes tendem a

englobar uma funcionalidade do negócio, por vezes implementada como um subsistema completo. Outra diferença é que na SOA existe o conceito de maximizar a partilha de componentes, e numa arquitetura de micro-serviços o objetivo é minimizar essa partilha [48].

Assim os micro-serviços emergiram da sua utilização real, tirando partido da melhor compreensão de sistemas e arquiteturas, como uma forma de melhorar a SOA [41].

2.2.2 Vantagens e desvantagens da arquitetura de micro-serviços

Em baixo são descritas algumas das vantagens da utilização da arquitetura de micro-serviços em relação a uma arquitetura mais tradicional, como a monolítica [22]:

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

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

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

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

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

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

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

Embora existam vantagens, existem também determinados inconvenientes associados à utilização de micro-serviços [22]:

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

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

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

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

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

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

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

2.2.3 Arquitetura de micro-serviços

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

Esta secção está organizada de acordo com a taxonomia representada na Figura 2.3, adaptada de [24], sendo que certos tópicos na figura, estando relacionados, serão abordados em conjunto. Algumas abordagens descritas, como em termos de comunicação, gestão de dados, descoberta de serviços, entre outras, são um subconjunto baseado em [60].

2.2.3.1 Design

Abordagem (*Approach*) No desenvolvimento de uma aplicação com uma arquitetura de micro-serviços, a existência (ou inexistência) de software antigo que deve transitar para esta arquitetura influencia a abordagem inicial a ser seguida, podendo ser **Brownfield**, caso exista a necessidade de migrar software antigo, ou **Greenfield** caso não seja necessária a integração com software existente [24, 28].

Suporte Arquitetónico (*Architectonic Support*) A arquitetura de um sistema tem de ser idealizada de forma a cumprir as funcionalidades e restrições definidas, sendo que para isto é necessário seguir certas metodologias. Por exemplo, a utilização de uma **Arquitetura de Referência** ou a **Arquitetura orientada ao Modelo** (*Model-Driven Architecture* - MDA) podem auxiliar neste processo [24].

2.2.3.2 Implementação (*Implementation*)

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

Desenvolvimento de Funcionalidades - *Frameworks* Um micro-serviço implementa um conjunto de funcionalidades e o seu desenvolvimento beneficia da utilização de uma *framework* que ofereça algumas funcionalidades base, permitindo assim acelerar o seu desenvolvimento. Segue-se a descrição de um conjunto popular de *frameworks* que auxiliam o desenvolvimento de micro-serviços, tendo como base a análise apresentada em [23].

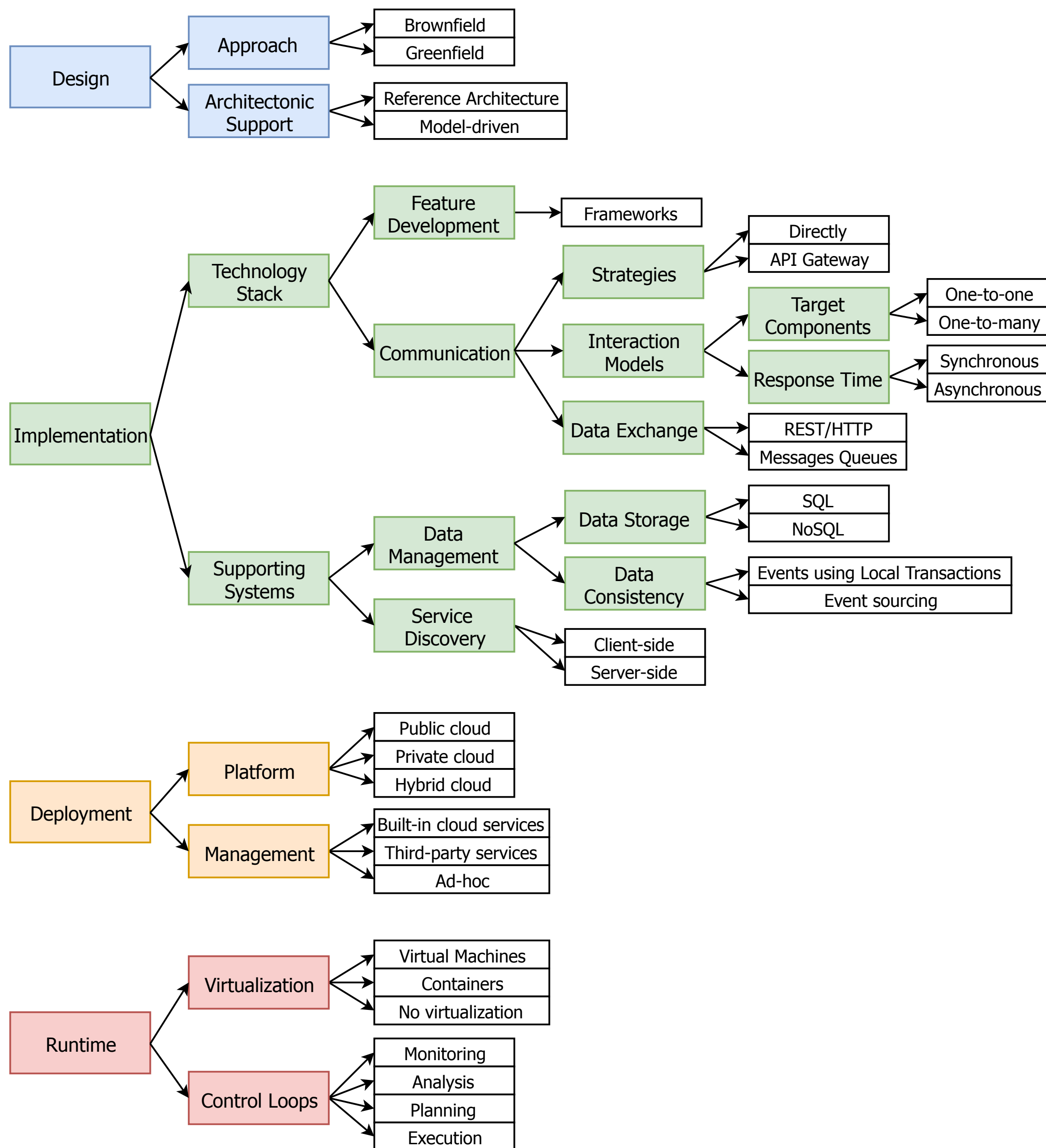


Figura 2.3: Taxonomia da arquitetura de micro-serviços. Adaptado de [24].

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

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

- **Light Rest 4J**: é uma *framework* de APIs REST construída com base na light-4j, também uma *framework* de micro-serviços rápida, leve e nativa. O Light Rest 4J é umas das *frameworks* Java mais rápidas de micro-serviços [47].

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

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

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

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

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

1. **Pedidos feitos diretamente aos micro-serviços**: neste caso, o cliente faz pedidos diretamente aos micro-serviços necessários, por exemplo através de um *endpoint* público que mapeia no *load balancer* [2] de cada micro-serviço. Os pedidos são então distribuídos pelas várias instâncias desse micro-serviço, caso existam [60].

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

   - Devido à granularidade fina que os micro-serviços apresentam, o cliente pode ter que fazer múltiplos pedidos a diferentes micro-serviços de forma a obter todos os dados pretendidos, o que pode levar a um aumento da latência da operação.

[2] Um *load balancer* tem como objetivo encaminhar os pedidos dos clientes para servidores capazes de responder a esses pedidos, de forma a maximizar a velocidade e a capacidade de utilização e de distribuir a carga pelo sistema [42].

- Caso exista a necessidade de uma modificação num micro-serviço e que, posteriormente, leve à separação em dois micro-serviços distintos, a modificação é bastante difícil uma vez que o cliente utiliza diretamente o *endpoint*.

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

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

   Porém, existe a possibilidade de se criar um *bottleneck* no desenvolvimento da API Gateway, uma vez que esta tem de ser atualizada para expor corretamente os *endpoints* dos micro-serviços [49, 60].

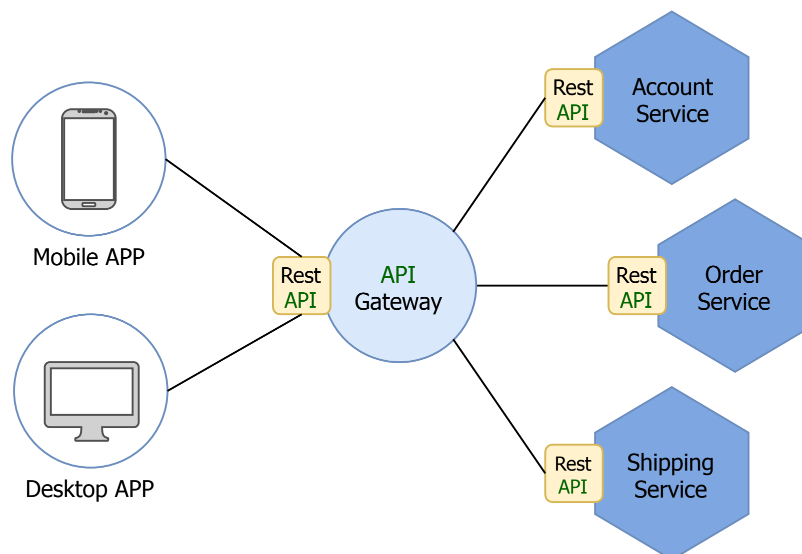


Figura 2.4: Utilização de uma API Gateway com micro-serviços. Adaptado de [60].

De um modo geral, a utilização de uma API Gateway é considerada uma melhor abordagem no desenvolvimento das aplicações do que a comunicação direta com os micro-serviços [60].

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

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

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

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

Tabela 2.2: Estilos de IPC. Adaptado de [60].

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

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

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

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

A abordagem com recurso a mensagens permite que a comunicação seja realizada de forma assíncrona, levando a que o serviço/cliente que comunica não fique bloqueado à espera da resposta (caso exista). Permite ainda que as mensagens fiquem armazenadas

num *buffer*, sendo que o serviço que as recebe pode processá-las à medida que vá tendo disponibilidade. Tal garante que, eventualmente, todas as mensagens recebidas sejam processadas, mesmo que de forma mais lenta.

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

Consoante as necessidades do sistema, estes dois tipos de IPC podem ser combinados, tirando proveito dos benefícios de cada um [60].

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

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

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

Apesar dos benefícios, há um nível de complexidade maior quando é necessário consultar ou atualizar dados que estão dispersos nas diferentes bases de dados, pois não é possível fazer *queries* a base de dados externas ao próprio micro-serviço, tendo tal que ser realizado com recurso a APIs. Problemas neste tipo de arquitetura incluem a implementação de transações com consistência dos dados dos vários micro-serviços, e realizar *queries* que devolvam dados de múltiplos micro-serviços e que usam tipos distintos de bases de dados (SQL e NoSQL).

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

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

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

Este problema de consistência é derivado da falta de transações tradicionais, tendo que ser implementados mecanismos que garantam que as duas operações de alteração e subsequente notificação sejam vistas como uma operação atómica, ou seja, sejam executadas como uma só do ponto de vista externo do sistema [60].

Este problema pode ser ultrapassado através das seguintes abordagens:

- **Publicar eventos com recurso a transações locais**: é necessário ter uma tabela auxiliar, em cada micro-serviço, onde vão sendo escritos os eventos quando se efetua alguma atualização numa dada tabela. Estas duas operações são feitas dentro de uma transação, garantido que ambas são escritas, ou nenhuma delas é. Existe também um processo que vai fazendo *queries* à tabela de eventos e os publica no *Message Broker*. Esta abordagem tem limitações, não podendo ser utilizada em base de dados NoSQL, dado que estas (tipicamente) não têm o conceito de transação [55, 60].

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

  Apesar de resolver o problema da consistência dos dados, o *event sourcing* tem desvantagens para fazer consultas aos dados, tendo por isso que ser usado *command query responsability separation* (CQRS) [51, 60]. O CQRS é uma arquitetura em que são utilizadas diferentes base de dados para operações de leitura e escrita.

  Após ser inserido algum valor (evento) no *Event Store*, a base de dados de leitura pode ser atualizada de forma síncrona ou assíncrona. Se for realizada de forma síncrona é possível oferecer consistência forte ou consistência eventual. Se for realizada de forma assíncrona será consistência eventual [74].

Descoberta de serviços (*Service Discovery*) Num sistema com a arquitetura de micro-serviços, como foi referido anteriormente (secção 2.2.3.2), muitas das vezes existe a necessidade de comunicação entre vários micro-serviços. Ao contrário de uma aplicação monolítica, onde os recursos têm uma localização estática (ou muda muito raramente ao longo do tempo), numa arquitetura deste género, a localização dos recursos é atribuída dinamicamente devido, por exemplo, a falhas ou a atualizações [60].

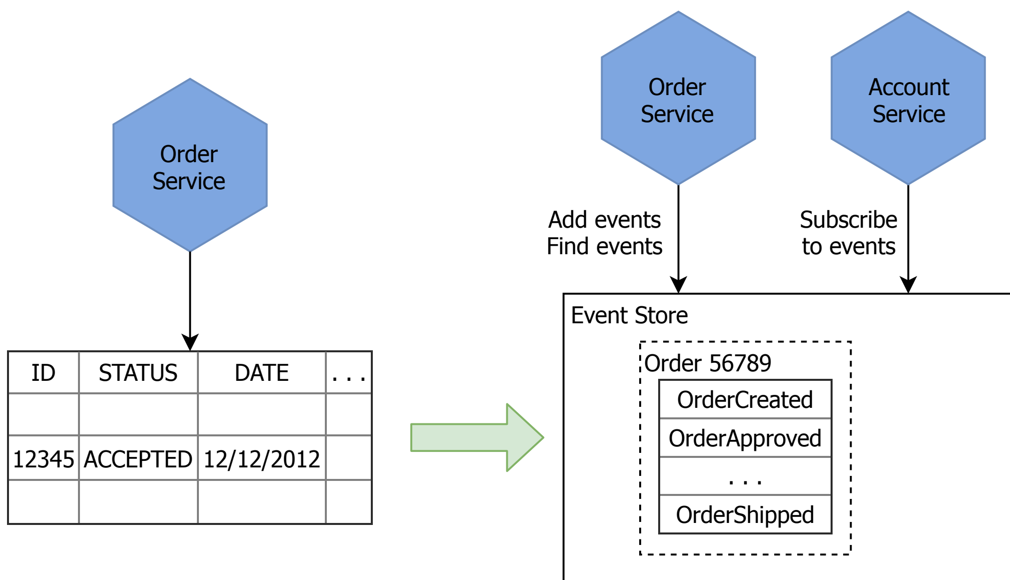


Figura 2.5: Utilização de *event sourcing*. Adaptado de [60].

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

Ambos os padrões para localizarem serviços necessitam de um componente denominado *service registry*. Este é um componente que tem uma base de dados, onde são guardadas as localizações dos micro-serviços, posteriormente contactado pelos clientes ou outros micro-serviços quando os necessitam. Existem duas formas principais em que os serviços são registados e eliminados do *service registry*: os micro-serviços podem registar-se eles mesmos, por exemplo, através de uma API REST e atualizar a sua localização, bem como eliminá-la; ou pode existir outro componente que analisa os micro-serviços e os vai registando no *service registry* [59, 60].

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

  Este padrão tem o inconveniente de fazer com que o cliente fique dependente do *service registry*, tendo que ser implementada uma lógica diferente para clientes diferentes (usando distintas linguagens de programação e *frameworks*) [50, 60].

- **Descoberta do lado do servidor**: o cliente ou o serviço contactam o *load balancer*, em que este faz uma pesquisa no *service registry* para obter a localização do recurso, e em seguida redireciona o pedido para o serviço correspondente. Este padrão tem o benefício de abstrair o cliente do *service registry*, dado que o cliente apenas contacta o *load balancer*. Apesar disso, apresenta o senão de ter que se implementar e gerir um *load balancer*, um componente que tem de estar sempre disponível [56, 60].

2.2.3.3 *Deployment*

Plataforma (*Platform*) A arquitetura de micro-serviços é um exemplo de utilização de *cloud-native*, que consiste em seguir abordagens de desenho, desenvolvimento e execução de aplicações para ambientes modernos e dinâmicos como a *cloud* (nos seus vários tipos: *public cloud*, *private cloud* ou *hybrid cloud*, descritas na secção 2.3.1). Estas técnicas permitem obter sistemas mais desacoplados, resilientes, administráveis e observáveis. O objetivo geral é permitir modificações de forma mais frequente e melhorar a escalabilidade [5, 13, 20, 32].

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

Esta gestão pode ser conseguida através de soluções já fornecidas pela própria *cloud*, ferramentas externas ou com o desenvolvimento de mecanismos próprios para uma aplicação. Por exemplo, os principais fornecedores de serviços na *cloud*, como a Amazon, Microsoft e Google disponibilizam funcionalidades como o *auto scaling*, permitindo que haja automaticamente um aumento ou uma diminuição de recursos em determinados serviços, conforme as necessidades. Em relação a ferramentas externas aos fornecedores de serviços existe, por exemplo, a RightScale [3] que disponibiliza funcionalidades para a gestão dos recursos na *cloud* [24].

2.2.3.4 Tempo de execução (*Runtime*)

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

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

- **Uma instância de serviço por *Host***: uma única instância de um serviço encontra-se isolada no seu próprio *Host*. Sendo que existem dois tipos específicos para este tipo de padrão:
  - **Instância de serviço por máquina virtual**: cada serviço é instalado numa VM e posteriormente é feito o *deployment* dessa VM.

    Um dos benefícios desta abordagem é o isolamento completo de cada serviço em relação aos outros serviços, possibilitando a definição dos recursos (CPU,

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

RAM) que cada VM pode utilizar. Permite também, ao utilizar uma infraestrutura na *cloud*, o recurso a estratégias disponíveis de *load balancing* (balanceamento da carga) e *auto scaling* (escalonamento automático) das máquinas virtuais.

Apesar destas vantagens, este padrão tem o inconveniente de ser menos eficiente em termos de recursos quando comparado com o padrão de múltiplas instâncias (referido no próximo ponto), pois tem o custo adicional de criar a VM, o que inclui o sistema operativo. Fazer o *deployment* de uma nova versão de um serviço também é um pouco lento, uma vez que tem de se instalar o serviço na VM [58, 60].

– **Instância de serviço por *container***: cada serviço corre no seu próprio *container* [4]. Os serviços são instalados como uma imagem de um *container*, depois podem ser lançados vários *containers* numa máquina (física ou virtual). Este padrão tem vantagens similares ao padrão de uma instância por máquina virtual, mas é uma tecnologia mais leve e rápida, sendo bastante célere a iniciar um serviço. Contudo, existem desvantagens na utilização deste padrão. Ao contrário do padrão de uma instância por VM, no padrão de *containers* existe a partilha do núcleo do sistema operativo entre os *containers*, tornando-o menos seguro [57, 60].

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

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

  Porém, também existem desvantagens quando se utiliza este padrão. Ao ter instâncias de serviços diferentes na mesma máquina, estas não se encontram totalmente isoladas umas das outras, podendo haver conflitos nos recursos e no software que cada uma utiliza. Por outro lado, também é difícil limitar os recursos que cada serviço pode utilizar e, consequentemente, monitorizar os recursos utilizados por cada um deles [54, 60].

[4] Um *container* é um mecanismo de virtualização ao nível do sistema operativo, que permite a existência de múltiplas instâncias isoladas. Um *container* permite isolar o código, contendo bibliotecas e as definições num único pacote, minimizando conflitos entre diferentes softwares que se encontram na mesma infraestrutura [15].

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

O exemplo seguinte, que consiste na utilização de um micro-serviço para a deteção de anomalias e para a análise das suas causas em aplicações móveis e IoT, descrito em [4] é uma das abordagens possíveis.

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

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

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

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

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

2.2.4 Transição de uma aplicação monolítica para micro-serviços

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

A utilização de metodologias incrementais consiste, por exemplo, em ir adicionando novas funcionalidades e em criar extensões de funcionalidades já existentes em forma de micro-serviços, modificando a aplicação monolítica de uma forma complementar. Ao logo do tempo, as funcionalidades implementadas pela aplicação monolítica vão diminuindo até desaparecem ou poderem ser migradas para um novo micro-serviço.

Existem várias estratégias para refatorizar uma aplicação em micro-serviços [60]:

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

   Com esta estratégia é necessário adicionar dois componentes:

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

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

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

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

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

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

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

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

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

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

2.2.5 Utilização da arquitetura de micro-serviços na prática

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

2.2.5.1 Yelp

A Yelp [5] é o exemplo de uma empresa que fez a transição de uma aplicação monolítica para uma aplicação com uma arquitetura baseada em serviços. Esta mudança foi necessária devido ao facto do código da aplicação monolítica, 'yelp-main', ser demasiado extenso (vários milhões de linhas) e da equipa ter aumentado, levando a um aumento do tempo de resposta na efetivação das modificações ou na implementação de novo código.

Foi referido em [29] que a aplicação monolítica, 'yelp-main' ainda possui bastante código, sendo difícil que a mesma desapareça, embora tenha passado a ser mais uma aplicação de *frontend*, responsável pela agregação e visualização de dados dos serviços de *backend*.

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

Este problema foi resolvido através do Swagger [6] que fornece uma linguagem para documentar a interface dos serviços HTTP/JSON. Foi utilizado o swagger-py para criar automaticamente um cliente em Python para o mesmo serviço através de uma especificação Swagger. Recorreu-se ainda ao Swagger UI para fornecer uma diretoria centralizada para as interfaces dos serviços, permitindo uma facilidade em termos de descoberta da disponibilidade dos serviços.

[5] Yelp: `https://www.yelp.com/`
[6] Swagger: `https://swagger.io/`

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

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

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

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

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

Estes problemas foram resolvidos através da mudança para um sistema de descoberta de serviços baseado no SmartStack. Cada cliente tem uma instância HAProxy que está ligada ao localhost, que é configurado dinamicamente a partir das informações armazenadas no ZooKeeper [7]. Um cliente pode depois utilizar um serviço ligando-se ao *load balancer* do localhost, que redireciona o pedido para uma instância do serviço. Esta abordagem provou que é altamente fiável e a maior parte dos serviços de produção está a utilizá-la [29].

2.2.5.2 HubSpot

A HubSpot [8] desenvolve uma plataforma de marketing e vendas para fornecer uma maior presença online dos seus clientes.

Cada equipa é composta por um líder técnico e dois *developers* que trabalham com um gestor de produto e um designer. As equipas são pequenas para evitar problemas de

[7] ZooKeeper: é um serviço centralizado para manter informações de configuração, nomes, fornecer sincronização distribuída e serviços de grupo [67].

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

escalamento e demasiadas comunicações, permitindo também que o líder técnico esteja focado no produto e possa auxiliar os *developers*.

Os produtos são compostos por mais de 300 serviços e dezenas de aplicações de *frontend*. A maior parte dos micro-serviços são desenvolvidos em Java com a *framework* Dropwizard, e a parte de *frontend* utiliza Backbone e React em CoffeScript. Os serviços comunicam através de APIs REST ou de um sistema de mensagens como o Apache Kafka. Esta arquitetura permite iterações rápidas, dado que existem mais de 1000 unidades que podem ser *deployed* separadamente, permitindo que escalem independentemente.

Existe uma complexidade adicional para perceber qual o desempenho de uma arquitetura distribuída, bem como em gerir determinadas configurações (ex.: serviços a utilizarem versões diferentes de uma biblioteca partilhada). A mudança de uma aplicação monolítica em equipas pequenas pode não ter benefícios até que o código atinja uma certa dimensão ou o número de *developers* seja significativo.

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

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

O EC2 da Amazon tem sido bastante utilizado pela HubSpot. Utiliza o Apache Mesos [9] e uma aplicação que desenvolveu, denominada Singularity, que permite gerir os *clusters* EC2 de uma forma mais otimizada, permitindo uma maior densidade de serviços por servidor, reduzindo custos. Também permite que os *developers* não tenham intervenção nas instâncias, sendo a plataforma a lidar com determinados problemas [8].

2.3 *Cloud Computing*

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

A computação em *Cloud* foi precedida, na década de 1990, pela computação em Grid (*Grid Computing*) que permitia aos utilizadores obter computação a pedido para resolver problemas científicos complexos, através do acesso a recursos computacionais [19]. A

[9] Apache Mesos: http://mesos.apache.org/

computação em *Cloud* diferiu desse modelo ao utilizar tecnologias de virtualização a nível aplicacional e hardware, para fornecer e partilhar recursos de uma forma dinâmica [76], possibilitando um modelo de negócio vantajoso para grandes empresas como a Amazon, Google, Microsoft, etc.

De acordo com *The National Institute of Standards and Technology* (NIST), *Cloud Computing* é um modelo de computação que permite, de uma forma prática, o acesso a um conjunto de recursos de computação partilhados (por exemplo, redes, servidores, armazenamento, aplicações e serviços) que podem rapidamente ser configurados e fornecidos com o mínimo esforço de gestão ou interação do fornecedor de serviços [37].

O modelo de computação em *Cloud* fornece algumas funcionalidades distintas/únicas [37]:

- ***Self-service* e a pedido**: um utilizador pode obter capacidades computacionais consoante as suas necessidades, de forma automática, sem a necessidade de intervenção humana por parte do fornecedor de serviço.

- **Acessível pela rede**: os serviços estão disponíveis na rede e são acessíveis através de mecanismos standards com o recurso a smartphones, tablets ou computadores.

- **Agrupamento dos recursos**: os recursos computacionais do fornecedor são agrupados num modelo *multi-tenant* com diferentes recursos físicos e virtuais que são atribuídos dinamicamente consoante os pedidos dos utilizadores. Na maior parte das vezes o utilizador não tem controlo sobre a localização exata dos recursos, podendo em alguns casos especificar a localização num nível de abstração mais alto (por exemplo, ao nível do país).

- **Elasticidade rápida**: as capacidades podem ser requisitadas e libertadas elasticamente (por vezes de forma automática), para se escalar rapidamente de acordo com a necessidade. Para o utilizador, as capacidades disponíveis dos recursos parecem ser ilimitadas e podem ser definidas em qualquer momento.

- **Serviço monitorizado**: o uso dos recursos (armazenamento, processamento, largura de banda, entre outros) pode ser monitorizado, possibilitando que os sistemas de *cloud* controlem e otimizem automaticamente a sua utilização.

O modelo de negócio que a computação em *Cloud* utiliza é um modelo orientado ao serviço, ou seja, os recursos, quer a nível de hardware e/ou software, são disponibilizados como serviços que o utilizador requisita quando necessitar [76]. Os serviços podem ser agrupados em 3 categorias, consoante o tipo de serviço que é fornecido ao utilizador [37, 76]:

- ***Infrastructure as a Service* (IaaS)**: são fornecidos ao utilizador recursos de infraestruturas, como processamento, armazenamento, redes, normalmente através de

máquinas virtuais (VMs), onde pode ser instalado e executado qualquer tipo de software, desde sistemas operativos a aplicações.

- ***Platform as a Service*** **(PaaS)**: são fornecidas capacidades para o utilizador realizar o desenvolvimento e *deployment* de aplicações utilizando linguagens de programação, *frameworks* e bibliotecas suportadas pelo fornecedor do serviço.

- ***Software as a Service*** **(SaaS)**: é fornecido ao utilizador a capacidade de executar aplicações através da Internet, em que estas podem ser acedidas através de um browser ou da instalação de uma aplicação cliente.

2.3.1 Modelo de *Deployment*

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

Os principais modelos de *deployment* consistem em [37, 76]:

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

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

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

2.3.2 Virtualização de recursos

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

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

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

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

- A **virtualização completa** consiste numa máquina virtual em que esta é executada numa cópia exata do hardware. Tal só é possível quando a abstração do hardware fornecida pela VMM [10] é uma réplica exata do hardware físico. Neste caso, qualquer sistema operativo é executado no hardware sem modificações, desde que possua uma arquitetura que seja virtualizável. As VMMs VMware são exemplos de virtualização completa.

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

  Existem razões para que seja adotada a para-virtualização:

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

  Este tipo de virtualização é também considerado, pois existem algumas arquiteturas, como x86, que não são facilmente virtualizáveis [35].

2.3.3 Gestão de recursos e agendamento

A gestão de recursos é uma funcionalidade essencial em qualquer sistema, e está relacionada com três critérios base de avaliação de um sistema: performance, funcionalidade e custo. Uma gestão de recursos ineficiente impacta negativamente e de forma direta a performance e o custo do sistema, e indiretamente a sua funcionalidade [36].

2.3.3.1 Gestão de recursos

Uma *cloud* é um sistema com um número bastante considerável de recursos partilhados que são sujeitos a pedidos inesperados. Assim a gestão destes recursos requer políticas complexas e decisões para a sua otimização [36].

As estratégias para a gestão de recursos diferem consoante o tipo de serviço fornecido: IaaS, PaaS e SaaS. Em todos os casos, os fornecedores de serviços são afetados com flutuações de carga, um comportamento que desafia a elasticidade da *cloud*. Em certos

[10] Uma *Virtual Machine Monitor* (VMM) é um software que permite a criação e a gestão de máquinas virtuais (VMs), sendo também conhecida por *Virtual Machine Manager* e *Hypervisor* [66].

casos, quando um pico de carga pode ser previsto, os recursos podem ser provisionados à priori. Em casos em que não são esperados picos de carga, a situação é mais complicada, podendo ser utilizado o *auto scaling* se: 1. Existir um conjunto de recursos que podem ser libertados e alocados a pedido; 2. Existir um sistema de monitorização que permita decidir em tempo real a realocação de recursos. O *auto scaling* é suportado em serviços PaaS, mas é mais complicado de implementar em IaaS devido à falta de standards.

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

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

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

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

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

2.3.3.2 Agendamento

O agendamento é uma funcionalidade crítica na gestão de recursos na *cloud*. Este é responsável pela partilha e multiplexagem a vários níveis. Um servidor pode ser partilhado entre várias máquinas virtuais, em que cada VM pode suportar várias aplicações, e cada aplicação pode consistir em múltiplos *threads*. O agendamento de CPU suporta a virtualização de um processador, em que cada *thread* atua como um processador virtual; um link de comunicação pode ser multiplexado entre vários canais virtuais, um para cada fluxo [36].

Um algoritmo de agendamento, para além de cumprir os seus objetivos, deve ser eficiente, justo e *starvation-free*. Por exemplo, os objetivos de um *scheduler* para um sistema de *batch* são maximizar a taxa de transferência (o número de trabalhos completos por

unidade de tempo) e minimizar o tempo de resposta (o tempo entre a submissão de um trabalho e a sua conclusão).

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

2.3.4 Infraestruturas de *cloud*

Existem bastantes ofertas no que diz respeito a infraestruturas de *cloud* públicas, sendo que as mais utilizadas são as da Amazon, Microsoft e Google, sendo esperado que tenham 76% da receita neste mercado em 2018 [9].

A descrição que se segue ilustra a diversidade de recursos disponibilizados pelos fornecedores de *cloud* principais (a Amazon com a Amazon Web Services [11] [AWS], a Microsoft com o Azure [12] e a Google com a Google Cloud Platform [13] [GCP]) e que permite que diferentes micro-serviços possam ser desenvolvidos, e depois executados, usando tecnologias diversas e que sejam as mais adequadas às funcionalidades pretendidas.

2.3.4.1 Computação

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

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

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

O Microsoft Azure permite *auto scaling* através de Virtual Machine Scale Sets. Tem ainda dois serviços de *containers*, o Azure Container Service que é baseado no Kubernetes e o Container Services que utiliza o Docker Hub e Azure Container Registry para a gestão. Possui também um serviço único denominado Service Fabric, concebido para aplicações com uma arquitetura baseada em micro-serviços.

A Google disponibiliza o Google Compute Engine (GCE) que é idêntico ao serviço EC2 da AWS. Em relação a *containers* é disponibilizado o Kubernetes Engine, devido do facto da Google ter participado no projeto do Kubernetes [27, 73].

[11] AWS: `https://aws.amazon.com`
[12] Azure: `https://azure.microsoft.com`
[13] GCP: `https://cloud.google.com`

2.3.4.2 Armazenamento e base de dados

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

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

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

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

A Google tem um conjunto menor de serviços de armazenamento. Tem o Cloud Storage que é um serviço para o armazenamento de objetos e também tem a opção de Persistent Disk. Em relação às bases de dados, a GCP tem a Cloud SQL e uma base de dados relacional Cloud Spanner. Para base de dados NoSQL tem a Cloud Bigtable e Cloud Datastore, mas não fornece serviços de backup [27, 73].

2.3.4.3 Localização

Em relação aos locais onde estão presentes, todos têm infraestruturas que cobrem o mundo [73]. A Amazon é responsável por uma maior cobertura, com 49 zonas de disponibilidade, tendo planos para disponibilizar mais 12 zonas [2]. A Microsoft e a Google têm ambas 36 zonas, tendo a Microsoft perspetivas para incorporar mais 6 zonas [26, 39].

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

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

2.4 *Edge Computing*

Embora a computação em *Cloud* permita simplificar o processamento e armazenamento dos dados através de uma infraestrutura altamente disponível e com grande capacidade, toda a computação ocorre de uma forma centralizada nos servidores dos fornecedores de serviços (*cloud*). Tal implica que, na realização de operações, os clientes necessitam sempre de comunicar com os servidores. Este requisito leva a que haja um aumento das comunicações entre os dispositivos dos utilizadores e os servidores, os quais poderão não ter tempos de resposta adequados, sendo mais grave no caso de aplicações que requeiram uma baixa latência [68].

A computação na *Edge* (*Edge Computing*) vem possibilitar que parte da computação necessária a uma aplicação seja realizada na periferia da rede, ou seja, em dispositivos mais próximos dos dispositivos dos utilizadores, *edge nodes*, como por exemplo, *routers*, *switches* e *base stations*. Neste contexto, os dispositivos para além de consumirem os dados, também os produzem, isto é, estes pedem recursos e serviços à *cloud*, mas também realizam computação para a *cloud* [63].

A Figura 2.6 representa a localização da computação na *Edge*.

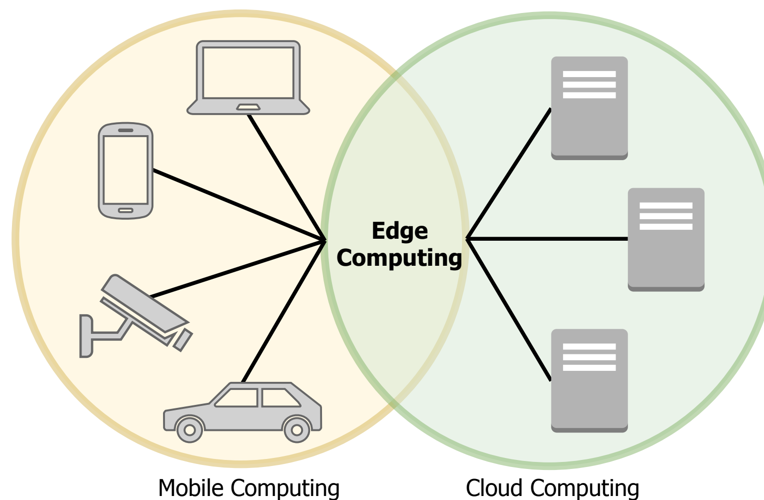


Figura 2.6: Localização da computação na *Edge*. Adaptado de [10].

2.4.1 Benefícios da *Edge Computing*

Realizar o processamento na *edge* permite obter certas vantagens em relação à computação em *Cloud*, sendo que as principais são [63, 68]:

- Diminuir os tempos de latência de certas operações, resultando numa melhor experiência de utilização para o utilizador;
- Processar dados na *edge*, enviando o resultado para a *cloud*, diminuindo assim o volume de dados transferidos entre o utilizador e a *cloud*;

- Reduzir a carga nos servidores na *cloud*, de forma a que fiquem com maior disponibilidade para outra tarefas e também para que o consumo de energia seja menor.

2.4.2 Desafios da *Edge Computing*

Uma vez que se trata de um tipo de computação relativamente recente, existem ainda certos desafios que necessitam de alguma atenção.

- **Computação nos *edge nodes***: a computação pode ser executada em diversos nós que estão localizados entre o dispositivo na *edge* e a *cloud*, mas nem todos estes nós são indicados para certos tipos de computação como, por exemplo, as *base stations* que possuem processadores de sinal digital (*Digital Signal Processors* - DSPs), e que não são indicados para a computação geral [68].

  Esta questão poderá vir a ser ultrapassada, por exemplo, nos *routers* wireless, ao serem atualizados para terem mais capacidade de processamento. No entanto, este processo implica custos bastante elevados [38, 68].

- **Programação**: a *cloud* permite ao programador um nível de abstração em que o código desenvolvido será *deployed* em máquinas semelhantes. Na *edge computing* existe uma heterogeneidade nos dispositivos o que dificulta a forma como o código é desenvolvido, pois tem de ser adequado a cada tipo de dispositivo/plataforma [63].

- **Descoberta de *edge nodes***: a existência de inúmeros dispositivos na *edge* requer uma filtragem para que apenas sejam usados os que preencham os requisitos necessários, sendo que esta descoberta não pode ser manual devido ao seu elevado número [68].

- **Privacidade e segurança**: são duas características bastante relevantes em qualquer sistema, e quando se fala em computação na *Edge* não é diferente. Por exemplo, ao utilizar um *router* para processar certos dados, este não pode ser acedido por um utilizador mal-intencionado, dando-lhes a possibilidade de aceder a dados privados [68].

  A inexistência de mecanismos eficientes de proteção dos dados na *edge* é um problema já que, tratando-se de um ambiente bastante dinâmico, torna-a uma rede mais vulnerável. Alguns dispositivos possuem capacidades bastante limitadas, fazendo com que os métodos de segurança utilizados atualmente possam não ser os mais adequados [63].

2.5 Computação na *Cloud* e *Edge*

A computação em plataformas heterogéneas, composta pela *cloud* e *edge*, com elevado número de recursos disponíveis na *edge*, levanta assim um conjunto de desafios em termos do desenvolvimento e execução das aplicações. Estes desafios incluem a localização

adequada dos serviços e a sua coordenação, o seu *deployment*, dependendo dos recursos dos nós computacionais, a resposta adequada ao volume de acessos por parte dos utilizadores, a garantia de questões sobre segurança e privacidade, que modelos de custos são vantajosos tanto para os fornecedores de serviços como para os utilizadores, etc.

A computação osmótica (*Osmotic Computing*) [69] é um dos paradigmas recentes que pretende dar resposta a esses desafios. O seu objetivo é permitir um *deploy* automático de micro-serviços tanto na *edge* como na *cloud* através de uma gestão eficiente dos recursos disponíveis, obtendo uma melhor capacidade computacional mais perto do utilizador [61, 69]. Este paradigma implica uma gestão dinâmica dos serviços na *cloud* e na *edge*, tendo em conta possíveis problemas relacionados com o *deployment*, rede e segurança.

A computação osmótica realça os novos desafios resultantes da heterogeneidade dos recursos da *cloud* e da *edge*, bem como os relacionados com a elasticidade da *cloud*. Na perspetiva deste paradigma, os micro-serviços são instalados em *containers* e *deployed* na *cloud* ou na *edge*, e a migração dos micro-serviços deve ser realizada de forma dinâmica e eficiente para evitar problemas, como paragens ou degradação da aplicação [69].

2.5.1 Monitorização em arquiteturas heterogéneas

Muitas das dificuldades em monitorizar a atividade de um sistema heterogéneo com estas caraterísticas prendem-se com a escala e complexidade da infraestrutura considerada como alvo do *deployment* de micro-serviços, tal como realçado no paradigma da computação osmótica [69].

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

Torna-se assim difícil implementar técnicas eficientes de monitorização que suportem a resolução de problemas que contribuam para a coordenação dos micro-serviços e sua qualidade de serviço quando, por exemplo, os micro-serviços têm de ser migrados para a periferia e o seu estado tem de continuar a ser monitorizado.

2.5.2 Orquestração de micro-serviços e controlo da elasticidade

Num sistema *edge* e *cloud*, a orquestração dos micro-serviços é um problema complexo devido à dificuldade em estimar o comportamento da carga de trabalho em termos de volume de dados a serem processados, taxa de chegada de dados, tipos de *queries*, número de utilizadores ligados a diferentes tipos de micro-serviços, etc. Sem determinadas informações sobre os micro-serviços, como a carga de trabalho de cada um deles, é difícil saber qual a escala em termos de recursos que deve ser escolhida para os micro-serviços [69].

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

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

2.5.3 Migração de serviços

Em ambientes de *cloud*, a migração pode permitir que seja realizada uma gestão mais eficiente dos recursos. Esta consiste no processo de mover VMs entre diferentes máquinas físicas, podendo ser utilizada para atingir vários objetivos, como a redução de energia, balanceamento de carga e tolerância a falhas [64].

2.5.3.1 Métodos de migração

A migração pode ser classificada em: *non-live* e *live*. Na migração *non-live*, primeiro a VM é colocada em suspensão antes de se iniciar o processo de transferência para outro local, e depois é novamente colocada em execução quando as tarefas de migração terminarem. Durante este tipo de migração, os serviços que são inseridos na VM não estarão acessíveis.

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

2.5.3.2 Objetivos da migração *live*

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

1. **Consolidação do servidor**: a gestão da energia pode ser conseguida através da consolidação do servidor, isto é, da otimização do número de máquinas físicas utilizadas, diminuindo o seu número total. Esta deve ser utilizada quando existe um grande número de máquinas físicas subutilizadas, devendo proceder-se à migração de VMs de máquinas físicas menos sobrecarregadas para outras mais sobrecarregadas.

2. **Balanceamento de carga**: permite diminuir o desequilíbrio ao nível de utilização dos recursos das máquinas físicas. A carga total do sistema pode ser ajustada através da recolocação de VMs de máquinas físicas sobrecarregadas para outras com menos carga.

2.5.3.3 Métricas de migração *live*

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

Podem ainda ser consideradas, consoante o tipo de serviço:

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

A identificação de como os micro-serviços podem ser migrados da *cloud* para a *edge* (e ao contrário), bem como os fatores que influenciam essa migração ainda são um desafio [61, 69].

2.6 *Containers*

Os *containers* são uma tecnologia de virtualização que permite a instalação de aplicações num único pacote, contendo as suas dependências em termos de bibliotecas e definições. Esta tecnologia possibilita a execução de múltiplas aplicações (*containers*) numa mesma máquina que tenha suporte à execução de *containers*, sem que existam conflitos entre eles [1, 7, 15].

Os *containers* são um mecanismo de virtualização bastante leve, em que um *container* pode ser iniciado em poucos segundos, e dado que não são uma virtualização completa do hardware físico como as máquinas virtuais, têm um menor *overhead*. A Figura 2.7 mostra a diferença entre as máquinas virtuais e os *containers*.

Atualmente existem diversos motores de *containers*, como o CoreOS RKT [14], o Mesos Containerizer [15] ou o Docker [16], sendo o Docker o mais utilizado, representando 83% da utilização de motores de *containers* em produção [65].

Para que um *container* seja iniciado, tem que ser dada a informação sobre qual a imagem que se pretende executar. Cada imagem contém a aplicação (ou aplicações) e todo o *software* necessário para a execução da mesma(s), sendo que pode estar disponível apenas localmente, ou em repositórios privados ou públicos, como o Docker Hub [17]. Cada *container* possibilita também ser configurado em termos de recursos que pode utilizar, por exemplo a nível de CPU, RAM, disco, etc.

[14] CoreOS RKT: `https://coreos.com/rkt/`
[15] Mesos Containerizer: `http://mesos.apache.org/documentation/latest/mesos-containerizer/`
[16] Docker: `https://www.docker.com/`
[17] Docker Hub: `https://hub.docker.com/`

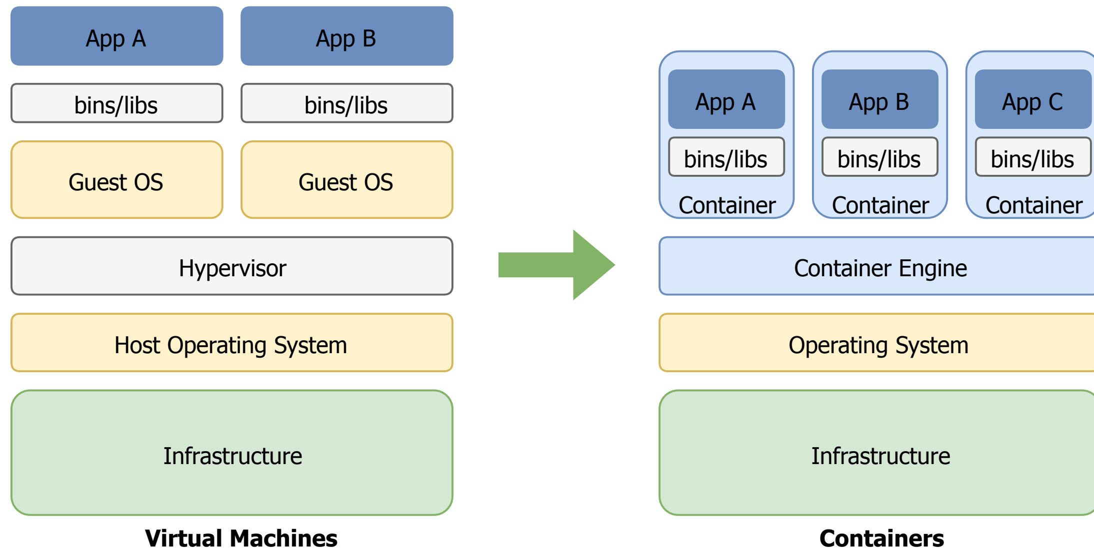


Figura 2.7: Máquinas virtuais vs. *containers*. Adaptado de [46].

2.6.1 *Containers* e micro-serviços

Como descrito anteriormente em 2.2.3.4, uma possibilidade de executar micro-serviços é com recurso a *containers*. Esta opção tem diversas vantagens, pois os micro-serviços são aplicações pequenas, que podem ser implementadas em diversas linguagens e *frameworks*, sendo que com a utilização de *containers* é possível obter uma abstração em que os micro-serviços são instalados contendo todas as suas dependências. Assim, podem ser executados micro-serviços diferentes com requisitos diferentes numa só máquina, sem que existam conflitos entre eles. Isto permite também que a monitorização dos micro-serviços seja feita ao nível do *container*, o que facilita a identificação dos recursos reais que são utilizados por um dado micro-serviço, bem como todo o *software* necessário para a sua execução [30].

2.6.2 Orquestração de *containers*

A execução de múltiplos *containers* em simultâneo torna necessária a existência de mecanismos que façam a sua gestão de uma forma eficiente e eficaz. A orquestração de *containers* é um conjunto de operações acordadas entre os fornecedores de serviços (*cloud*) e os proprietários das aplicações, para realizar o *deploy*, monitorizar e controlar dinamicamente as configurações dos recursos, de forma a garantir uma certa qualidade de serviço (QoS). Esta orquestração engloba o *deployment* inicial dos *containers*, mas também tem de efetuar a sua gestão em tempo de execução [6].

Devido ao elevado dinamismo e à complexidade do ambiente onde os *containers* são executados (por exemplo, um *cluster* com múltiplos nós em diferentes *data centers*), as decisões tomadas pela orquestração devem ter em conta as variações do ambiente e ser ajustadas em tempo de execução. Tal significa a necessidade da existência de mecanismos

autonómicos que façam essas decisões sem a intervenção humana.

Para realizar a orquestração de *containers* de uma forma autonómica existe a necessidade de ter em consideração certos fatores, como a gestão dos recursos em tempo de execução, a monitorização dos *containers*, etc.

2.6.2.1 Plataformas de orquestração

Atualmente já existem ferramentas que permitem a orquestração de *containers*. Contudo, ainda não fornecem funcionalidades verdadeiramente autonómicas que se adaptem aos mais variados cenários.

O Kubernetes [18] é uma ferramenta de orquestração de *containers* que possibilita a sua gestão num *cluster* de nós, sendo, atualmente, a ferramenta mais utilizada, com valores a chegar aos 51% de quota de mercado [65].

O Kubernetes oferece bastantes funcionalidades, como a substituição de um *container* que falha, a possibilidade de *auto scaling* de *containers* com base na taxa de utilização de CPU, entre outros. Por exemplo, o Kubernetes permite alterar automaticamente o número de *containers* através de um controlador denominado HPA (*Horizontal Pod Autoscaler*). O HPA é implementado através de um ciclo de controlo com um período de 30 segundos, em que a cada iteração é calculado o número de *containers* necessários com base na proporção entre o número de *containers* atual e a utilização de CPU, obtendo-se assim a utilização média de CPU. O resultado obtido é comparado com o valor limite de CPU previamente definido, e é então decidido se é necessário aumentar, diminuir, ou manter o número de *containers* atual. Para além da configuração padrão do HPA, em que é utilizada a métrica de CPU, é também possível configurar outro tipo de métricas [6, 31, 33].

O Docker Swarm é uma ferramenta que vem integrada com o Docker e que permite a criação de um *cluster* de nós para a execução de *containers*. Para cada tipo de *container* é possível especificar o número de réplicas pretendido, bem como alterá-lo em tempo de execução, sendo que, quando existe uma falha num determinado *container*, é inicializado um novo. Esta é a segunda ferramenta mais utilizada para a gestão de *containers* representando 11% de quota de mercado [65].

Apesar das diversas vantagens que estas ferramentas oferecem na gestão de *containers*, ainda não são muito desenvolvidas em termos de funcionalidades no que diz respeito a mecanismos de adaptação quando existe a variação no número de acessos aos *containers*, nem na identificação da melhor localização para a execução dos mesmos com base na proveniência dos acessos.

2.7 Trabalho relacionado

Existem alguns trabalhos correntes que partilham parte dos objetivos que são pretendidos alcançar com a solução, por exemplo, a nível da elasticidade dos micro-serviços

[18] Kubernetes: https://kubernetes.io/

e a computação na *edge*. Na secção 3.6 é depois realizada uma comparação entre estes trabalhos e a solução desenvolvida.

2.7.1 CAUS - Custom AutoScaler

A solução proposta em [31] centra-se essencialmente em fornecer mecanismos através de um processo de adaptação que confere funcionalidades elásticas aos micro-serviços executados em *containers*, ou seja, permite aumentar ou diminuir o número de *containers* em execução, de um dado tipo de serviço, consoante as necessidades.

Nesta solução são considerados micro-serviços sem estado, onde a respetiva carga de trabalho é baseada em filas, isto é, cada *container* fica a escutar uma fila de pedidos, onde são inseridos os pedidos para serem processados. O método está dividido em duas partes: a primeira adapta o número de *containers* necessários de um dado tipo de serviço, com base nas alterações da intensidade de carga; a segunda, tem como objetivo gerir *containers* adicionais para dar resposta a mudanças imprevisíveis na intensidade de carga.

2.7.1.1 Processo de adaptação

De forma a conferir características elásticas aos micro-serviços (executados em *containers*), o processo de adaptação é conseguido através de um controlador elástico, sendo que é necessário informação relativa à capacidade que um único *container* consegue processar.

Assim que o controlador é iniciado, este começa a monitorizar periodicamente a intensidade da carga dos *containers* que se encontram em execução. Em seguida, na fase de análise, consoante o valor obtido na fase de monitorização, é tomada a decisão de aumentar, diminuir ou manter o número de *containers*. Na fase de planeamento, o controlador garante que o número calculado de *containers* fica entre os valores mínimo e máximo pré-definidos. De forma a evitar a mudança entre duas decisões contraditórias, o controlador utiliza um período de tempo de 3 minutos entre as tomadas de decisões de escalar. Por fim, na fase de execução, o controlador aumenta ou diminui o número de *containers* em execução através da invocação de APIs da plataforma.

As características que são descritas como inovadoras do controlador elástico são baseadas em três partes:

Conhecimento: capacidade Os pedidos são enviados para uma fila específica em que o micro-serviço fica a escutar. Posteriormente, o controlador precisa da informação sobre a capacidade que uma instância de um dado micro-serviço consegue processar, isto é, a intensidade máxima de carga de trabalho que um *container* pode lidar, denotada por μ. É assumido que o trabalho é homogéneo, portanto cada pedido tem a mesma carga. Esta metodologia pode ser alargada a várias filas com diferentes tipos de pedidos, onde é necessário saber a capacidade estimada para cada uma delas.

Monitorização: intensidade da carga O valor da intensidade da carga, denotado por λ, indica a taxa na qual a fila aumentou no último minuto. Na implementação do protótipo é obtido um sinal através do Prometheus, utilizando a função de taxa (*rate function*) com o intervalo de um minuto. Este sinal indica a necessidade de escalar. Através de técnicas preditivas, os recursos são provisionados para unidades de horas, sendo que ao ter um mecanismo reativo, em que as suas decisões têm como base um sinal que reflete a carga de trabalho, é possível dimensionar os recursos para unidades de tempo mais refinadas.

Análise: reagir Com a obtenção do valor da intensidade da carga (λ), e com a informação sobre a capacidade de um *container* (μ), o controlador elástico calcula o número de *containers* necessários para corresponder à intensidade atual, através do rácio $\frac{\lambda}{\mu}$. Contudo, existem questões que impedem a utilização de apenas um mecanismo reativo. Um dos desafios está relacionado com os atrasos em refletir a alteração da magnitude referente à intensidade da carga e com o tempo necessário para iniciar os *containers*. O outro desafio consiste na capacidade para lidar com picos de curta duração que não são refletidos na intensidade de carga em utilização. Estas questões são abordadas através da utilização de *containers* adicionais de forma a fornecerem capacidade de reserva.

Análise: gerir recursos de reserva A gestão do número de *containers* adicionais, efetuada pelo processo de adaptação, é baseada em dois parâmetros. O primeiro parâmetro é o número inicial de *containers* adicionais (si) que deve sempre coexistir com o número de *containers* calculado pelo mecanismo reativo. O outro parâmetro é um valor limite ($threshold$), em percentagem, que determina quando a capacidade de reserva deve escalar. Quando é obtido o valor da intensidade de carga, o controlador elástico calcula a percentagem com a qual os *containers* de reserva estão a contribuir para a nova taxa de chegada. Se a percentagem estiver acima do limite definido, o controlador fornece um *container* adicional, caso contrário, se a percentagem for menor que o limite, o controlador diminui o número de *containers* de reserva em uma unidade.

2.7.2 ENORM - Edge Node Resource Management

Em [71], foi desenvolvido um protótipo de uma *framework* de *Fog Computing*, com o objetivo de realizar computação na periferia (*edge*) da rede, mais próximo dos utilizadores. Os resultados obtidos foram menor latência na comunicação, menor tráfego de dados para a *cloud* e reduzir a frequência da comunicação entre os utilizadores e a *cloud*, através da gestão eficiente dos recursos.

2.7.2.1 Arquitetura

A arquitetura é composta por três camadas. A camada de cima é a da *cloud*, onde são executados os servidores aplicacionais. Para permitir a utilização da *cloud* com nós da

edge, a *framework* realiza o *deploy* de um gestor de servidor da *cloud* em cada servidor aplicacional. Este servidor:

1. Comunica com potenciais nós da *edge*, para pedir serviços de computação;

2. Realiza o *deploy* de servidores particionados num nó da *edge*;

3. Recebe atualizações do nó da *edge* para atualizar a vista global do servidor aplicacional da *cloud*.

A camada de baixo é a camada dos dispositivos dos utilizadores. Num modelo de execução apenas na *cloud*, os dispositivos conectam-se aos servidores aplicacionais através de nós de encaminhamento de tráfego. Os serviços básicos [19] de *routers* na rede wireless local são utilizados para comunicar com um servidor centralizado na *cloud*, contudo com a *framework* ENORM, as capacidades de computação dos nós da *edge* são utilizadas.

A camada do meio é a camada dos nós da *edge*, tendo sido focado a utilização de apenas de um único nó da *edge*. Por exemplo, é iniciada uma aplicação em múltiplas regiões cobertas por nós da *edge*. São estabelecidas conexões com os respetivos servidores aplicacionais na *cloud* e é realizado o *deploy* de um servidor aplicacional particionado nos nós da *edge*.

O servidor particionado difere do servidor aplicacional da *cloud*, pois os dados locais relevantes para os utilizadores abrangidos por um determinado nó da *edge* são mantidos. É mantida a vista global no servidor da *cloud* e o nó da *edge* fica responsável por atualizar o servidor da *cloud*. Quando os nós da *edge* não conseguem fornecer serviços de computação, ou estes não melhorem a QoS da aplicação, então o servidor aplicacional na *edge* é terminado e os utilizadores são encaminhados para o servidor da *cloud*.

Num nó da *edge* são necessários cinco componentes:

1. **Distribuidor de recursos**: num nó da *edge*, o seu serviço básico não pode ser comprometido, tendo por isso maior prioridade sobre qualquer carga de trabalho. É necessário saber os recursos disponíveis de cada nó, sendo da responsabilidade do distribuidor de recursos controlar o CPU e a memória disponíveis.

2. **Gestor da *edge***: o gestor da *edge* é composto por dois subcomponentes, o gestor do nó e o gestor de servidor. O gestor do nó é responsável por tratar dos pedidos recebidos pelo gestor de servidor, realizados por um servidor da *cloud*. Quando o gestor da *cloud* faz um pedido, o gestor do nó decide se aceita ou não o pedido (dependendo da existência de recursos no nó da *edge* e se a prioridade da aplicação que realizou o pedido é maior ou igual a um servidor em execução no nó). A resposta é enviada ao servidor da *cloud* que fez o pedido e, quando a resposta for aceite, o gestor de servidor inicializa um *container*, reserva as portas necessárias e atualiza as configurações de *firewall*.

[19] Serviço básico: por exemplo, o serviço básico de um ponto de acesso Wi-Fi é encaminhar Internet.

3. **Monitor**: periodicamente é monitorizado um conjunto de métricas relacionadas com os servidores aplicacionais da *edge*, nomeadamente a latência da comunicação (através do comando ping) e a latência da computação (através de *timestamps* e *logs* do servidor). As métricas são utilizadas pelo auto-scaler para verificar a necessidade de reservar ou libertar recursos a um servidor aplicacional.

4. **Auto-scaler**: com base nas métricas obtidas pelo distribuidor de recursos e do monitor, o auto-scaler reserva ou liberta recursos dinamicamente aos *containers*, que executam os servidores aplicacionais. Esta reserva é necessária para: a) assegurar a existência de recursos suficientes para o serviço básico, e que não existem sobrecargas; b) modificar os recursos alocados para suportar mais utilizadores ou novos servidores aplicacionais. Se um servidor aplicacional não conseguir obter recursos, ou se conseguir, mas sem melhorias de performance na aplicação, o gestor da *edge* termina o servidor aplicacional.

5. **Servidor aplicacional da *edge***: um servidor particionado da *cloud* fica em execução no nó da *edge*. Este interage com os dispositivos dos utilizadores, antes de encaminhar os dados para a *cloud*.

2.7.2.2 Gestão de recursos

Fornecimento O mecanismo de fornecimento num nó da *edge* inclui três passos, *handshaking*, *deployment* e terminação. Quando um gestor da *edge* inicia, é verificado se o nó pode suportar serviços na *edge*. Se for possível, é inicializado um protocolo de *handshaking*.

- ***Handshaking***: o nó da *edge* fica à espera de pedidos dos gestores da *cloud*. Se o nó puder fornecer serviços ao gestor da *cloud*, então este é notificado e envia um pedido com a configuração de um servidor da *edge*. Isto inclui o nome da aplicação, o nível de prioridade, portas necessárias, os utilizadores que irão utilizar o servidor da *edge* e os requisitos de latência. O *handshaking* serve para identificar o nó da *edge* no qual será feito o *deploy* da aplicação, inicializar um *container* no nó e configurar as *firewalls*.

  É feita uma verificação dos recursos e da prioridade quando é recebido o pedido com a configuração. Se o novo pedido tiver uma prioridade mais baixa que a dos servidores em execução ou não existirem recursos suficientes para o novo *container*, o pedido é rejeitado. O gestor da *cloud* pode então decidir pedir serviços a um outro nó da *edge*. Se o pedido passar nas verificações, o gestor da *edge* verifica a disponibilidade das portas propostas. É gerada uma porta para o acesso remoto com o gestor da *cloud*. Caso as portas sejam alocadas com sucesso, é iniciado um *container* com um sistema operativo e uma imagem padrão fornecida pelo gestor da *edge*. O *container* recebe uma quantidade de recursos padrão (um núcleo de CPU e 200 MB de RAM) e, quando é terminada a inicialização, o gestor da *edge* configura

o *container* de modo a que a aplicação executada no mesmo fique disponível para os utilizadores e para ser gerido pelo servidor da *cloud*. Depois da configuração, é enviada uma mensagem para o gestor de servidor da *cloud*.

- ***Deployment***: o gestor de servidor da *cloud* realiza o *deploy* de um servidor aplicacional particionado na *edge* e instala o software necessário no *container*. Para cada servidor aplicacional, o respetivo *container* é personalizado pelo gestor de servidor da *cloud*, evitando que sejam gastos recursos com um *container* genérico. Assim que o servidor aplicacional é inicializado no nó da *edge*, pode começar a receber pedidos dos utilizadores. O gestor de servidor da *cloud* redireciona os utilizadores da aplicação abrangidos pelo nó da *edge*, através de um ficheiro de configuração que aponta para o servidor da *edge* ao invés do servidor da *cloud*. Assim que os utilizadores estão conectados, os recursos do *container* podem ser alterados (podem ser alocados mais recursos, se disponíveis, ou desalocados se não forem necessários).

- **Terminação**: o gestor da *edge* decide quando um servidor aplicacional da *edge* precisa de ser removido do nó, podendo ser terminado em três casos (considerados no mecanismo de *auto scaling*): 1. quando não existem recursos disponíveis para suportar o serviço; 2. o serviço já não é necessário (o servidor da *edge* esteve inativo durante um período de tempo); 3. o serviço da *edge* não melhorou a QoS da aplicação. Podem ser terminados um ou mais *containers*. Quando um servidor aplicacional da *edge* é terminado, os dados que contêm as atualizações locais são migrados para a *cloud*. Isto é realizado através da base de dados chave-valor, Redis [20], que suporta a migração de dados entre dois servidores. Os utilizadores são redirecionados para o servidor da *cloud*, até que seja fornecido um novo nó da *edge* pelo gestor de servidor da *cloud*.

Auto scaling É importante escalar os recursos alocados a um servidor aplicacional da *edge*, pois: 1. os nós da *edge* têm recursos limitados (sendo necessários principalmente para os serviços básicos) e, 2. o servidor aplicacional requer mais ou menos recursos para corresponder aos objetivos de QoS (considerada apenas a latência operacional). O mecanismo de *auto scaling* acontece periodicamente (a cada 5 minutos) durante a execução de um servidor aplicacional num nó da *edge*. Para o *auto scaling* são necessários, o gestor da *edge*, o distribuidor de recursos e o monitor.

O gestor da *edge* mantém uma lista dos servidores aplicacionais em execução e as suas prioridades. A prioridade é definida pelo gestor da *cloud* que possui a aplicação, sendo esta estática e não mudando durante a execução. A aplicação com maior prioridade é analisada primeiro pelo auto-scaler, sendo medida a latência da rede através de pings entre os utilizadores e o servidor. É depois verificado se a quantidade de recursos disponíveis no nó da *edge* (obtida pelo distribuidor de recursos) é suficiente para satisfazer a

[20]Redis: https://redis.io/

quantidade mínima de recursos para o servidor aplicacional da *edge*. Se a verificação de recursos falhar, o servidor com maior prioridade, juntamente com os outros com menor prioridade, é migrado de volta para a *cloud*. Isto é realizado para garantir que os serviços básicos dos nós da *edge* mantenham a prioridade mais alta. Se a verificação dos recursos for válida, então o auto-scaler verifica se é necessário o servidor aplicacional da *edge* (se os utilizadores estão abrangidos por esse nó ou se a latência da aplicação pode ser reduzida no nó).

A existência de utilizadores (se os utilizadores estão conectados ao nó da *edge*) e a latência da rede são utilizadas para decidir se o servidor da *edge* pode fornecer melhorias ou se a migração do servidor aplicacional para a *cloud* ou para outro nó da *edge* pode ser mais benéfica. A latência da aplicação é comparada com a latência objetivo e, caso a latência da aplicação seja maior, são alocados mais recursos ao *container*, caso seja menor que o objetivo, são removidos recursos do *container*.

Quando o mecanismo de *auto scaling* decide realizar um *scale-up*, isto é, alocar mais recursos a um servidor aplicacional, o monitor da *edge* verifica se existem mais recursos disponíveis no nó da *edge*. Se existirem recursos, são adicionadas ao *container* uma ou mais unidades de recursos (uma unidade - um núcleo de CPU e 200 MB de RAM). Se não existirem recursos suficientes para realizar o *scale-up*, são terminados *containers* com menor prioridade, um de cada vez, até que hajam recursos suficientes. Num *scale-down*, é removida uma unidade de recursos alocada ao *container* que contém o servidor aplicacional. O auto-scaler repete o processo para o próximo servidor da lista, até falhar a verificação dos recursos no nó da *edge*.

2.8 Sumário

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

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

Foi ainda possível obter mais conhecimentos sobre propostas de soluções já existentes para a resolução dos problemas identificados (na secção 1.2), como a computação osmótica, cujo objetivo é permitir o *deployment* automático dos micro-serviços quer na *cloud* quer na *edge*. A solução CAUS, na secção 2.7.1, também fornece algumas características interessantes sobre como dotar os micro-serviços de funcionalidades elásticas, isto é, permitir a sua replicação, umas das características com maior relevância para a solução.

A seguir, no capítulo 3 será abordada de forma mais detalhada a arquitetura da solução, bem como os possíveis cenários de migração e replicação possíveis.

Capítulo 3

Solução proposta

Este capítulo tem como objetivo detalhar a arquitetura do componente de gestão de micro-serviços na *cloud* e *edge* e todos os subcomponentes necessários para o funcionamento do sistema. Na secção 3.1 é descrita, de forma geral, a arquitetura global do sistema onde está inserida a solução. Nas secções 3.2, 3.3, 3.4 e 3.5 são abordadas a arquitetura da solução, as suas principais funcionalidades e restrições. É apresentada, na secção 3.6, uma comparação entre os trabalhos relacionados e a solução. Por fim, na secção 3.7, são referidos os possíveis cenários de migração e replicação, onde são analisadas as decisões que podem ser tomadas.

3.1 Arquitetura de um sistema de gestão de micro-serviços com propriedades autonómicas

O componente de gestão de micro-serviços está incorporado numa Arquitetura de micro-serviços autonómica idealizada para permitir uma gestão dinâmica de aplicações baseadas em micro-serviços em infraestruturas híbridas de *cloud* e *edge* [34].

Esta arquitetura inclui três componentes fundamentais: componente de gestão de micro-serviços (abordado nesta dissertação), componente de gestão de base de dados e componente de monitorização (ambos abordados noutras dissertações), que comunicam entre si de forma a garantir as funcionalidades pretendidas. É incluída uma descrição geral dos componentes externos a esta dissertação, como forma de clarificar as suas inter-dependências, e o funcionamento do componente de gestão de micro-serviços.

As funcionalidades principais de cada um dos componentes são:

Componente de gestão de micro-serviços Este componente (detalhado na secção 3.2) é responsável por controlar o *deployment* das instâncias dos micro-serviços pelos nós

disponíveis (na *cloud* e na *edge*), englobando as seguintes operações:

1. Migrar instâncias existentes dos micro-serviços;
2. Criar novas instâncias (i.e., réplicas) dos micro-serviços;
3. Remover instâncias que já não são necessárias para o funcionamento correto e eficiente da aplicação (de forma a minimizar o custo de operação).

Como os micro-serviços podem ter de interagir com outros, este componente tem de contemplar funcionalidades que permitam a comunicação eficiente entre diferentes micro-serviços. Deve também tratar a ligação dos micro-serviços às instâncias de base de dados em conjunto com o componente de gestão de base de dados.

Estas decisões são complexas, necessitando considerar um número elevado de decisões de *deployment* possíveis em tempo de execução, tais como:

- os indicadores de desempenho quer a nível dos serviços quer dos nós onde eles executam, tais como a latência em processar os pedidos ou consumo de recursos;
- a necessidade de considerar as dependências dos micro-serviços (i.e., que micro-serviços comunicam entre si e a frequência com que o fazem, tendo em vista avaliar se deve ser efetuada a migração ou replicação de um ou mais micro-serviços);
- as restrições relacionadas com a capacidade dos nós individuais.

Componente de gestão de base de dados Gere o ciclo de vida e as interações dos vários sistemas de armazenamento de dados de uma aplicação, englobando as seguintes operações:

1. migrar uma instância de base de dados (total ou parcial);
2. criar uma nova instância de base de dados (ou seja, uma nova réplica, total ou parcial);
3. remover uma instância de base de dados existente.

A criação de novas instâncias de base de dados de um micro-serviço tem de ter em conta os recursos disponíveis das máquinas (físicas ou virtuais) para a sua execução. A replicação pode ser total ou parcial (uma fração dos dados armazenados) dependendo da dimensão da base de dados e dos recursos no nó de destino.

A gestão das réplicas da base de dados pode ser complexa pela necessidade de manter os dados replicados consistentes (através de protocolos que garantam um certo nível de consistência, forte ou fraca). Este componente deve conseguir também localizar as várias instâncias de base de dados ativas no sistema e garantir uma ligação adequada entre os micro-serviços e as réplicas das suas base de dados (por exemplo, se um serviço é migrado para um nó da *edge*, é necessário decidir se é criada uma nova réplica da base de dados e onde, ou se se utiliza uma já existente).

Componente de monitorização O componente de monitorização deve ser executado de maneira distribuída em recursos computacionais localizados na *cloud* e na *edge*. Para cada recurso é relevante obter métricas tais como:

1. capacidade disponível dos recursos (CPU, Memória, Largura de Banda);
2. consumo corrente dos recursos dos micro-serviços e base de dados;
3. indicadores de desempenho associados às operações de micro-serviços e base de dados (que permitem averiguar se a QoS está ser cumprida), como a latência dos acessos, número de acessos recebidos e a sua origem, e identificação de erros e falhas (por exemplo, deteção que um micro-serviço falhou para que se possa criar um novo).

Desenvolver uma solução de monitorização tem vários desafios. A obtenção de informações a partir dos micro-serviços e das bases de dados requer o uso de mecanismos ligados diretamente a esses componentes, ou a exposição de APIs padrão que são utilizadas por esses componentes para eles próprios fornecerem informações relevantes. A monitorização da capacidade disponível nos recursos pode ser extraída do sistema operativo, mas isso envolve o suporte a diferentes sistemas operacionais e ambientes de execução (por exemplo, virtualização, *containers*). A monitorização, o pré-processamento dos dados obtidos, e a propagação desses dados para os componentes relevantes (distribuídos) na arquitetura de micro-serviços autonómica, consomem bastantes recursos computacionais e de rede. Isto implica que o próprio componente de monitorização tenha que ser flexível, de modo a que possa adaptar o número de métricas a monitorizar, ou realizar a sua agregação, e a periodicidade com que essas métricas são obtidas. O objetivo é evitar uma saturação da capacidade dos nós e/ou a capacidade da rede de comunicação.

3.2 Arquitetura do componente de gestão de micro-serviços

O objetivo geral do componente de gestão de micro-serviços é contribuir para a obtenção de um sistema capaz de gerir micro-serviços de uma forma eficiente e automática. Pretende-se o mínimo de intervenção humana possível, de modo a que a QoS apresentada aos utilizadores seja sempre mantida nos intervalos definidos, através das decisões de replicação e migração dos micro-serviços.

A Figura 3.1 representa uma visão geral dos componentes necessários para o funcionamento do componente de gestão de micro-serviços. A infraestrutura engloba tanto nós na *edge* (*edge node*) como nós na *cloud*. Em relação aos nós na *cloud*, existe a possibilidade de serem provenientes de diferentes fornecedores de *cloud*, sendo que estes nós podem ser máquinas virtuais (VMs) inicializadas consoante o necessário para executar tanto micro-serviços como componentes do sistema.

Os nós formam um *cluster* cujo objetivo é ter um conjunto de recursos para a execução dos micro-serviços. Este *cluster* constituído por um *master* (representado com M na figura),

e por vários *workers* (representados com W na figura), onde o *master* tem conhecimento de todos os nós que formam o *cluster*.

Os micro-serviços, representados na figura, apresentam a designação de e-μS (micro-serviço estendido), pois este é composto por um determinado micro-serviço da aplicação, juntamente com um componente de sistema, o componente para registar e descobrir serviços. Este componente é explicado com maior detalhe na secção 3.2.3.2.

O componente de gestão de micro-serviços (*µServices Management*) é executado num nó na *cloud* (não pertencente ao *cluster*), em que lhe é indicado, aquando da sua inicialização, qual o nó que irá ser o *master*, que tanto pode ser na *cloud* como na *edge*. Este componente depois trata de inicializar outros subcomponentes, como o de monitorização (*Monitoring*), descrito em 3.2.3.4, que é utilizado para monitorizar os nós e os micro-serviços.

Quando uma aplicação de micro-serviços é inicializada, o componente de gestão de micro-serviços trata de lançar os micro-serviços que compõem a aplicação. Caso existam serviços do tipo *frontend*, o componente de gestão de micro-serviços trata de inicializar um outro sub-componente de sistema, *load balancer*, para balancear a carga entre as réplicas respetivas, podendo existir vários para cada tipo de serviço de *frontend*, um por local/região.

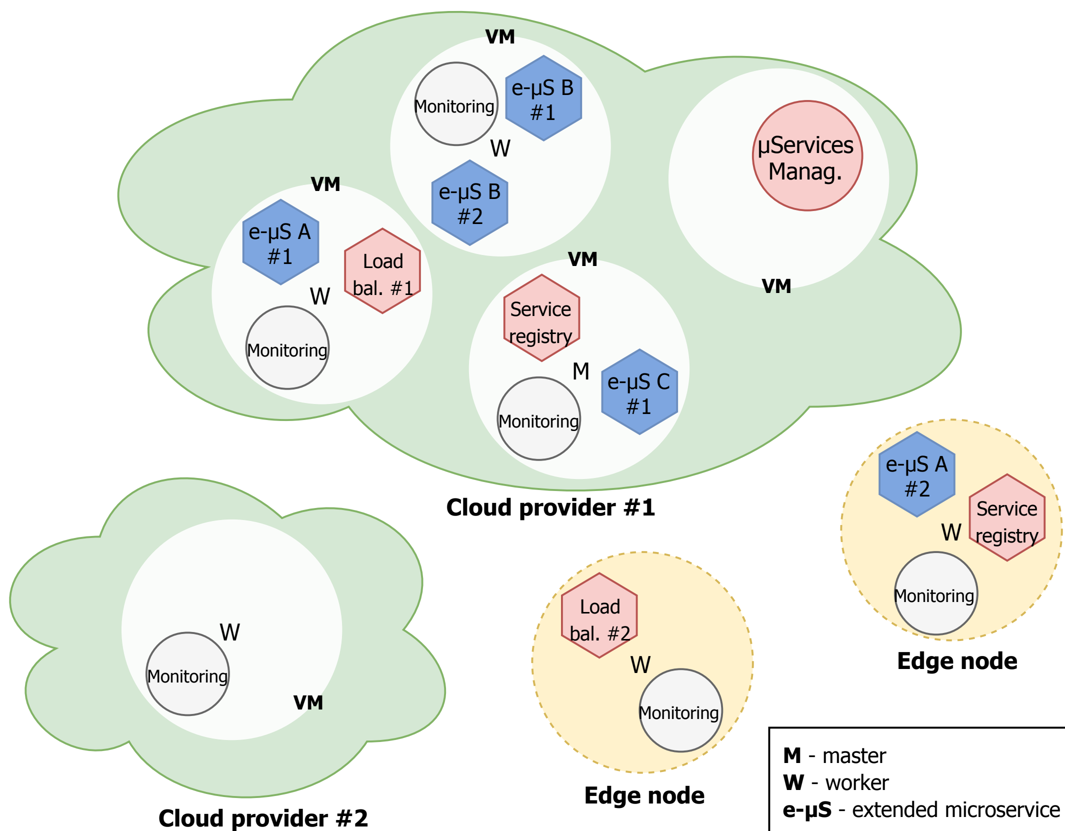


Figura 3.1: Visão do sistema de gestão de micro-serviços proposto e componentes associados.

3.2.1 Informações necessárias à execução de serviços e nós

Para que o componente de gestão de micro-serviços funcione, necessita de certas informações pré-configuradas, em relação aos serviços e aos nós. É também importante obter informações sobre o estado dos mesmos (métricas), sendo estas recolhidas por um componente de monitorização.

3.2.1.1 Serviços

Os serviços são executados sob a forma de *containers* e a informação necessária ao seu funcionamento inclui:

- **Informações para iniciar a execução**:
  - Tipo de serviço (*frontend*, *backend*, base de dados);
  - Repositório da imagem do serviço;
  - Portas do serviço (em que portas o serviço pode ser acedido);
  - Comando para iniciar a sua execução (por exemplo, parâmetros de entrada necessários para inicializar o serviço);
  - Dependências em relação a outros serviços (refere-se à comunicação com outros serviços).
- **Informações de funcionamento**:
  - Número mínimo de réplicas do serviço em execução;
  - Número máximo de réplicas do serviço em execução;
  - Parâmetros/métricas limites para o correto funcionamento de uma réplica do serviço (por exemplo, limite em termos de memória RAM).
- **Informações de monitorização**:
  - Latência ao serviço;
  - Acessos ao serviço (número de acessos e a sua proveniência);
  - Largura de banda;
  - Recursos utilizados pelo serviço (CPU, RAM, . . .).

3.2.1.2 Máquinas (nós)

Informação associada aos nós, a qual pode diferir consoante se encontrem na *edge* ou na *cloud*:

- **Informações de funcionamento - *edge* e *cloud***:

  - Parâmetros/métricas limites para o correto funcionamento de um nó (por exemplo, limite em termos de memória RAM).

- **Informações de funcionamento - *edge***:
  - Informações sobre a localização do nó: continente, região, país e cidade.

- **Informações de monitorização - *edge* e *cloud***:
  - Recursos utilizados pelo nó (CPU, RAM, . . .);
  - Largura de banda.

Os nós na *cloud* também têm informações sobre a sua localização, mas esta é menos precisa (no máximo é possível saber o continente e a região) e não é necessário ser armazenada, já que é possível consultá-la com recurso a APIs fornecidas pelo fornecedor de *cloud*.

3.2.2 Funcionalidades principais oferecidas pelo componente de gestão

O componente de gestão de micro-serviços integra várias funcionalidades:

3.2.2.1 Tomada de decisão

A associação de limites às capacidades de funcionamento de um serviço pode ser traduzida em regras que são resolvidas através de um motor de regras, com um resultado dependendo dos valores (parâmetros), representando o estado corrente do sistema. As situações representadas através de regras são:

- Deteção se um ou mais parâmetros ultrapassam os valores definidos, tendo como resultado a indicação da necessidade de replicação ou migração do serviço;
- Deteção se os parâmetros não atingem um valor mínimo definido, tendo como resultado a indicação da necessidade de remoção da réplica;
- Nada é feito, caso os valores estejam dentro dos limites mínimos e máximos definidos.

Quando o resultado da regra é replicar/migrar um serviço, é possível definir a prioridade sobre onde deve ser colocado o novo serviço em execução, isto é, se num nó na *edge* ou num nó na *cloud*. Caso seja definido um nó na *edge* e não houver nenhum disponível, é colocado o serviço na *cloud*.

As regras associadas aos nós, por sua vez, permitem capturar as situações seguintes:

- A inclusão de um novo nó, caso os recursos para a execução dos *containers* não seja suficiente, sendo possível definir a prioridade onde vai ser colocado o novo nó, isto é, um nó na *edge* ou nó na *cloud*. Caso seja definido um nó na *edge* e não houver nenhum disponível, é adicionado na *cloud*;

- A remoção de um nó, caso os seus recursos estejam a ser subutilizados;
- Nada, se se encontrar dentro dos limites.

O motor de regras permite que os valores associados às regras sejam configurados em vários modos:

- **valor efetivo**: que corresponde ao valor exatamente lido para determinada métrica;
- **valor da média**: que corresponde à média atual para determinada métrica;
- **percentagem de desvio da média**: que corresponde à percentagem de desvio do valor atual em relação à média de uma determinada métrica;
- **percentagem de desvio do último valor**: que corresponde à percentagem de desvio do valor atual em relação ao último valor lido de uma determinada métrica.

3.2.2.2 Agendamento de eventos

É possível agendar certos eventos, em que durante um período de tempo, é expectável que determinado serviço tenha, por exemplo, um número de acessos mais elevado. Tal permite aumentar a quantidade mínima de réplicas necessárias para o serviço, onde necessário, de forma a suportar o volume de acessos. O mesmo se procede quando é expectável que, devido a determinado acontecimento, o número de acessos a um serviço diminua, podendo ser diminuída a quantidade mínima de réplicas em execução para o tipo de serviço.

A Tabela 3.1 representa um evento em que é expectável que o número de acessos ao Serviço de Pagamento aumente, devido a um maior volume de compras na época do Natal.

Tabela 3.1: Exemplo de um agendamento de evento.

Descrição	Serviço	Data início	Data fim	Nº mínimo de réplicas
Natal	Serviço de Pagamento	23/12/2018 23:59	26/12/2018 23:59	10

3.2.3 Subcomponentes necessários ao sistema de gestão

Para o correto funcionamento do sistema são necessárias certas funcionalidades, como a comunicação entre micro-serviços de uma forma mais abstrata, o balanceamento de carga entre os serviços, métricas dos nós e serviços, sendo que estas são fornecidas por subcomponentes externos ao componente principal.

3.2.3.1 Componente de registo de serviços (*service registry*)

É um componente, representado como *Service registry* na Figura 3.1, que permite que os serviços em execução sejam registados e descobertos, de forma a que possam comunicar com outros de uma forma mais abstrata, ou seja, não é necessário conhecer o *endpoint* do serviço destino, apenas o tipo de serviço. De uma forma simples, este é um componente que guarda os *endpoints* dos serviços, organizados por tipo de serviço, contendo também mais informações sobre eles, como a sua localização e se estão ativos. Este componente é também um *container* que pode ser replicado por vários nós, de forma a que o acesso ao mesmo seja eficiente.

3.2.3.2 Componente para registar e descobrir serviços

É um componente que é adicionado à imagem do serviço da aplicação, sendo então a imagem composta por este componente e pelo próprio serviço. Quando é iniciado um *container*, o componente e serviço ficam em execução. Este componente tem o objetivo de registar o serviço da aplicação no *service registry* quando o serviço está em funcionamento, e de removê-lo caso exista algum problema (por exemplo, o serviço termina inesperadamente). Quando o serviço da aplicação necessita de comunicar com um outro serviço, envia um pedido a este componente, e este comunica com o *service registry* para saber os *endpoints* do serviço desejado. É aplicado um algoritmo (Algoritmo 1) para escolher o melhor serviço e para balancear a carga e, como resposta, obtém-se o *endpoint*. Através deste componente de gestão local, o desenvolvimento dos serviços fica facilitado na parte da comunicação com outros serviços, sendo que para isso apenas é necessário invocar certas APIs.

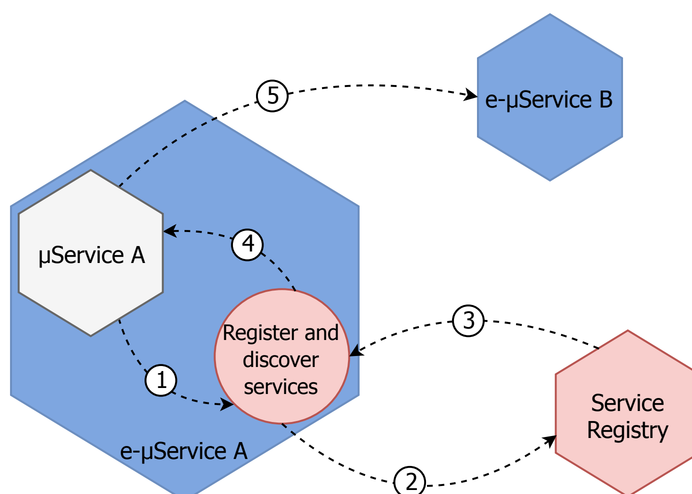


Figura 3.2: Comunicação de um serviço com outro serviço, com recurso ao componente usado para registar e descobrir serviços.

A Figura 3.2 mostra a composição dos *containers* que utilizam o componente para registar e descobrir serviços, sendo visível no micro-serviço A estendido (e-μService A). É

também representado como se procede quando os micro-serviços necessitam de comunicar com outros, sendo que para o micro-serviço A comunicar com o micro-serviço B, são feitos os seguintes passos:

1. O micro-serviço A comunica com o componente local (no mesmo *container*) usado para registar e descobrir serviços, através de uma API, enviando no pedido que pretende um *endpoint* do micro-serviço do tipo B.

2. O componente usado para registar e descobrir serviços comunica com o componente de registo de serviços (*service registry*), que contém os *endpoints* dos serviços, a pedir os *endpoints* de serviços do tipo B.

3. O componente de registo de serviços devolve os *endpoints* dos serviços do tipo B ao componente de registar e descobrir serviços.

4. O componente usado para registar e descobrir serviços envia o *endpoint* de um serviço B para o micro-serviço A.

5. O micro-serviço A tem o *endpoint* do serviço B, e pode fazer a comunicação.

Este componente vai guardando também o número de vezes que o serviço requisitou os *endpoints* de outros tipos de serviço, enviando periodicamente essa informação para o componente de monitorização de pedidos (detalhado na secção 4.2.4) do nó onde está em execução. Quando um utilizador acede a serviços do tipo *frontend*, o serviço contacta com o respetivo componente para registar e descobrir serviços, fornecendo a informação que houve um pedido proveniente de um determinado local (localização do utilizador).

3.2.3.3 Componente de balanceamento de carga

Este componente, representado como *Load Balancer* na Figura 3.1, tem como funcionalidade balancear a carga dos serviços de *frontend*. Cada serviço do tipo *frontend* pode ter vários *load balancers*, existindo um por cada região onde os serviços de *frontend* estão disponíveis, de forma a balancear a carga de cada local. Os *load balancers* têm acesso a todas as réplicas dos serviços independentemente do local onde se encontram, de forma a quando não existirem mais réplicas na sua região, ou estas estiverem sobrecarregadas em termos de pedidos em relação a outras, poderem redirecionar os pedidos para serviços de ouros locais.

A Figura 3.3 representa o funcionamento dos *load balancers* para serviços do tipo A. Os clientes comunicam com o respetivo *load balancer*, em que este depois redireciona o pedido para umas das réplicas, dando prioridade às réplicas em execução na mesma região do *load balancer*. Caso estas estejam com mais pedidos que as réplicas do outro local, o pedido é redirecionado para uma dessas réplicas.

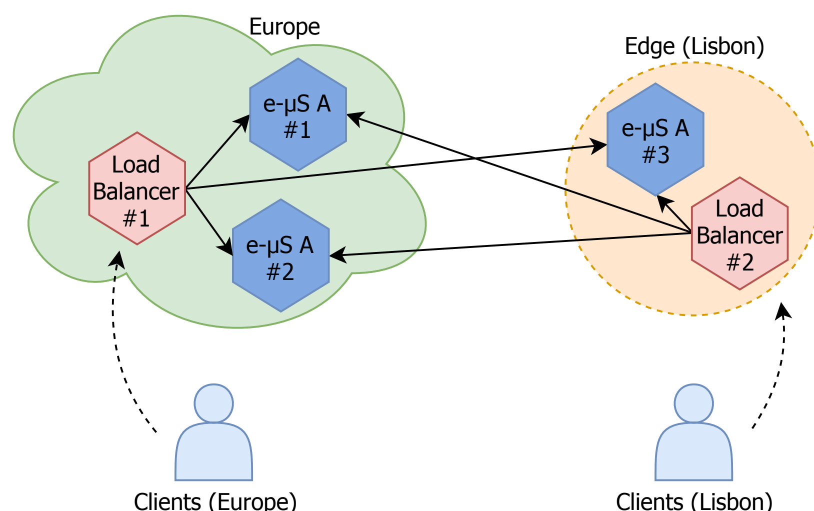


Figura 3.3: Funcionamento do componente de balanceamento de carga.

3.2.3.4 Componente de monitorização

De forma a analisar o estado dos nós e serviços, é necessário um componente que os monitorize de modo que o componente de gestão de micro-serviços consiga realizar as suas funcionalidades e tomar as suas decisões, isto é, replicar, migrar, parar serviços, e iniciar ou parar nós.

Este componente pode ser dividido pelo tipo de recurso que monitoriza, ou seja, existir um componente de monitorização para os nós, e outro para os serviços. Na secção 4.2.5 é detalhado o componente de monitorização dos nós, e em relação à monitorização dos serviços esta é descrita nas secções 4.1.5 e 4.2.4. Apesar de não fazer parte do âmbito desta dissertação, houve necessidade de incorporar algumas funcionalidades de monitorização de forma a validar o sistema.

3.3 Soluções de escalabilidade

Para tratar das situações em que os serviços estão sobrecarregados e/ou com uma performance deficiente, foram consideradas abordagens de escalabilidade horizontal, ou seja, aumentar o número de serviços de um dado tipo em execução (replicação), ou diminuir esse mesmo número quando os serviços e os seus recursos estão a ser subutilizados. Esta foi a abordagem considerada, em detrimento de uma solução com escalabilidade vertical, pelos seguintes motivos:

1. **É mais genérica**: quando existe sobrecarga num dado serviço, apenas é necessário iniciar uma nova réplica com os mesmos recursos, em vez de ter que colocar em execução um novo serviço com uma configuração específica dos recursos, ou alterar a configuração de um serviço já em execução, que pode até nem ser configurável.

2. **A gestão é mais fácil**: todos os serviços são idênticos, isto é, todos têm as mesmas características e recursos disponíveis.

3. **Resolver casos que a escalabilidade vertical não permite**: se um serviço está sobrecarregado ou com uma performance deficiente, aumentar certos recursos pode não resolver o problema, pois pode estar a ser limitado pela própria tecnologia em que o serviço está desenvolvido, ou a nível de infraestruturas (largura de banda).

4. **Utilização de uma abordagem coerente**: se tivesse sido escolhida a abordagem de escalabilidade vertical, essa teria que ser sempre complementada com a de escalabilidade horizontal, pois com esta é possível resolver ou atenuar uma maior percentagem de situações, sendo esse valor menor com uma escalabilidade vertical.

Apesar desta escolha, a abordagem de escalabilidade vertical pode ser benéfica. Se um serviço estiver apenas sobrecarregado a nível de CPU (por exemplo, pode ter sido configurado para utilizar um só núcleo), pode ser possível aumentar a capacidade configurada do CPU do serviço, não sendo necessário aumentar o número de réplicas, podendo ser benéfico ao nível dos recursos totais a serem utilizados.

Em relação aos nós onde os serviços se encontram em execução, foi também considerada uma abordagem de escalabilidade horizontal, isto é, fazer variar o número de nós que o sistema tem disponível, de acordo com o necessário para executar os serviços pretendidos. Quando é necessário um novo nó, este é adicionado ao *cluster*, ficando disponível para a execução de serviços.

Os benefícios da escalabilidade horizontal dos nós passam também pela generalização da solução, ou seja, não é necessário configurar nós específicos para executar os serviços. Por exemplo, quando um nó está a ter uma maior utilização de CPU, poderia ser configurado um nó com maior capacidade a nível de CPU e os serviços passariam a ser executados neste novo nó, terminando o antigo. Esta solução apenas é possível no contexto da *cloud*, mas mesmo assim não resolveria todos os casos, pois existem limites nos recursos que se podem atribuir a cada máquina. Assim, também é relevante a utilização de uma abordagem de escalabilidade horizontal quando são atingidos esses limites máximos, por exemplo, a nível de RAM. Como a gestão dos serviços engloba tanto recursos na *cloud* como na *edge*, a abordagem de escalabilidade vertical teria ainda mais restrições, pois na *edge* não é possível proceder à alteração dinâmica dos recursos disponíveis.

3.4 Interligação dos subcomponentes e funcionalidades: processo de reconfiguração

Tendo os subcomponentes necessários, bem como a abordagem definida em termos de escalabilidade, da interligação daqueles resulta o processo de reconfiguração dinâmica, que é o núcleo do componente de gestão dos micro-serviços. É com a sua execução que

o objetivo do sistema é atingido, ou seja, é possível manter a QoS para as aplicações de micro-serviços, através de replicação ou migração de serviços.

O processo de reconfiguração é dividido por fases, e executado de uma forma periódica. A reconfiguração pode consistir em aumentar/reduzir o número de *containers* de um dado serviço, ou não fazer nada caso se detete que os mesmos estão de acordo com os valores pretendidos. O mesmo se aplica para o caso dos nós do *cluster*. Este processo é periódico, sendo executado de x em x segundos, tendo uma decisão para cada tipo de serviço, e apenas uma decisão para os nós. Por omissão, a decisão limita-se a um incremento/diminuição de uma unidade, caso se verifique essa necessidade. Tal alteração unitária permite: 1. tomar decisões mais precisas, gradualmente; 2. não alterar drasticamente os recursos utilizados evitando oscilações que provoquem comportamentos que prejudiquem a performance do sistema em geral; 3. manter os custos em termos de tempo de criação de *containers*, ou custos monetários na inclusão de novos nós, em valores mais baixos.

Apesar das vantagens, esta opção apresenta a desvantagem de que, se for realmente necessária a incorporação de mais do que uma réplica de um dado serviço num determinado momento, por exemplo, devido a um pico em termos de acessos, o sistema (serviço replicado) demora mais tempo a chegar ao estado pretendido.

O processo de reconfiguração, correspondente ao *'Control Loop'* apresentado na Figura 2.3, está representado de uma forma resumida na Figura 3.4, a parte referente aos serviços, e na Figura 3.5 na parte que diz respeito aos nós, sendo descrito em detalhe cada um dos passos a seguir.

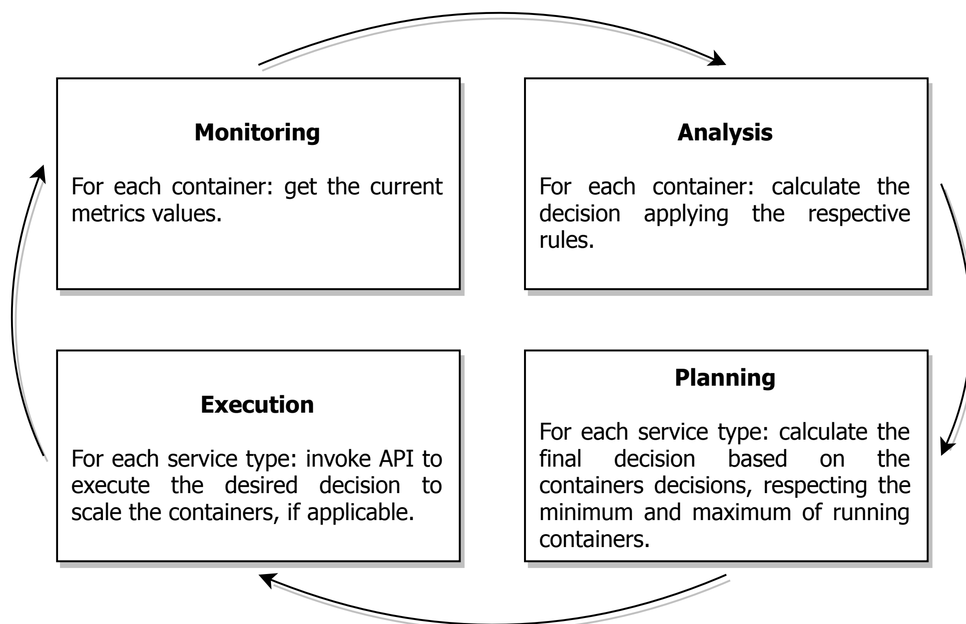


Figura 3.4: Parte do processo de reconfiguração dinâmica referente aos serviços.

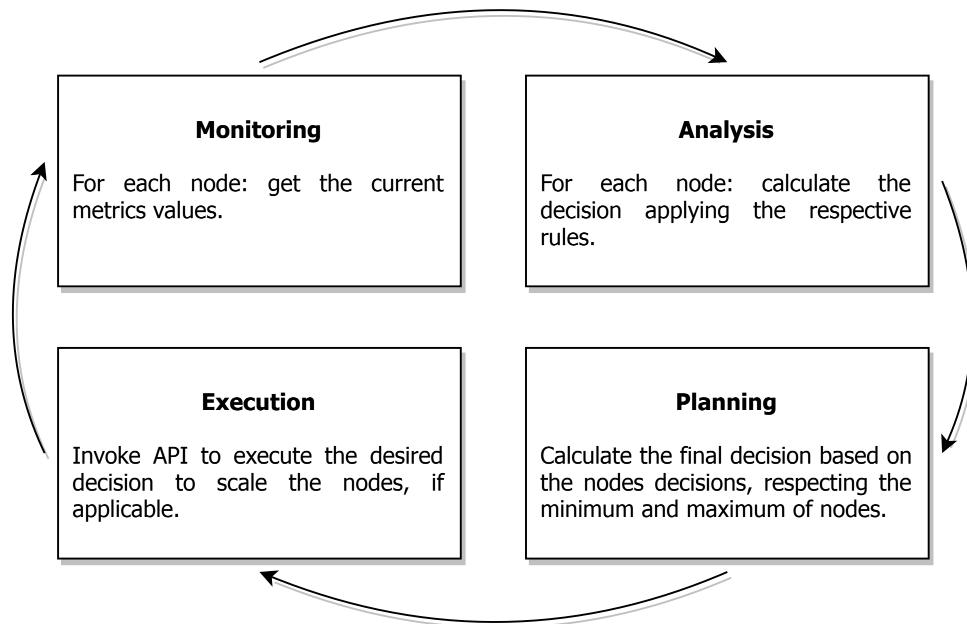


Figura 3.5: Parte do processo de reconfiguração dinâmica referente aos nós.

3.4.1 Monitorização

Serviços São recolhidas as métricas atuais de todos os serviços em execução, guardando o valor em base de dados, onde cada métrica representa uma linha. A Tabela 3.2 representa a estrutura de cada linha.

Tabela 3.2: Composição da linha de uma métrica dos serviços.

Nome	Descrição
ContainerId	Id do container
ServiceName	Nome do tipo de serviço
Field	Nome da métrica
MinValue	Valor mínimo lido
MaxValue	Valor máximo lido
SumValue	Somatório dos valores lidos
LastValue	Último valor lido
Count	Número de leituras
LastUpdate	Data da última atualização

Nós São recolhidas as métricas de cada nó pertencente ao *cluster*, guardando o valor em base de dados. Cada nó tem um conjunto de métricas associado, sendo que cada métrica de um nó corresponde a uma linha na base de dados. Esta apresenta uma estrutura idêntica à dos serviços, sendo visível na Tabela 3.3.

Tabela 3.3: Composição da linha de uma métrica dos nós.

Nome	Descrição
Hostname	Hostname do nó
Field	Nome da métrica
MinValue	Valor mínimo lido
MaxValue	Valor máximo lido
SumValue	Somatório dos valores lidos
LastValue	Último valor lido
Count	Número de leituras
LastUpdate	Data da última atualização

3.4.2 Análise

Serviços Para cada um dos *containers* são executadas as respetivas regras, tendo como valores de entrada as métricas com os diferentes modos para o valor (valor efetivo, média ou % de desvio) recolhidas na fase de monitorização. Depois da execução das regras, é guardado o resultado obtido para cada *container* (replicar, migrar, parar ou nada), sendo estes resultados guardados numa lista para cada tipo de serviço, ordenada por uma prioridade. Esta é calculada, em primeiro pelo resultado da regra, sendo que 'replicar' e 'migrar' têm maior prioridade e 'nada' tem menor e, de seguida, é calculada pela prioridade da regra despoletada, e por último pela quantidade de recursos utilizados (mais recursos utilizados, maior prioridade).

Em relação à replicação ou migração só passará para uma decisão efetiva, se, consecutivamente (número determinado de vezes), for detetada a necessidade de replicação ou migração. Este valor é dois, por omissão, podendo no entanto ser alterado. Da mesma forma, a decisão de parar um réplica será tomada caso, durante um número consecutivo de vezes, tenha sido detetada que existe a necessidade de parar o serviço. Neste caso, o valor por omissão é três, havendo a possibilidade de o alterar.

Nós Para cada nó são executadas as regras configuradas, tendo como valores de entrada as métricas obtidas pela fase de monitorização. Depois da execução das regras, é guardado o resultado obtido para cada nó (iniciar nó, parar nó ou nada), sendo estas guardadas numa lista, ordenada por uma prioridade. Esta prioridade é calculada, em primeiro pelo resultado da regra, sendo que 'iniciar nó' tem maior prioridade e 'nada' tem menor, seguida pela prioridade da regra despoletada e, por último, pela quantidade de recursos utilizados (mais recursos utilizados, maior prioridade).

Para ser tomada uma decisão efetiva de iniciar um novo nó, é necessário ser detetada um número consecutivo de vezes essa mesma necessidade (por omissão o valor é dois, podendo ser alterado). O mesmo se processa no caso de parar um nó, sendo esse número, por omissão três, podendo igualmente ser alterado.

3.4.3 Planeamento

Serviços Tendo a análise realizada para todos os *containers*, nesta fase é decidido o que irá ser feito para cada tipo de serviço: começando-se por consultar o primeiro elemento da lista de prioridades dos *containers* para decidir o que fazer.

Se o primeiro elemento tiver o resultado de:

- Nada: o número de réplicas para o serviço mantém-se inalterado, pois o *container* com maior prioridade tem como decisão 'nada', a decisão com menor prioridade.

- Replicar/migrar: neste caso a decisão é 'replicar' ou 'migrar', mas a lógica do planeamento é idêntica, apenas diferindo em duas questões:

  1. Quando a decisão é 'migrar', o *container* que deu origem a esta decisão é parado/suspenso após um certo período de tempo, e quando a decisão é 'replicar' apenas é adicionada uma nova réplica, ficando as duas em execução;
  2. Quando a decisão é 'replicar', esta só pode acontecer caso o número de réplicas atual seja inferior ao máximo de réplicas permitido para o tipo de serviço.

  O local escolhido para a execução do novo *container* é obtido com base na quantidade de acessos por local. Os locais de proveniência dos acessos são filtrados, para que se tenham em consideração apenas aqueles que contemplem uma quantidade mínima de acessos (valor configurável, por omissão é 15%), sendo depois escolhido o local com base no seguinte:

  1. Se existirem locais sem réplicas em execução, então escolher o local sem réplicas com maior quantidade de acessos;
  2. Se não existirem locais sem réplicas, então escolher o local com maior taxa de acessos por número de réplicas em execução no local.

  Depois da obtenção do local é determinado, com base na prioridade da regra, se o novo *container* irá ser executado num nó da *edge* ou na *cloud*. Se a regra tiver como prioridade a *edge* e tal não for possível, é executado na *cloud*.

- Parar: neste caso a decisão é parar/suspender um *container* que está em execução, diminuindo assim o número de réplicas para o tipo de serviço. Se o número de réplicas atual for superior ao número de réplicas mínimo configurado para o tipo de serviço, caso contrário, o serviço continua em execução. Neste caso é escolhido o último elemento da lista que tem a decisão 'parar', pois podem existir ainda mais *containers* com a decisão 'parar' a consumirem menos recursos que o 1º elemento da lista.

Nós Concluída a análise dos nós, é decidido se existe a necessidade de iniciar um novo nó, se se pode proceder à remoção/paragem de um nó, ou mesmo não ser realizada nenhuma ação.

É analisada a lista de prioridades dos nós para decidir a ação a realizar, sendo que se o primeiro elemento tiver o resultado de:

- Nada: os nós estão em funcionamento de acordo com as regras definidas, não sendo necessário realizar nenhum tipo de ação, pois o nó com maior prioridade tem como decisão 'nada', a decisão com menor prioridade.
- Iniciar nó: neste caso a decisão é adicionar um novo nó ao *cluster*, tendo este como localização a mesma que tem o nó que deu origem à decisão (nó sobrecarregado). Depois decidir-se-á se o nó deverá ser colocado na *edge*, ou na *cloud*, de acordo com o definido na regra. Se for definido como tendo prioridade na *edge* e não existir nenhum nó disponível, será adicionado um nó na *cloud*. De forma a que o nó sobrecarregado fique com mais recursos disponíveis, ou seja, volte a ficar com níveis de recursos abaixo do máximo configurado, são migrados um ou mais *containers* para o novo nó.
- Parar nó: neste caso é removido um nó que está a executar *containers*, devido à sua subutilização de recursos. O nó é removido apenas se o número atual de nós do sistema for superior ao número de nós mínimo configurado, caso contrário o nó continua em funcionamento. Neste caso é escolhido o último elemento da lista que tem a decisão 'parar', pois existe a possibilidade de haver outros nós com a decisão 'parar' a consumir menos recursos que o primeiro elemento da lista. Caso o nó tenha *containers* em execução, estes são migrados para um outro nó disponível num local idêntico, e só depois o nó é removido.

3.4.4 Execução

Serviços Para cada tipo de serviço são executadas as decisões finais (exceto quando a decisão é 'nada'):

- Replicar/migrar: 1. Invocar API que devolve um nó disponível, tendo como parâmetros de entrada o local pretendido, a preferência por um nó na *cloud* ou *edge*, e a utilização média de recursos do *container*; 2. Invocar API para replicar ou migrar um *container* para o nó devolvido no passo anterior.
- Parar: invocar API para parar/suspender o *container*.

Quando é invocada a API para obter um nó disponível para a execução de um determinado *container*, se não existir nenhum com recursos disponíveis (RAM disponível, espaço em disco disponível, etc.), é inicializado um novo nó no local pretendido (ou próximo) e adicionado ao *cluster*.

Nós É executada uma decisão final no conjunto de todos os nós (exceto quando a decisão é 'nada'):

- Iniciar nó: verificar qual a prioridade em termos de localização do novo nó, *edge* ou *cloud*.

  Se for na *edge* e existir um nó disponível para utilização: 1. Invocar API para inicializar todos os componentes de sistema necessários; 2. Em seguida invocar API para adicionar o nó ao *cluster*; 3. Invocar API para migrar um ou mais *containers* do nó sobrecarregado para o novo nó.

  Caso seja na *cloud*: 1. Invocar API para iniciar uma nova VM (nó) na *cloud*; 2. Invocar API para inicializar todos os componentes de sistema necessários; 3. Em seguida invocar API para adicionar o nó ao *cluster*; 4. Invocar API para migrar um ou mais *containers* do nó sobrecarregado para o novo nó.

- Parar nó:

  Caso seja um nó na *edge*: 1. Invocar API para remover o nó do *cluster*; 2. Invocar API para remover os componentes de sistema.

  Caso seja na *cloud* realizar um último passo: 3. Invocar API para eliminar/suspender VM (nó).

3.5 Restrições e limitações

Apesar das vantagens que esta arquitetura apresenta, existem certas restrições e limitações que não podem ser controladas totalmente, nomeadamente a nível da infraestrutura e dos micro-serviços.

3.5.1 Infraestrutura

No caso da infraestrutura, um dos pontos mais relevantes é em termos da largura de banda que, apesar da rede atual já suportar a transferência de grandes quantidades de dados, esta pode ficar congestionada, por exemplo, devido a picos em termos de acessos aos serviços. Uma das formas de tentar minimizar este problema passa pela distribuição dos serviços por vários centros de dados (várias regiões), possibilitando que os dados sejam transferidos por vários locais e por vários canais. Outra possibilidade é ter os serviços em execução mais próximos dos utilizadores, isto permite também reduzir o tráfego de dados que é enviado para a *cloud*.

Uma outra limitação a nível da infraestrutura está relacionada com os nós da *edge*. Normalmente, os dispositivos disponíveis na *edge* para a execução de micro-serviços, têm uma menor capacidade em termos de recursos quando comparados com a *cloud*, por exemplo, a nível de CPU, RAM. Assim, nem todos os serviços poderão ser executados

nestes dispositivos, perdendo-se assim certas vantagens que advêm da utilização de uma infraestrutura mais próxima dos utilizadores.

3.5.2 Serviços

Uma das principais restrições, no suporte à execução de serviços, prende-se com as dependências que os serviços podem ter em relação a outros. As dependências podem também ser de diferentes níveis, podendo ser classificadas consoante a comunicação realizada entre os serviços. Isto é, se existir uma maior comunicação com um serviço 'A' do que em relação a um outro serviço 'B', podemos classificar que existe uma maior dependência com o 'A'. A complexidade que esta questão tem subjacente, leva a que não seja tida em consideração na tomada de decisão.

3.6 Análise comparativa entre a solução e os trabalhos relacionados

As soluções analisadas na secção 2.7, CAUS e ENORM, propõem funcionalidades e integram conceitos partilhados com a solução desenvolvida. Foi realizada uma análise comparativa tendo em conta as funcionalidades que cada solução disponibiliza, sendo esta representada na Tabela 3.4.

Tabela 3.4: Comparação entre as soluções.

Funcionalidade / **Solução**	CAUS	ENORM	μS Management
Utilização da *cloud*	Não [a]	Sim	Sim
Utilização da *edge*	Não	1 nó	Sim
Gestão automática dos nós (*cloud*/*edge*)	Não/Não	Sim/1 nó	Sim/Sim
Auto-scaling de serviços configurável (vários parâmetros de configuração)	1 parâmetro	1 parâmetro [b]	Sim
Local de execução de μS dinâmico (com base na origem dos acessos)	Não	Sim	Sim
Adaptação a picos de acessos (previsíveis/imprevisíveis)	Não/Sim	Não/Sim	Sim/Sim
Suporte às aplicações de μS (comunicação entre μS, *load balancers*)	Não	Não	Sim
Fácil integração de serviços já desenvolvidos	Não	Não	Sim
Gestão de múltiplas aplicações em simultâneo	Não	Sim	Sim

[a] Não diretamente, mas possível integrar num *cluster* do Kubernetes.
[b] Apenas uma réplica com alterações na quantidade de recursos alocados.

Com base na análise comparativa entre as soluções, é possível verificar que a solução desenvolvida apresenta um conjunto de funcionalidades mais completo em comparação com as outras soluções. Este conjunto de funcionalidades engloba tanto aquelas consideradas críticas, como a replicação automática de serviços tendo em consideração a origem dos acessos, e também funcionalidades complementares que auxiliam a execução dos micro-serviços e facilitam o seu desenvolvimentos em certos aspetos, como a comunicação entre serviços e *load balancers*. Estas funcionalidades (ainda que não tratadas na sua plenitude devido à sua complexidade) proporcionam as bases para que no futuro possam ser estendidas e tornar a solução mais completa.

3.7 Cenários de migração e replicação

Existem vários cenários em que a migração ou replicação de serviços pode ser útil para o bom desempenho da aplicação em geral como forma de manter a QoS. Estes cenários têm que ter em conta diferentes fatores, como o tipo de serviço em questão, isto é, se é um serviço de *frontend*, *backend* ou base de dados, as dependências em relação a outros serviços e o tipo de métricas que leva a desencadear a decisão de replicar ou migrar.

3.7.1 Fatores que influenciam a replicação e migração

Os fatores mais relevantes encontram-se divididos em dependências e tipos de serviços e nas métricas retiradas pelo componente de monitorização relativamente aos serviços.

3.7.1.1 Dependências e tipos de serviço

O tipo do serviço, bem como as suas dependências, podem ter influência na decisão final a ser tomada, ou seja, é preciso ter em conta estes fatores, de forma a manter a QoS dos serviços e aplicações.

Frontend No caso de um serviço ser do tipo *frontend*, na maior parte dos casos, este tem dependências em relação a outros serviços de tipo *backend*. Ou seja, é necessário ter em conta a localização de outros serviços no momento da migração ou replicação, de forma a que a comunicação entre eles não seja comprometida. Por exemplo, é necessário considerar que a latência não seja muito elevada, nem que a transferência de grandes quantidades de dados congestione os canais de comunicação.

Backend Num tipo de serviço de *backend* este pode ou não ter dependências, sendo que se as tiver em relação a outros serviços, pode ser tanto de *backend* como de base de dados. É necessário também ter em conta que este tipo de serviços pode ser utilizado por outros, nomeadamente de *frontend* ou *backend*.

Os serviços de *backend* são portanto aqueles que mais fatores têm a considerar quando é necessário replicar ou migrar. Tem de ser analisada a melhor forma de proceder para

que seja benéfico para o próprio serviço, bem como para os outros, ou que não prejudique a sua execução.

Base de dados Trata-se de uma serviço sem dependências, mas que tem outros serviços na sua dependência, sendo estes do tipo de *backend*. É necessário ter em atenção que podem existir vários serviços de *backend* a comunicar com um dado serviço de base de dados, sendo importante, caso seja necessário, a escolha do local onde se irá replicar ou migrar o serviço de base de dados.

É importante referir que um serviço de base de dados pode ter diferentes variantes, ou seja, consoante a situação podemos ter necessidade de uma base de dados completa ou uma base de dados particionada de forma a ter um tamanho mais reduzido, caso existam restrições em termos de espaço.

Este tipo de serviço pode ter também a questão, quando perante a inicialização de uma instância, de popular a base com dados, podendo levar um maior tempo até estar disponível para os outros serviços.

3.7.1.2 Métricas

As métricas, bem como os seus valores, fornecem diferentes tipos de informações, que depois são utilizadas com as outras vertentes para efetuar uma decisão sobre os serviços em que, consoante o seu tipo e o seu valor, podemos ter diferentes vertentes de decisão.

Latência ao serviço A latência é uma métrica que se pretende que tenha um valor o mais baixo possível, sendo que um dos fatores no qual esta varia está relacionado com a distância entre o cliente (utilizador ou serviço) e o serviço pretendido. Teoricamente, e excluindo outros fatores, a latência tende para valores mais elevados quanto maior for esta distância, pelo que uma das possibilidades para tentar diminuir os valores desta métrica é colocar o serviço mais próximo do cliente.

Acessos ao serviço Os acessos ao serviço podem reunir parâmetros como o número de acessos por unidade de tempo, número de acessos em simultâneo, bem como o local de origem desses mesmos acessos. Um serviço pode ser projetado para suportar um número máximo de acessos em simultâneo, podendo ter comportamentos incorretos se este número for ultrapassado, existindo a necessidade de distribuir a carga por outros serviços do mesmo tipo (réplicas).

Largura de banda A largura de banda utilizada pelo serviço é um fator que pode indicar a existência de um volume considerável de dados em trânsito pela rede, sendo eventualmente necessário tomar uma decisão para tentar diminuir este valor.

Recursos utilizados Quando os recursos utilizados pelo serviço (CPU, RAM, etc.) começam a atingir determinados valores, pode ser um indicador de sobrecarga no serviço, sendo necessário distribuir a carga por outras réplicas do mesmo tipo.

3.7.2 Cenários considerados

Os cenários considerados contemplam apenas os casos em que é detetada uma necessidade de replicar ou migrar os serviços, não sendo considerados aqueles em que as réplicas estão a ser subutilizadas, pois nestes apenas é preciso suspender/remover a réplica, consoante os valores previamente definidos.

3.7.2.1 Serviço sem dependências

Considerando apenas serviços sem dependências, podemos ter diferentes cenários, consoante os valores das outras vertentes.

1. **Serviço de *frontend***: um serviço de *frontend*, em situações consideradas normais, não tem um outro serviço que dependa dele, e se tivermos em conta que não tem dependências em relação a outros serviços, a decisão apenas recai na métrica que mais influência tem, sendo este o caso mais simples.

    1.1. **Acessos e recursos utilizados**: caso a métrica que desencadeou um evento no sistema esteja relacionada com os acessos ou com os recursos utilizados, pode significar uma situação de possível sobrecarga na réplica, sendo que a decisão a realizar é a replicação. Esta replicação é efetuada num nó próximo do local onde se registou o maior número de acessos ao serviço, podendo ser num nó na *edge* ou na *cloud*, dependendo das prioridades definidas.

    1.2. **Latência e largura de banda**: caso a métrica que originou um evento no sistema esteja relacionada com a latência ou com a largura de banda, pode significar um comportamento em que a infraestrutura não esteja a dar resposta necessária. A decisão a realizar é migrar o serviço para um local mais próximo dos acessos, de forma a tentar descongestionar a rede e diminuir a latência. Idealmente, se existir um nó disponível na *edge* mais próximo dos utilizadores é esse o local escolhido, caso contrário pode ter de ser migrado para um nó noutra região da *cloud*.

2. **Serviço de *backend***: um serviço de *backend* pode ou não ter outros serviços de que depende, mas considerando os serviços sem dependências, apenas é necessário considerar as métricas com maior influência.

    2.1. **Acessos e recursos utilizados**: neste caso, tal como em serviços de *frontend*, este tipo de métricas pode indicar uma sobrecarga na réplica, sendo a decisão a ser tomada uma replicação de forma a tentar balancear a carga por mais

réplicas. Esta replicação é efetuada num nó próximo do local onde se registou maior número de acessos ao serviço, podendo ser num nó na *edge* ou na *cloud*.

2.2. **Latência e largura de banda**: neste caso, similar aos serviços de *frontend*, pode significar um comportamento em que a infraestrutura não esteja a dar resposta à quantidade de dados transferidos ou que o serviço possa estar distante de onde está a ser acedido. Neste caso, a decisão a tomar é migrar o serviço para um local mais próximo dos acessos, de forma a tentar descongestionar a rede e diminuir a latência. Como são serviços de *backend*, de uma forma geral não são os utilizadores que acedem a este tipo de serviços diretamente, sendo os acessos provenientes de outros serviços de *frontend* ou *backend*. Assim, deve-se migrar para um local próximo da origem dos acessos, mas noutra região da atual de forma a que os canais de comunicação possam corresponder ao necessário.

3. **Serviço de base de dados**: os serviços de base de dados são o último nível em termos do tipo de serviços, não tendo por isso outras dependências relativamente a outros serviços, sendo apenas importante a métrica que desencadeou um evento. Este tipo de serviços é gerido por um componente externo, o componente de gestão de base de dados, tratando-se apenas de cenários da forma como esse componente poderia proceder.

   3.1. **Acessos e recursos utilizados**: quando é detetado que uma réplica de um serviço de base de dados está com muitos acessos ou os recursos que está a utilizar estão elevados, pode significar que existe uma sobrecarga na réplica. A decisão a realizar é então a adição de uma nova réplica deste tipo de serviço.

   3.2. **Latência e largura de banda**: neste caso pode significar um comportamento em que a infraestrutura não esteja a dar resposta à quantidade de dados transferidos ou que o serviço possa estar distante de onde está a ser acedido. A decisão a realizar é migrar o serviço para um local mais próximo dos acessos, de forma a tentar descongestionar a rede e diminuir a latência.

Na Figura 3.6 está representado um cenário em que é decidido a replicação do micro-serviço A. Inicialmente existe apenas uma réplica do micro-serviço A, em execução na *cloud* na região da Europa, mas devido a um elevado volume de acessos é decidido que existe a necessidade de replicar o serviço. Esse serviço é replicado num local próximo da proveniência dos acessos, neste caso é replicado na *cloud* na região da América.

Na Figura 3.7 está representado um cenário em que é decidido a migração do micro-serviço A. Devido a uma latência elevada no acesso ao micro-serviço A, existe a necessidade de realizar uma migração para tentar diminuir esse valor. Ao ter um nó disponível na *edge*, no local onde é detetado um maior volume de acessos, sendo neste caso em Lisboa, é realizada a migração do serviço para esse nó, de forma a que o serviço fique mais perto dos utilizadores.

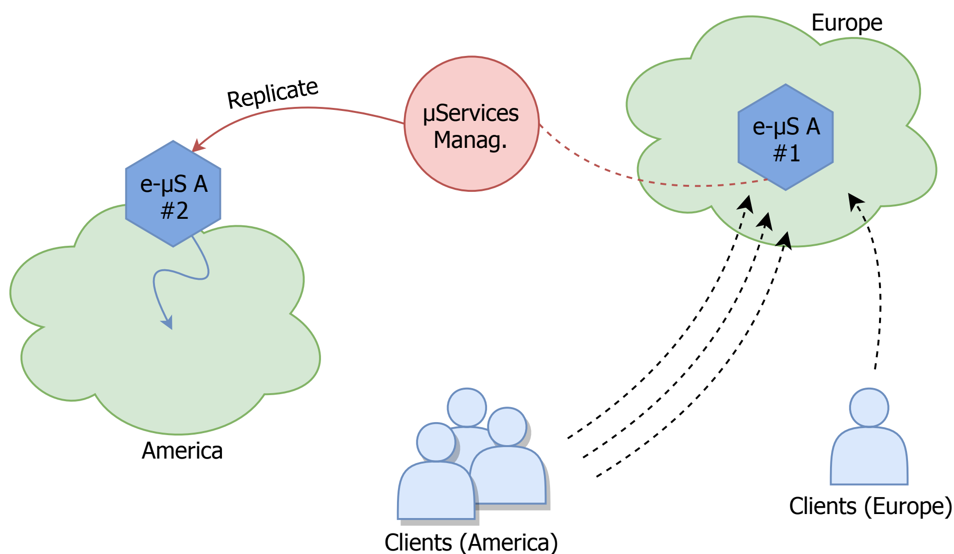


Figura 3.6: Decisão de replicação de um serviço sem dependências.

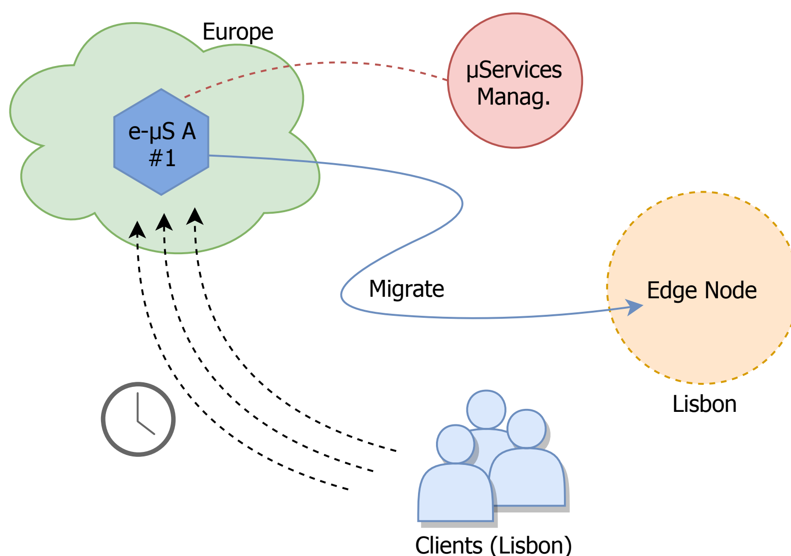


Figura 3.7: Decisão de migração de um serviço sem dependências.

3.7.2.2 Serviço com dependências

Para além de outros fatores, a existência de dependências entre os serviços em processos de replicação e migração torna a decisão mais complexa, pois pode implicar a migração e/ou replicação dessas dependências.

4. **Serviço de *frontend***: um serviço de *frontend* normalmente tem dependências em relação a outros serviços, sendo que a decisão de replicar/migrar tem que ter em conta essas mesmas dependências bem como a métrica com maior influência.

   4.1. **Acessos e recursos utilizados**: neste caso, as dependências não têm influência na decisão, visto que é um cenário que afeta apenas a réplica em questão, sendo

a decisão a realizar igual à descrita no cenário sem dependências (1.1), ou seja, a replicação. Caso os serviços do qual depende tenham um comportamento similar, isto é, elevado número de acessos e sobrecarga nos recursos, a decisão passa por replicar também esses serviços. Tal deve ser feito de preferência no mesmo nó ou num próximo, que evite um aumento indesejável da latência entre esses serviços dependentes.

4.2. **Latência e largura de banda**: a decisão a realizar neste caso, é igual ao do cenário sem dependências (1.2), isto é, a migração do serviço. Um outro aspeto relevante prende-se com os serviços do qual este depende, sendo que o ideal quando da migração do serviço de *frontend* é a migração das suas dependências de forma a minimizar a latência entre serviços. Caso não seja possível a migração total das dependências para o mesmo nó onde será migrado o serviço de *frontend*, é feita a migração para um nó relativamente próximo.

5. **Serviço de *backend***: um serviço de *backend* pode ou não ter dependências em relação a outros serviços, sendo que estes serviços podem ser do tipo *backend* ou base de dados. Para decidir se um determinado serviço de *backend* migra ou replica há que ter em conta as suas dependências (se existirem), bem como os valores e tipos de métricas.

5.1. **Acessos e recursos utilizados**: neste cenário, a decisão é a mesma que a descrita no cenário sem dependências (2.1), a replicação do serviço. Esta decisão é independente de existirem ou não dependências em relação a outros serviços. Contudo se o serviço depender de um outro serviço do tipo de base de dados, o componente de gestão de base de dados terá como possibilidade fornecer uma nova réplica para ficar mais próximo do novo serviço de *backend*. Isto se o serviço de base de dados estiver também com sobrecarga a nível dos acessos e recursos e se o componente de gestão de base de dados assim decidir. Contudo o ideal é a existência de um nova réplica do serviço de base de dados mais próxima da nova réplica do serviço de *backend*. Em relação às outras dependências, só existe a necessidade de replicação se também se detetar um determinado volume de acessos ou estiver a utilizar demasiados recursos.

5.2. **Latência e largura de banda**: a decisão a realizar é igual à descrita no cenário sem dependências (2.2), isto é, a migração do serviço, sendo que, as suas dependências devem também ser migradas (se as tiver), idealmente para o mesmo nó. Caso tal não seja possível, por exemplo, por falta de recursos do nó, migrar para o local mais próximo. Se o serviço tiver dependências em relação a serviços de base de dados, o componente de gestão de base de dados indicará uma réplica adequada para o serviço de *backend* a ser migrado.

6. **Serviço de base de dados**: os serviços de base de dados como não tem dependências em relação a outros serviços, as decisões são idênticas aos cenários já desenvolvidos anteriormente em 3.

Capítulo 4

Implementação

O presente capítulo tem como objetivo detalhar aspetos relevantes a nível da implementação do componente principal de gestão de micro-serviços, na secção 4.1, bem como os subcomponentes necessários para o funcionamento do sistema, na secção 4.2.

4.1 Componente de gestão de micro-serviços

O componente principal de gestão de micro-serviços foi desenvolvido em Java, tendo como base a *framework* Spring Boot. Na Figura 4.1 é representado uma visão detalhada de todos componentes internos que constituem o componente de gestão de micro-serviços, sendo estes detalhados em seguida.

4.1.1 Interface de utilizador (UI)

Para o desenvolvimento da interface de utilizador foi escolhido o React [1], uma biblioteca JavaScript que permite a criação de interfaces web interativas de uma forma simples, para além de ser bastante eficiente ao nível da renderização dos componentes da interface.

4.1.1.1 Funcionalidades

Através da interface é possível executar um vasto conjunto de operações, sendo que estas se encontram subdivididas em várias páginas:

- **Aplicações**: nesta página, representada na Figura 4.2, é possível ver a listagem das aplicações de micro-serviços que estão introduzidas no sistema, bem como adicionar novas. Em cada aplicação existe ainda a opção de inicializar, que lança os micro-serviços correspondentes no local pretendido.

[1] React: https://reactjs.org/

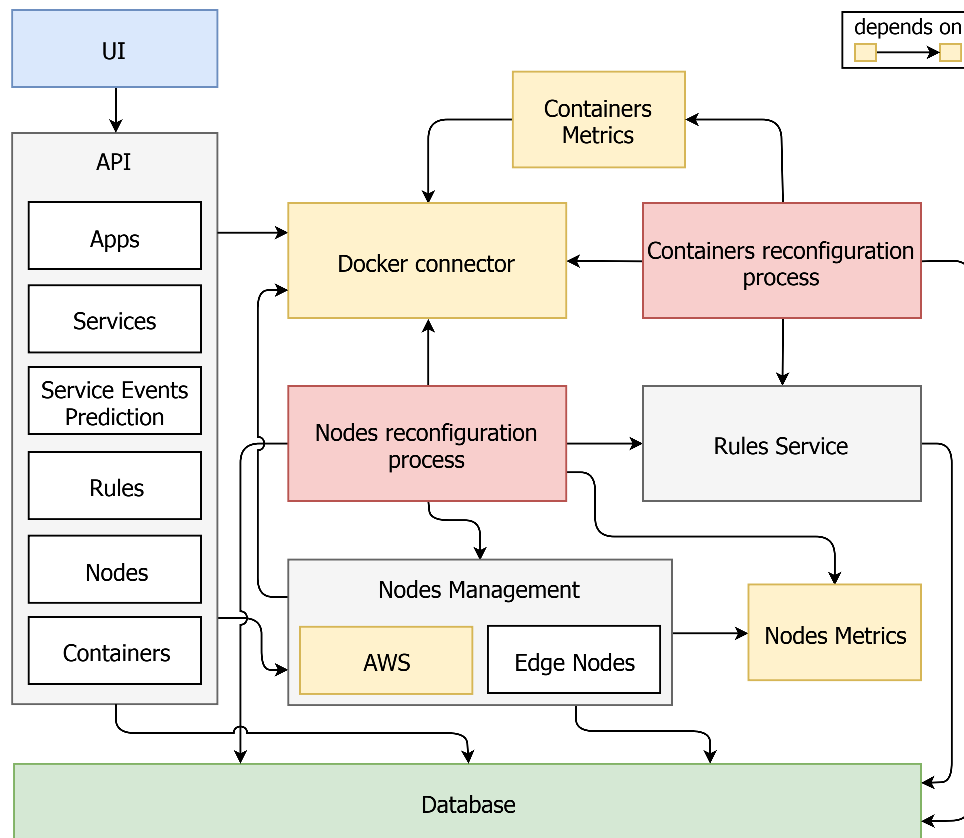


Figura 4.1: Visão detalhada do componente de gestão de micro-serviços.

- **Serviços**: nesta página, representada na Figura 4.3, estão todos os micro-serviços que compõem as aplicações, bem como os outros serviços (componentes) necessários no sistema, sendo possível adicionar novos.

- **Nós na *edge***: nesta página, representada na Figura 4.4, são apresentados os nós da *edge* que estão disponíveis para serem adicionados ao *cluster* de nós para executarem serviços. É possível editar os existentes, bem como adicionar novos.

- **Regiões**: nesta página, representada na Figura 4.5, é possível consultar e configurar os locais, mais precisamente as regiões onde é possível ter nós para executar serviços.

- ***Containers***: nesta página, representada na Figura 4.6, é possível consultar todos os *containers* em execução nos nós que fazem parte do *cluster*, quer pertençam às aplicações ou sejam de sistema. É também possível inicializar, migrar e replicar *containers* manualmente.

- **Nós**: nesta página, representada na Figura 4.7, é possível consultar todos os nós que fazem parte do *cluster*, sendo possível remover um dado nó, ou adicionar um novo.

- **Eureka**: nesta página, representada na Figura 4.8, é possível inicializar novos servidores Eureka, nas regiões pretendidas.

Figura 4.2: Página das aplicações.

Figura 4.3: Página dos serviços.

Figura 4.4: Página de nós da *edge*.

Figura 4.5: Página das regiões.

- ***Load balancers***: nesta página, representada na Figura 4.9, é possível inicializar novos *load balancers* para os serviços nas regiões pretendidas.

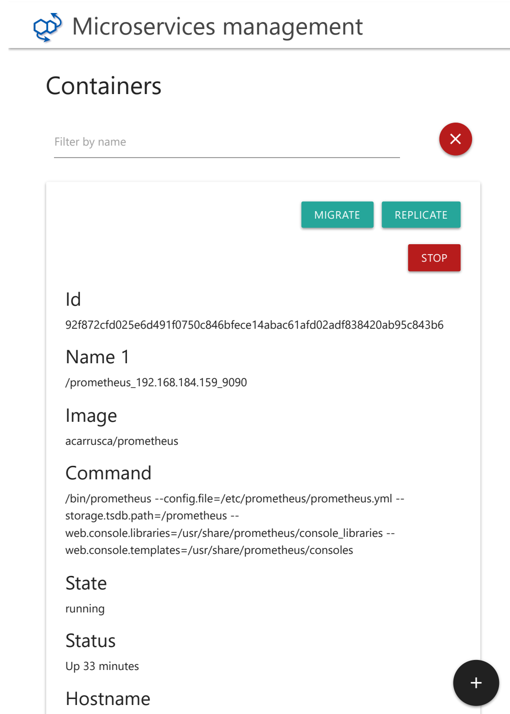


Figura 4.6: Página dos *containers*.

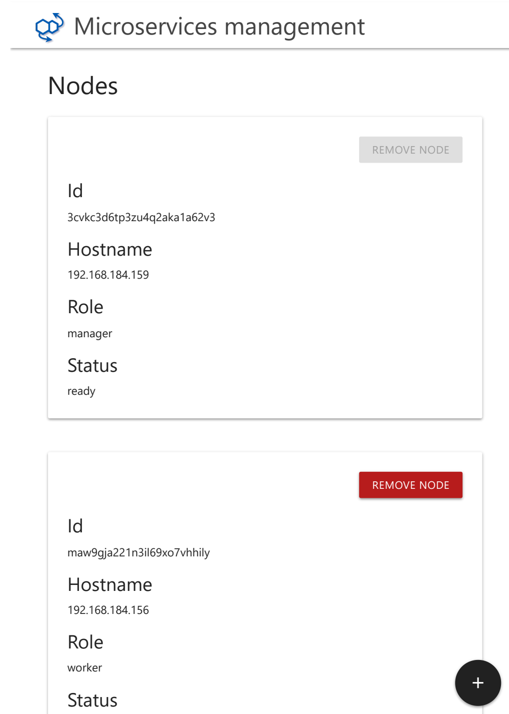


Figura 4.7: Página dos nós.

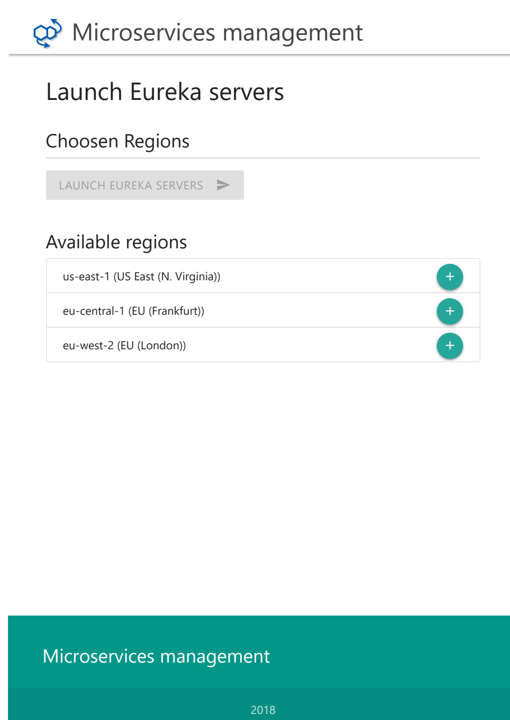


Figura 4.8: Página de inicialização de servidores Eureka.

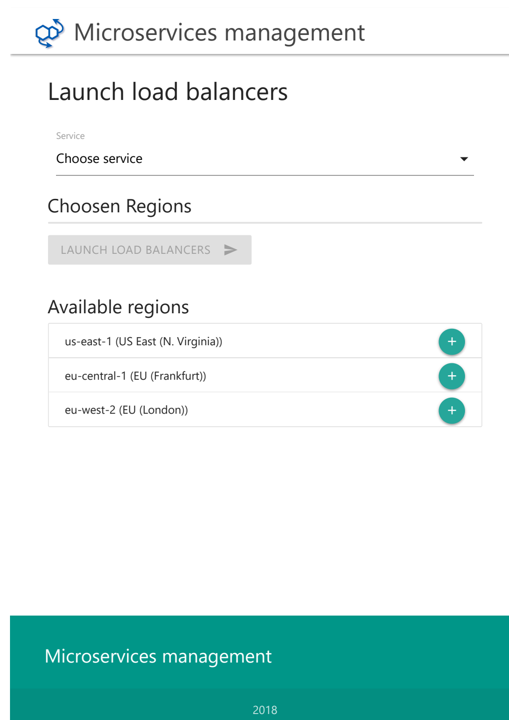


Figura 4.9: Página de inicialização de *load balancers*.

- **Gestão de regras**: nesta página, representada na Figura 4.10, são apresentadas as

sub-páginas onde é possível configurar as regras das aplicações, dos nós e os agendamento de acontecimentos ao nível dos serviços.

- **Gestão de métricas simuladas**: nesta página, representada na Figura 4.11, são apresentadas as sub-páginas onde é possível configurar valores para as métricas quer dos serviços quer do nós. Estas páginas apenas são necessárias para efetuar testes.

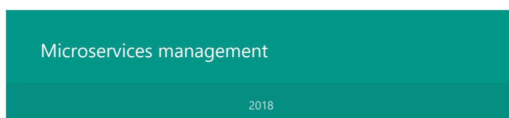
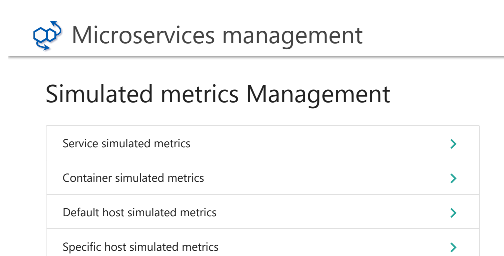


Figura 4.10: Página inicial de gestão das regras.



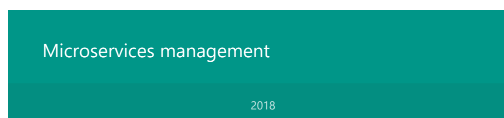


Figura 4.11: Página inicial de gestão das métricas simuladas.

4.1.2 APIs

O componente de gestão de micro-serviços expõe um conjunto de APIs REST que são utilizadas na interface para realizar as funcionalidades pretendidas, podendo estas também ser utilizadas por outros serviços externos para incorporar as funcionalidades do sistema, sem necessidade de utilizar a interface integrada. Estas APIs estão agrupadas de acordo a representação identificada na Figura 4.1.

4.1.3 Base de dados

Para o armazenamento dos dados necessários ao componente de gestão de micro-serviços, tais como as regras e as informações sobre os serviços e aplicações, entre outros, foi utilizado o H2 [2], um sistema de gestão de base de dados relacional desenvolvido em Java, que tem a possibilidade de ser executado em memória.

[2]H2: http://www.h2database.com/

Neste caso a base de dados, fica em execução em memória juntamente com o componente de gestão de micro-serviços, dado tratar-se de um protótipo, mas num caso real esta deveria ser executada em separado do componente e de uma forma persistente.

4.1.4 Gestão de *containers* e nós do *cluster* - Docker

A ferramenta usada para a gestão dos *containers* foi o Docker, que tem de estar instalado em cada nó. Aquela ferramenta disponibiliza as APIs REST para a execução das operações pretendidas. Para estas, foi usado um cliente Java (docker-client [3]) desenvolvido pela Spotify [4], que o utiliza nos seus sistemas. As APIs são usadas pelo *Docker Connector* para realizar as funcionalidades pretendidas, como representado na Figura 4.1.

Em cada nó encontra-se ainda um componente, que é um *proxy* NGINX [5], que tem como principal objetivo introduzir segurança às APIs do Docker. Este componente está em execução em todos os nós sob a forma de um *container*. Quando é adicionado um novo nó ao *cluster*, é executado um comando através de SSH com esse nó para iniciar o *container*, sendo que depois, para executar qualquer operação a nível de *containers* é utilizado este componente.

Em relação aos nós, foi utilizado o Docker Swarm para realizar a sua gestão. É utilizado exclusivamente para o componente de gestão de micro-serviços identificar quais os nós que se encontram disponíveis para a execução de *containers*. Os nós são adicionados ou removidos do *cluster*, consoante o pretendido, através das APIs do Docker. A Listagem 4.1 mostra o código utilizado para adicionar um novo nó ao *cluster*.

Listagem 4.1: Adicionar um novo nó ao *cluster*.

```
1  /**
2  * Método para adicionar um novo nó ao cluster. Alguns métodos foram omitidos.
3  */
4  public class DockerCore {
5
6    private String dockerMasterNodeHostname;
7
8    public boolean joinSwarm(String hostname) {
9      final DockerClient dockerMaster = getDockerClient(dockerMasterNodeHostname);
10     final DockerClient dockerNode = getDockerClient(hostname);
11     boolean alreadyOnSwarm = false;
12     try {
13       Swarm swarm = dockerMaster.inspectSwarm();
14       String joinWorkerToken = swarm.joinTokens().worker();
15       for (Node node : dockerMaster.listNodes()) {
16         if (node.status().addr().equals(hostname)) {// Node already on Swarm
17           alreadyOnSwarm = true;
18         }
```

[3] Docker-client: https://github.com/spotify/docker-client

[4] Spotify: https://www.spotify.com/

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

```
19          }
20          if (alreadyOnSwarm) {
21            return true;
22          } else {// If node is not on swarm
23            this.leaveSwarm(dockerNode);
24            List<String> remoteAddrs = new ArrayList<>(1);
25            remoteAddrs.add(dockerMasterNodeHostname);
26            SwarmJoin swarmJoin = SwarmJoin.builder().listenAddr(hostname)
27              .advertiseAddr(hostname).joinToken(joinWorkerToken)
28              .remoteAddrs(remoteAddrs).build();
29            dockerNode.joinSwarm(swarmJoin);
30            return true;
31          }
32        } catch (DockerException | InterruptedException e) {
33          e.printStackTrace();
34          return false;
35        }
36      }
37    }
```

4.1.5 Obtenção de métricas

As métricas dos *containers* e dos nós são obtidas através de duas classes (representadas na Figura 4.1 como *Containers Metrics* e *Nodes Metrics*), que tratam da comunicação com os respetivos componentes de monitorização. No caso dos *containers* são utilizadas as APIs do Docker, e no caso dos nós são obtidas com recurso ao Prometheus (descrito em detalhe na secção 4.2.5). Foi seguida esta abordagem para permitir no futuro a modificação da forma como as métricas são obtidas. Tal irá permitir a ligação a componentes de monitorização mais completos sem haver muitas mudanças na restante aplicação, visto que a monitorização atual é algo simples devido ao facto de não ser o foco desta dissertação.

Limitações Não é possível obter todas as métricas pretendidas, sendo que para os nós apenas estão disponíveis: a percentagem de RAM utilizada e a percentagem de CPU utilizada. Para os *containers* estão disponíveis: a quantidade (bytes) e percentagem de RAM utilizada, a quantidade (tempo) e a percentagem de CPU utilizada, a quantidade de bytes transferidos e recebidos, e a quantidade de bytes transferidos e recebidos por segundo.

De forma a ultrapassar estas limitações (apenas para efeito de testes ao sistema), foi adicionada uma funcionalidade de simular métricas, para testar o comportamento do sistema quando são detetados certos valores para as métricas.

4.1.6 Serviço de regras (*Rules service*)

As regras tanto a nível dos serviços como dos nós, encontram-se armazenadas numa base de dados. Para o processamento das regras foi utilizado o Drools [6], um sistema de gestão de regras de negócios. As regras são convertidas de forma a serem compatíveis com o Drools, em que estas ficam num sistema de ficheiros em memória, apenas atualizado quando existe alguma alteração nas regras na base de dados. Esta abordagem modular permite que no futuro seja possível substituir o sistema de gestão de regras (Drools) por outro, mantendo as mesmas regras armazenadas em base de dados.

O modelo das regras permite que sejam conjugadas várias métricas simultaneamente. Estas regras têm ainda associada uma prioridade (um número inteiro), sendo que quando menor for esse número, maior prioridade terá a regra.

Em relação aos parâmetros utilizados nas regras, estes podem ser configurados para ter diferentes formas de utilização: pode ser utilizado o valor efetivo lido, a média da métrica, uma fórmula que traduz a percentagem de desvio do valor lido em relação à média, calculada com base na fórmula 4.1, ou pode ser utilizada uma outra fórmula que consiste na percentagem de desvio do valor lido em relação ao valor anterior, sendo calculada com base na fórmula 4.2.

$$\%DeviationOnAverage = \frac{actualValue - average}{average} \times 100 \tag{4.1}$$

$$\%DeviationOnLastValue = \frac{actualValue - lastValue}{lastValue} \times 100 \tag{4.2}$$

Se a percentagem de desvio for negativa, indica que o valor atual é inferior à média (ou último valor) do componente (*container* ou nó) para uma dada métrica, e se for positiva é superior.

Para serem adicionadas regras no sistema é necessário, em primeiro lugar, adicionar as condições, representadas na Figura 4.12, e em seguida adicionar a regra com as respetivas condições, como é apresentado na Figura 4.13.

4.1.6.1 Serviços

As regras dos serviços podem ser configuradas de duas formas: de forma global por aplicação, ou seja, todos os micro-serviços de uma dada aplicação têm associadas essas regras, e/ou de uma forma específica por serviço. As opções de decisão para este tipo de regras são: replicar, parar ou nada, tal como foi discutido na secção 3.2.2.1.

Limitações A decisão de migração, apesar da sua escolha ser possível nas opções de decisão, não se encontra implementada. Contudo o seu comportamento pode ser atingido através da definição de regras de replicação e de paragem de serviços.

[6]Drools: https://www.drools.org/

Figura 4.12: Página para adicionar uma nova condição.

Figura 4.13: Página para adicionar uma nova regra.

Não é possível especificar diretamente nas regras o local para onde um dado serviço será replicado. No entanto, está especificada a *edge* como prioritária, caso exista um nó disponível no local pretendido. Assim, caso a decisão seja replicar é escolhido o nó da *edge*, senão a replicação é feita para um nó na *cloud*.

4.1.6.2 Nós

As regras dos nós podem igualmente ser configuradas de duas formas: de forma geral para todos os nós, e/ou de forma específica para cada nó.

As regras de um determinado nó são depois compostas pelas regras gerais juntamente com as específicas de cada um. As opções de decisão para este tipo de regra são: iniciar (nó), parar (nó) ou nada.

Limitações Não é possível especificar nas regras o local onde adicionar um novo nó, sendo considerado que, caso exista um nó na *edge* disponível no local pretendido, este tem prioridade sobre a *cloud*, caso contrário, é adicionado um novo nó na *cloud*.

4.1.7 Gestão de nós

A gestão dos nós inclui tanto nós na *edge* como na *cloud*, e é realizada por uma classe específica, representada na Figura 4.1 como *Nodes Management*. Esta gestão engloba a

adição de novos nós na *cloud*, que consiste na inicialização de VMs, e também a eliminação de nós que foram removidos do *cluster*, ou seja, suspender/terminar as VMs.

Também trata de atribuir um nó disponível quando é necessário iniciar um novo *container*. A Listagem 4.2 mostra como é feita a procura de um nó disponível para a execução de um *container*. Inicialmente, é pesquisado se existe um nó do *cluster* no local pretendido. Caso não seja encontrado, é procurado na lista de nós da *edge* e, se existir um nó disponível num local idêntico, é esse nó que será atribuído (adicionado em primeiro lugar ao *cluster* e feita a inicialização dos componentes necessários). Caso contrário, será iniciada uma nova VM na *cloud* e será atribuído esse nó.

A verificação se um nó está disponível, isto é, tem recursos suficientes para a execução de um *container*, é realizada com base na memória RAM do nó, sendo definido um valor global para todos os nós que indica a taxa de utilização máxima de memória RAM que um nó pode ter, por exemplo 80%. Quando é requisitado um nó para a execução de um *container*, é verificado que a memória RAM em utilização do nó juntamente com a quantidade de memória RAM média do *container* não ultrapassa a percentagem limite definida, e caso não ultrapasse, significa que esse nó está disponível.

Listagem 4.2: Procura de um nó disponível para a execução de um *container*.

```
1  /**
2  * Método para procurar um nó diponível para execuçao de um container.
3  * Alguns métodos foram omitidos.
4  * Parametros de entrada: regiao, pais, cidade onde se pretende executar o
5  * container e a quantidade de ram media necessaria.
6  */
7  public class HostService {
8
9    public String getAvailableNode(String region, String country, String city,
10     double avgContainerMem) {
11     List<DockerSimpleNode> nodes = dockerCore.getAvailableNodes();
12     List<String> sameRegionsHosts = new ArrayList<String>(nodes.size());
13     List<String> sameCountryHosts = new ArrayList<String>(nodes.size());
14     List<String> sameCityHosts = new ArrayList<String>(nodes.size());
15     for (DockerSimpleNode node : nodes) {
16       String hostname = node.getHostname();
17       HostDetails hostDetails = getHostDetails(hostname);
18       if (hostDetails.getRegion().equals(region)) {
19         if (hostMetricsService
20           .hasHostAvailableResources(hostname, avgContainerMem)) {
21           sameRegionsHosts.add(hostname);
22           if (!country.equals("") && hostDetails.getCountry().equals(country))
23             sameCountryHosts.add(hostname);
24           if (!city.equals("") && hostDetails.getCity().equals(city))
25             sameCityHosts.add(hostname);
26         }
27       }
28     }
```

```
29      if (!sameCityHosts.isEmpty()) {
30        Random rand = new Random();
31        int index = rand.nextInt(sameCityHosts.size());
32        return sameCityHosts.get(index);
33      } else if (!sameCountryHosts.isEmpty()) {
34        Random rand = new Random();
35        int index = rand.nextInt(sameCountryHosts.size());
36        return sameCountryHosts.get(index);
37      } else if (!sameRegionsHosts.isEmpty()) {
38        Random rand = new Random();
39        int index = rand.nextInt(sameRegionsHosts.size());
40        return sameRegionsHosts.get(index);
41      }
42      return addHostToSwarm(region, country, city);
43    }
44  }
```

Limitações Em relação aos nós da *cloud* apenas é suportado a AWS, existindo a possibilidade de no futuro expandir à utilização de novos fornecedores de *cloud*, como o Azure. Também apenas é possível adicionar novos nós numa região da AWS, a 'us-east-1', havendo a possibilidade de adicionar suporte no futuro para todas as regiões disponíveis/pretendidas.

4.1.8 Processo de reconfiguração: nós

O processo de reconfiguração dinâmica referente aos nós, é executado de uma forma periódica. Quando o componente de gestão de micro-serviços é inicializado, inicia um `Timer` onde é executado um método de forma cíclica com um período pré-definido (`hostMonitorInterval`), podendo este ser alterado quando é inicializado o componente. A Listagem 4.3, mostra como é realizada essa mesma inicialização.

Listagem 4.3: Inicialização do `Timer` do processo de reconfiguração dos nós.

```
1   /**
2   * Método que inicia o Timer do processo de reconfiguracao dos nós.
3   * Alguns métodos foram omitidos.
4   */
5   public void initHostMonitorTimer() {
6     long delay = timerScheduleCount == 0 ? hostMonitorInterval : 0;
7     monitorHostTimer.schedule(new TimerTask() {
8       @Override
9       public void run() {
10        try {
11          monitorHostsTask();
12        } catch (Exception e) {
13          e.printStackTrace();
14        }
15      }
```

```
16    }, delay, hostMonitorInterval);
17    timerScheduleCount++;
18  }
```

O processo é executado de acordo com o referido na secção 3.4.

4.1.9 Processo de reconfiguração: *containers*

Tal como o processo de reconfiguração dos nós, para os *containers* procede-se de igual forma, podendo também ser definido o período da execução do `Timer` através do parâmetro `containerMonitorInterval`.

Importa referir que durante o processo é analisado se o número de *containers* para um dado tipo de serviço respeita o mínimo configurado para período atual, quer seja pelo valor pré-definido nas informações do serviço ou pelos valores definidos no Agendamento de acontecimentos (descrito na secção 3.2.2.2). Se o número for inferior, é automaticamente tomada a decisão de adicionar uma nova réplica.

O processo encontra-se explicado detalhadamente na secção 3.4.

4.2 Subcomponentes de sistema

Foram desenvolvidos e integrados vários componentes no sistema de forma a garantir as funcionalidades pretendidas e permitir um sistema mais modular. Estes componentes, externos ao componente de gestão de micro-serviços, incluem funcionalidades como o registo e descoberta de serviços, balanceamento de carga entre os serviços e a monitorização dos nós do sistema.

4.2.1 Componente de registo de serviços (*service registry*)

Este componente é uma aplicação desenvolvida em Java, com recurso à *framework* Spring Boot integrando o Eureka [7] da Netflix. O Eureka é um serviço baseado em REST usado principalmente na *cloud* para descobrir serviços, tendo como objetivo o *failover* de serviços, bem como o balanceamento de carga. Este componente é executado sob a forma de *container*, sendo necessário selecionar quais as regiões onde estarão presentes as réplicas do *service registry*, através da página representada na Figura 4.8.

4.2.2 Componente para registar e descobrir serviços

Este componente é desenvolvido em Go e destina-se a ser utilizado em cada micro-serviço que necessite de comunicar com outros serviços ou ser descoberto por outros serviços. Disponibiliza três APIs REST que apenas podem ser acedidas dentro do *container*, ou seja, não expõe as APIs para fora do *container*. As APIs disponibilizadas estão presentes na Tabela 4.1.

[7] Eureka: `https://github.com/Netflix/eureka`

Tabela 4.1: APIs para registar e descobrir serviços.

Método	Tipo	*Endpoint*	Descrição
GetAllAppsByName	GET	/apps/{appName}/all	Devolve todos os *endpoints* de uma dada aplicação (serviço) tendo como nome 'appName'.
GetAppByName	GET	/apps/{appName}	Devolve um único *endpoint* de uma dada aplicação (serviço) tendo como nome 'appName'.
Register	GET	/register	Regista a serviço no *service registry*.

Este componente pode registar o serviço automaticamente no *service registry* (servidor Eureka), depois de atingido o período de tempo configurado (por omissão é 30 segundos). Pode ser também o serviço a comunicar com o componente através da API 'Register', fazendo o registo manualmente apenas quando o serviço está totalmente inicializado. Periodicamente (o valor por omissão é 30 segundos), o componente verifica se a aplicação (serviço) se encontra em execução, e caso esteja, envia um *heartbeat* para o *service registry* de forma a informar que o serviço se encontra ativo. Se o componente detetar que o serviço não está em execução um determinado número de vezes (o valor por omissão é 3), este elimina o registo do serviço no servidor Eureka e termina o *container*.

Quando um *container* que possui este componente é terminado, o registo do serviço no *service registry* é anulado, ficando indisponível para a utilização por outros serviços.

Quando é invocada a API 'GetAppByName', o componente escolhe o serviço com base no algoritmo 1. Este algoritmo tem como objetivo a escolha do melhor serviço possível, dando como preferência o serviço com a localização mais semelhante com o serviço que invocou a API. O componente vai guardando os *endpoints* dos serviços, de forma a que não haja necessidade de comunicar tantas vezes com o *service registry*. Os *endpoints* são considerados válidos durante um período de tempo configurado (por omissão é 10 segundos).

4.2.3 Componente de balanceamento de carga

Este componente é um *load balancer* NGINX, que permite fazer o balanceamento de carga dos serviços de *frontend*. É executado dentro de um *container* em que está ainda presente uma aplicação que foi desenvolvida em Go, para permitir a reconfiguração dinâmica do *load balancer*. Ou seja, é possível adicionar ou remover réplicas em tempo de execução com recurso a APIs REST. As APIs disponíveis estão presentes na Tabela 4.2.

O *load balancer* NGINX utiliza a técnica de balanceamento de carga *Least Connections*, que redireciona os pedidos dos clientes para a réplica com menor número de conexões ativas. Esta é complementada com um peso em cada réplica, onde as réplicas localizadas na mesma região que o *load balancer* têm um peso maior, e as réplicas de outras regiões um menor, podendo estes valores ser configurados. Foi seguida esta abordagem para que fosse

Algoritmo 1 Escolha de uma réplica por tipo de serviço.

```
state
    thisContinent                                   ▷ continente do serviço
    thisRegion                                      ▷ região do serviço
    thisCountry                                     ▷ país do serviço
    thisCity                                        ▷ cidade do serviço

function GetAppByName(appName)
    appEndpoint ← ''
    apps ← eureka.getAppsByName(appName)
    appsContinent ← findByContinent(apps, thisContinent)
    appsRegion ← findByRegion(apps, thisRegion)
    appsCountry ← findByCountry(apps, thisCountry)
    appsCity ← findByCity(apps, thisCity)
    if appsCity.lenght > 0 then
        index ← random(0, appsCity.lenght)
        appEndpoint ← appsCity[index].endpoint
    else if appsCountry.lenght > 0 then
        index ← random(0, appsCountry.lenght)
        appEndpoint ← appsCountry[index].endpoint
    else if appsRegion.lenght > 0 then
        index ← random(0, appsRegion.lenght)
        appEndpoint ← appsRegion[index].endpoint
    else if appsContinent.lenght > 0 then
        index ← random(0, appsContinent.lenght)
        appEndpoint ← appsContinent[index].endpoint
    else
        index ← random(0, apps.lenght)
        appEndpoint ← apps[index].endpoint
    end if
    return appEndpoint
end function
```

Tabela 4.2: APIs da aplicação de reconfiguração do *load balancer*.

Método	Tipo	*Endpoint*	Descrição
GetServers	GET	/servers	Devolve a lista dos servidores de *frontend* (réplicas).
AddServer	POST	/servers	Adiciona um ou mais servidores de *frontend*.
DeleteServer	DELETE	/servers	Elimina um servidor de *frontend*.

dada prioridade às réplicas que se localizam na mesma região que o *load balancer*, mesmo que o número de conexões ativas nestes seja ligeiramente superior. Isto porque é preferível utilizar uma réplica mais próxima do utilizador do que redirecionar o pedido para uma réplica noutra região, onde o tempo de resposta iria ser maior. Por exemplo, considerando um peso de 5 e 115 conexões ativas para uma réplica A da mesma região que o *load balancer* e um peso de 4 e 100 conexões ativas para uma réplica B de outra região. Neste

caso, se fosse utilizada apenas a técnica *Least Connections*, o pedido seria redirecionado para réplica B, mas como as réplicas têm um peso associado, a decisão é baseada no valor $conexões \div peso$. Obtém-se assim $115 \div 5 = 23$, para a réplica A e $100 \div 4 = 25$, para a réplica B, sendo o pedido redirecionado para a réplica A, pois apresenta o valor mais baixo.

O *load balancer* tem ainda o módulo 'ngx_http_geoip_module' [8], que permite obter a localização do cliente que realizou o pedido, sendo depois esta informação enviada no *header* do pedido para as réplicas, de forma a que cada réplica saiba a origem do pedido. Sempre que é adicionada ou eliminada uma réplica é gerado um novo ficheiro de configuração pela aplicação de reconfiguração para o *load balancer* NGINX utilizar.

Estas APIs são utilizadas apenas pelo componente de gestão de micro-serviços, podendo ser configuradas para utilizarem autenticação, limitando assim o acesso exterior. São utilizadas quando é adicionada, ou removida uma réplica de um serviço do tipo *frontend*.

4.2.4 Componente para monitorização de pedidos

É um componente desenvolvido em Go, presente em cada nó com o objetivo de guardar as informações sobre o número de pedidos realizados aos serviços e qual a sua origem.

As informações são enviadas periodicamente pelo componente para registar e descobrir serviços de cada serviço. Elas são depois consultadas pelo componente de gestão de micro-serviços, para ter as informações sobre a quantidade e origem dos pedidos aos serviços e assim, se for necessário replicar, podê-lo fazer no local mais adequado. As informações estão estruturadas de acordo a Tabela 4.3.

Tabela 4.3: Estrutura dos dados recolhidos na monitorização dos pedidos.

Nome	Descrição
Service	Serviço requisitado
Continent	Continente do cliente/serviço que fez o pedido
Region	Região do cliente/serviço que fez o pedido
Country	País do cliente/serviço que fez o pedido
City	Cidade do cliente/serviço que fez o pedido
Count	Número de pedidos
Timestamp	*Timestamp* dos pedidos

São disponibilizas duas APIs REST para a consulta dos dados e para a sua introdução. As APIs estão descritas na Tabela 4.4.

4.2.5 Componente para monitorização dos nós

Devido ao facto do componente de gestão de micro-serviços necessitar de informações sobre o estado dos nós (a nível dos recursos), foi desenvolvido um componente de monitorização simplificado que permite obter métricas reais dos nós. Tal permite simular

[8] ngx_http_geoip_module: http://nginx.org/en/docs/http/ngx_http_geoip_module.html

Tabela 4.4: APIs para consultar e adicionar informações sobre os pedidos aos serviços.

Método	Tipo	*Endpoint*	Descrição
GetAllInfo	GET	/all/top/{seconds}	Lista a informação dos pedidos de todos os serviços nos últimos segundos (*seconds*).
AddInfo	POST	/add	Adiciona informações sobre os pedidos.

o correto comportamento do sistema, apesar de não fazer parte do âmbito, sendo essas funcionalidades fornecidas por um componente tratado numa outra dissertação.

Foi utilizado o Prometheus [9], um sistema *open-source* de monitorização e alertas, ficando este em execução sob a forma de *container* em cada nó, inicializado quando um novo nó é adicionado ao *cluster*. Para aceder às métricas pretendidas dos nós, foi utilizado um exportador, o Node exporter [10], que permite obter métricas de hardware e do sistema operativo em sistemas *NIX. O Node exporter fica em execução diretamente como um processo na máquina, em que depois o Prometheus é configurado para utilizar esse exportador, isto através do *endpoint* do Node exporter. A Figura 4.14 mostra um diagrama de como é feita a interação para obter as métricas de um determinado nó, pelo componente de gestão de micro-serviços com recurso ao Prometheus e ao Node exporter.

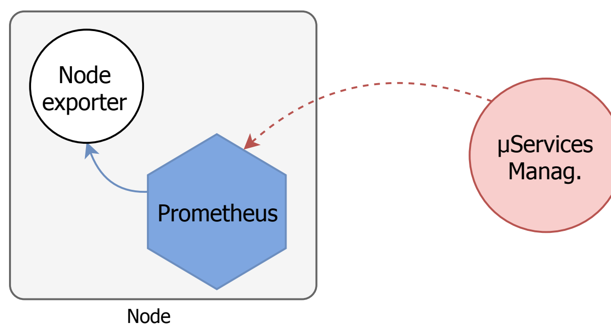


Figura 4.14: Utilização do Prometheus pelo componente de gestão de micro-serviços para a obtenção das métricas de um nó.

[9]Prometheus: https://prometheus.io/
[10]Node exporter: https://github.com/prometheus/node_exporter

Capítulo 5

Validação e Avaliação experimental

Na secção 5.1 deste capítulo é apresentado o caso de estudo, a aplicação *Sock Shop*, que foi utilizada para a validação realizada à solução proposta. São ainda descritas na secção 5.2 as adaptações que foram necessárias realizar nos micro-serviços que compõem a aplicação *Sock Shop*, de forma a cumprirem os requisitos da arquitetura proposta. Por fim, na secção 5.3, encontra-se descrita a avaliação dos resultados.

5.1 Caso de estudo

As avaliações ao protótipo desenvolvido do sistema foram realizadas através de uma aplicação de micro-serviços previamente desenvolvida denominada *Sock Shop* [72], que é composta por oito micro-serviços, implementados utilizando linguagens diferentes, sendo que alguns deles utilizam serviços de base de dados:

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

7. **Queue-master**: processa a fila de encomendas. Desenvolvido em Java.

8. **Frontend**: aplicação de *frontend* que interliga todos os outros micro-serviços. Desenvolvido em NodeJS.

Para além dos oito micro-serviços e as suas respetivas base de dados, é ainda utilizado o RabbitMQ [1] como serviço de mensagens.

A Figura 5.1 mostra, de uma forma simplificada, todos os micro-serviços que compõem a aplicação, bem como os padrões de interação que existem entre eles.

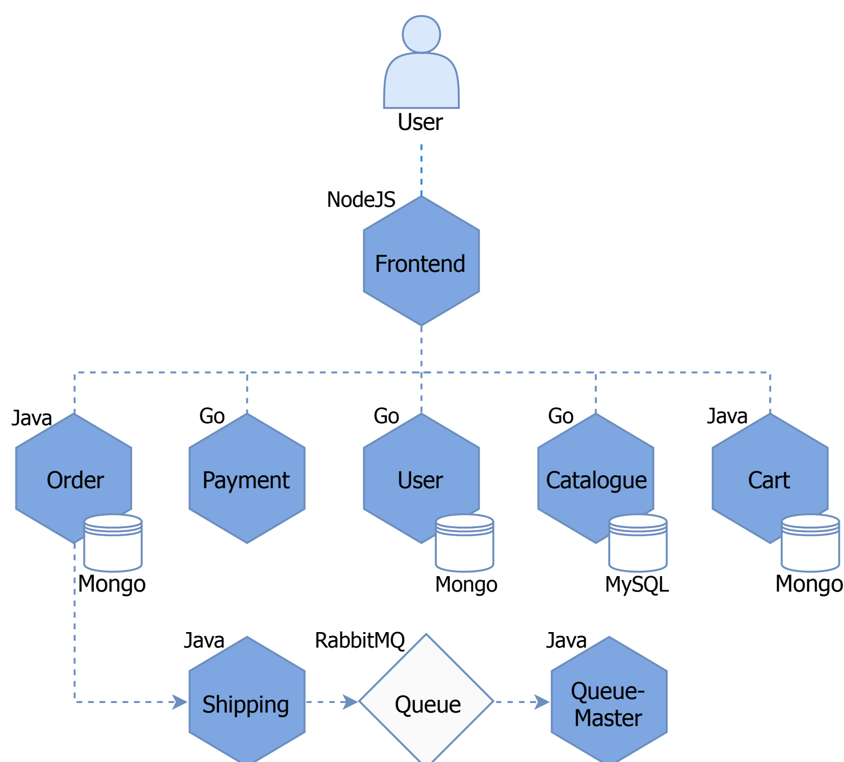


Figura 5.1: Arquitetura da aplicação *Sock Shop*.

5.2 Adaptação do caso de estudo à arquitetura da solução

Os micro-serviços da aplicação *Sock Shop* sofreram ligeiras alterações, tais como, a forma de comunicação entre os serviços. Inicialmente, a comunicação entre os micro-serviços era feita com recurso a *endpoints* estáticos inseridos diretamente no código. Num sistema tão dinâmico como o proposto, em que podem existir múltiplas réplicas de um dado serviço e em que as réplicas podem ser eliminadas (levando a que os serviços deixem de poder continuar a comunicar) tal seria inviável.

Os micro-serviços foram portanto alterados para utilizarem o componente para registar e descobrir serviços (descrito na secção 4.2.2), para a comunicação entre eles. Este

[1] RabbitMQ: https://www.rabbitmq.com/

componente tem também a funcionalidade de registar os respetivos serviços no *service registry*.

A Listagem 5.1 mostra como originalmente era obtido o *endpoint* do serviço Shipping no serviço Orders. O código foi alterado para uma nova versão representada na Listagem 5.2. As novas imagens Docker dos micro-serviços passaram a conter, para além da respetiva aplicação (micro-serviço), o componente de registar e descobrir serviços. Quando é inicializado um novo *container*, a aplicação e o componente são ambos colocados em execução, tendo sido utilizado um *script* de inicialização para esse fim.

Listagem 5.1: Versão original da obtenção do *endpoint* do serviço Shipping no serviço Orders.

```
1  /**
2  * Versao original para obter o endpoint do serviço Shipping no serviço Orders.
3  */
4  public class OrdersConfigurationProperties {
5    public URI getShippingUri() {
6      return new ServiceUri(new Hostname("shipping"), "/shipping").toUri();
7    }
8  }
```

Listagem 5.2: Versão modificada da obtenção do *endpoint* do serviço Shipping no serviço Orders.

```
1  /**
2  * Versao modificada para obter o endpoint do serviço Shipping no serviço
3  * Orders. Utiliza uma biblioteca sendo apenas necessário passar o nome do
4  * serviço.
5  */
6  public class OrdersConfigurationProperties {
7    private AppsApi apiInstance = new AppsApi();
8
9    private String getAppEndpoint(String appName) {
10     App app = apiInstance.getAppsByName(appName);
11     return app.getEndpoint();
12   }
13
14   public URI getShippingUri_V2() {
15     return new ServiceUri(getAppEndpoint("shipping"), "/shipping").toUri();
16   }
17 }
```

Uma outra alteração nos micro-serviços, está relacionada com o registo do serviço no *service registry*. Para tal, é feita uma chamada à API do componente de registar e descobrir serviços, que permite registar o serviço. Esta chamada apenas é realizada quando os serviços estão totalmente inicializados (e prontos a processar pedidos).

5.3 Avaliação experimental

A avaliação experimental foi realizada através de testes de carga à aplicação *Sock Shop*, em que foram analisados os tempos de respostas das principais funcionalidades da aplicação. A avaliação foi feita quer em cenários sem regras configuradas para a replicação dos serviços no componente de gestão de micro-serviços, quer em cenários com regras configuradas para a sua replicação, de forma a percebermos os potenciais benefícios fornecidos pelo componente de gestão de micro-serviços.

Um aspeto relevante em relação aos micro-serviços que dependem de um serviço de base de dados é que, devido à falta do componente de gestão de base de dados (discutido na secção 3.1), quando é decidido que existe necessidade de replicar um destes micro-serviços, a base de dados associada é também replicada integralmente. Assim, apenas existe uma base de dados de cada tipo por nó, significando que pode existir um ou mais micro-serviços a utilizar uma mesma base de dados, caso se encontrem em execução no mesmo nó.

5.3.1 Testes de carga aos micro-serviços da aplicação *Sock Shop*

Para realizar os testes de carga foi utilizado o k6 [2], uma ferramenta para realizar testes de carga que podem ser executados, quer localmente quer na *cloud*, com recurso ao serviço Load Impact Insights [3].

O k6 utiliza o conceito de utilizadores virtuais (vus), que representa o número de utilizadores em simultâneo a efetuar os testes num dado período de tempo.

Os testes foram executados na *cloud*, concretamente na AWS, utilizando máquinas t2.micro com o sistema operativo Ubuntu Server 16.04 (64-bit), tendo 1 vCPU (2.5 GHz, Intel Xeon Family) e 1 GB de memória RAM. O componente de gestão de micro-serviços é executado isoladamente numa VM, sendo os micro-serviços executados noutras VM que fazem parte do *cluster* de nós. A configuração inicial cria o *cluster* com 10 nós na região US-East-1 (Virgínia do Norte), sendo depois decidido pelo componente de gestão de micro-serviços se existe ou não necessidade de adicionar mais. Para simular os nós da *edge*, foram igualmente utilizadas máquinas da AWS, com as mesmas características, sendo colocadas noutras regiões mais próximas do local da realização dos testes.

Foi utilizado o Load Impact Insights para que fosse possível escolher a origem dos acessos aos serviços.

Nos testes, sem replicação e com replicação na *cloud*, os serviços encontram-se em execução na *cloud* (Virgínia do Norte), onde os pedidos têm origem a partir de Londres com destino o *load balancer* em execução na *cloud*, como representado na Figura 5.2.

Nos testes, com replicação na *cloud* e na *edge*, os serviços encontram-se inicialmente em execução na *cloud* (Virgínia do Norte), e posteriormente (após uma decisão de replicação) em nós da *edge* em Londres. Existem pedidos provenientes de Portland com destino ao

[2] k6: `https://docs.k6.io/docs`

[3] Load Impact Insights: `https://loadimpact.com/insights/`

load balancer em execução na *cloud*, e pedidos provenientes de Londres com destino ao *load balancer* em execução na *edge* em Londres. Este tipo de teste está representado na Figura 5.3.

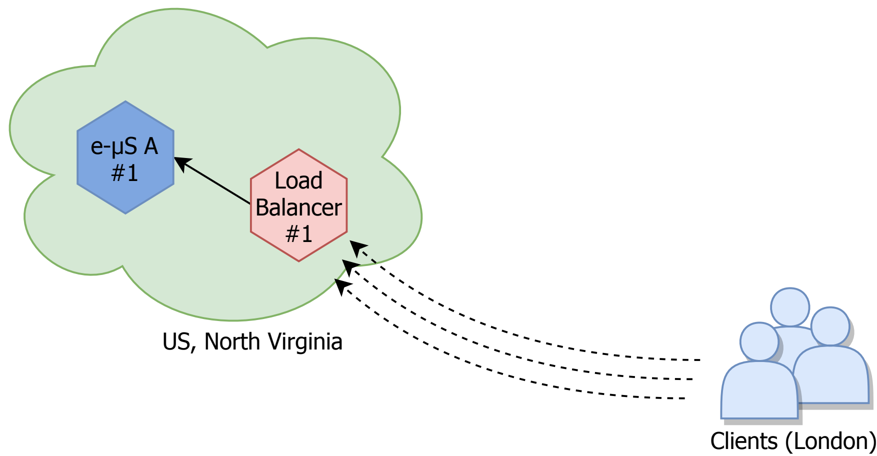


Figura 5.2: Cenário dos testes de carga sem replicação e com replicação na *cloud*.

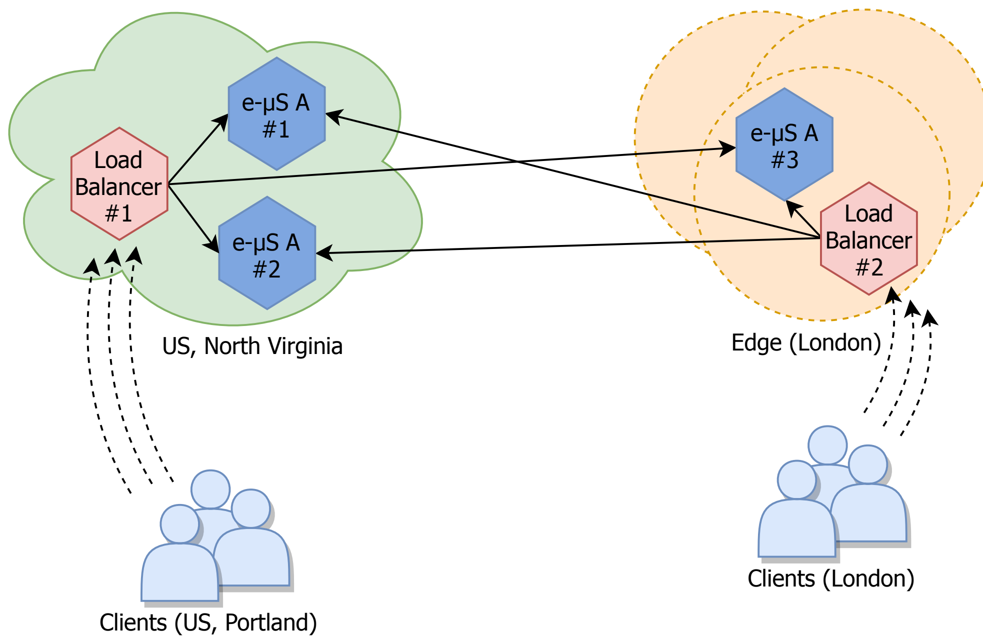


Figura 5.3: Cenário dos testes de carga com replicação na *cloud* e na *edge*.

Em relação ao componente de gestão de micro-serviços, foram feitas as seguintes configurações:

- O processo de reconfiguração ocorre de 25 em 25 segundos, sendo que são retiradas as métricas dos serviços em cada iteração deste processo;
- A replicação está configurada para ocorrer quando for detetada a necessidade de replicação em 2 iterações consecutivas do processo de reconfiguração;

- A remoção das réplicas é despoletada quando for detetada que uma réplica pode ser removida em 3 iterações consecutivas.

5.3.1.1 Catálogo dos produtos

Neste caso os micro-serviços acedidos são o Frontend e o Catalogue. É simulado o comportamento no acesso à página do catálogo, em que primeiro são consultados os dados do catálogo, isto é, os produtos existentes. O utilizador acede inicialmente ao micro-serviço Frontend, e este em seguida faz um pedido ao micro-serviço Catalogue. Quando são obtidos os dados do catálogo, são feitos novamente vários pedidos para obter as imagens associadas aos produtos, seguindo o mesmo procedimento.

Nos testes seguintes, o valor do tempo de resposta (*response time*) diz respeito à obtenção dos dados do catálogo e o valor da duração do grupo (*group duration*) está ligado ao tempo necessário para obter todas as imagens do catálogo (sendo estes pedidos feitos em paralelo, para simular o comportamento de um *browser*).

Os testes têm a duração de 7 minutos, onde foram introduzidas variações no número de utilizadores, aumentando gradualmente de 1 até 50 utilizadores, e reduzindo depois gradualmente novamente até 1 utilizador. Cada utilizador efetua pedidos, entre 1 a 3 segundos (aleatoriamente), após a receção da resposta.

Teste sem replicação na *cloud* Este teste, representado na Figura 5.4, pretende mostrar o comportamento da aplicação quando não existe replicação dos serviços.

Nas Figuras 5.5 e 5.6, é possível observar os valores das métricas dos serviços Frontend e Catalogue, durante a execução deste teste.

Como é possível verificar, com o aumento do número de utilizadores, verificou-se também um aumento dos tempos de resposta, tanto na obtenção dos dados do catálogo como nas imagens do mesmo. Este aumento começa a acentuar-se mais, por volta dos 40 utilizadores (vus) e os tempos de resposta degradam-se substancialmente acima desse número, sobretudo na duração do grupo (obtenção das imagens). Os valores do teste encontram-se na Tabela 5.1.

Tabela 5.1: Valores do acesso ao catálogo sem replicação.

Tipo de valor	Média (s)	Desvio padrão (s)	Máximo (s)
Tempo de resposta	9,2	13,8	57,4
Duração do grupo	31,7	38,5	120,0

Em relação ao comportamento dos micro-serviços, isto é, o valor das métricas apresentadas, é possível observar que as métricas com um aumento mais repentino, em ambos os casos, foram a taxa de utilização de CPU, e a transmissão de bytes por segundo. No caso da taxa de utilização de CPU, apresentam, contudo, valores completamente distintos, pois o serviço Frontend, na Figura 5.5, apresenta como pico máximo de utilização de

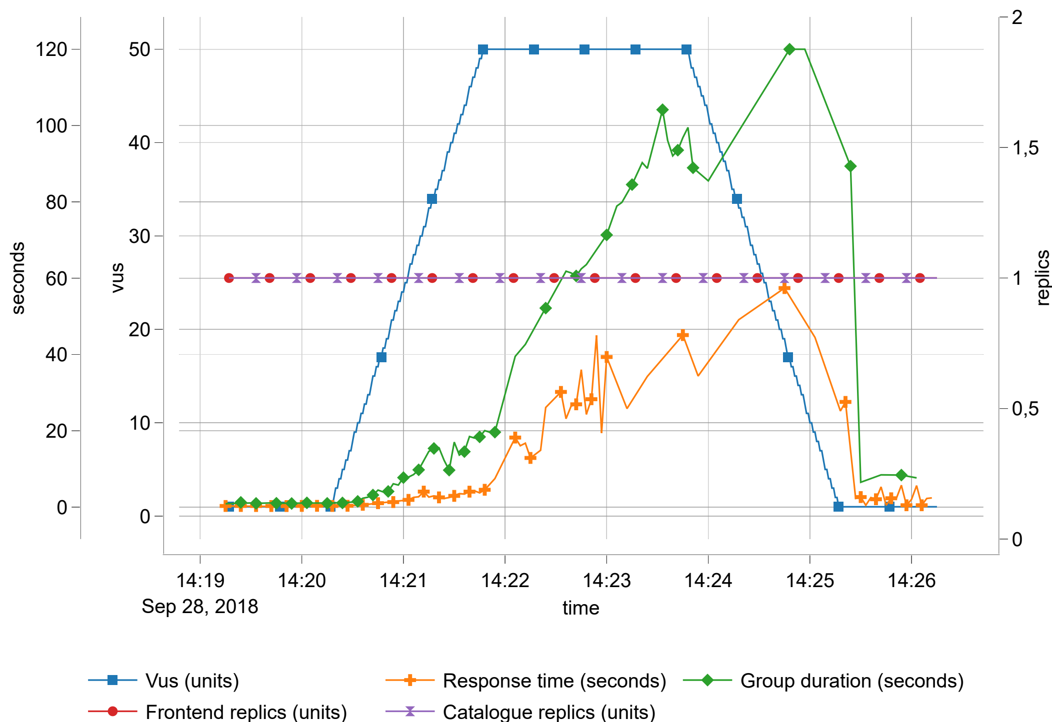


Figura 5.4: Teste de carga ao acesso do catálogo do produtos, sem replicação.

CPU, cerca de 70%, pico esse que, no Catalogue, passa pouco dos 4%, como é visível na Figura 5.6.

No serviço Frontend existe uma outra métrica relevante: a receção de bytes por segundo. Este indicador tem valores próximos da transmissão de bytes por segundo, já que os dados que o serviço Frontend recebe do serviço Catalogue são reenviados para os utilizadores. A taxa de utilização de memória RAM não é uma métrica muito reveladora, pois apresenta valores sem grandes oscilações. Em relação à utilização efetiva de memória RAM, esta vai aumentando ao longo do teste, podendo ser uma métrica a considerar.

No serviço Catalogue, tanto os valores de utilização efetiva de memória RAM, como a sua taxa de utilização apresentam comportamentos aproximados, aumentando gradualmente ao longo do teste. A receção de bytes por segundo, neste caso é praticamente irrelevante, visto que se mantêm em valores muito próximos do 0, ao longo de todo o teste.

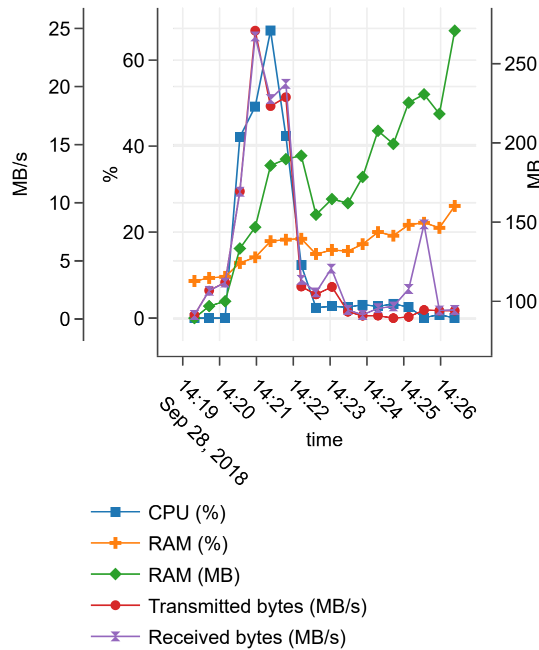


Figura 5.5: Métricas do serviço Frontend, durante o teste de carga ao acesso do catálogo dos produtos, sem replicação.

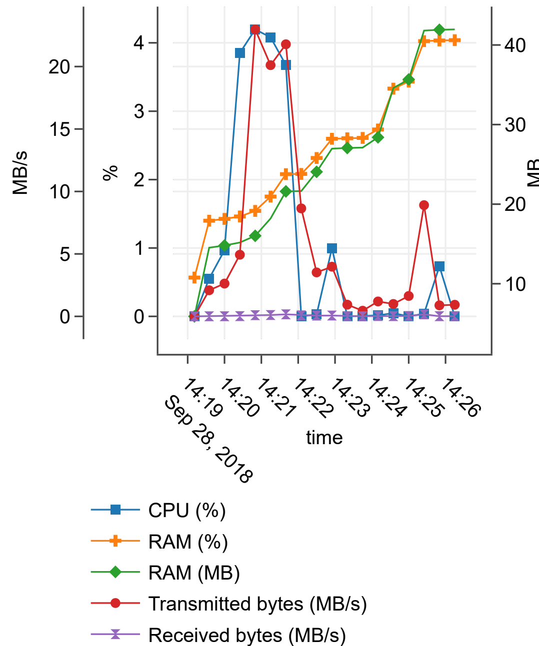


Figura 5.6: Métricas do serviço Catalogue, durante o teste de carga ao acesso do catálogo dos produtos, sem replicação.

Teste com replicação na *cloud* Este teste, representado na Figura 5.7, mostra o comportamento da aplicação quando estão configuradas regras para a replicação dos serviços Frontend e Catalogue.

As regras configuradas para os serviços Frontend e Catalogue foram decididas com

base nos valores das métricas obtidas no teste 5.3.1.1. Para a replicação do serviço Frontend, bem como para o serviço Catalogue, foi configurada a mesma regra 5.1.

$$TransmittedBytes/s \geq 4MB/s \Rightarrow Replicate \quad (5.1)$$

A execução deste teste levou a que o componente de gestão de micro-serviços replicasse 4 vezes o serviço Catalogue, ficando no final com 5 réplicas, e replicasse 4 vezes o serviço Frontend, ficando também com 5 réplicas. As métricas dos serviços Frontend estão apresentadas nas Figuras 5.8, 5.9, 5.10, 5.11 e 5.12. Em relação aos serviços Catalogue, os resultados são apresentados nas Figuras 5.13, 5.14, 5.15 e 5.16.

Uma nota relevante é que durante o período entre a decisão de replicação da 5ª réplica do serviço Catalogue e até esta ter iniciado, o teste finalizou, sendo que esta não recebeu pedidos e portanto não foi objeto de análise.

Os valores do teste, visíveis na Tabela 5.2, apresentam valores bastante inferiores ao teste sem replicação. Na Tabela 5.3, é possível ver as melhorias obtidas através da replicação dos serviços. Verifica-se um decréscimo tanto na obtenção dos dados do catálogo, como na obtenção das imagens. A média da obtenção dos dados do catálogo diminui 6,6 segundos e na obtenção das imagens diminui 20,1 segundos. Em relação ao desvio padrão também houve melhorias, diminuindo em 9,7 segundos na obtenção do catálogo e 25,2 segundo no caso das imagens, ou seja, houve uma menor dispersão em ambos os tempos. Por fim o máximo dos tempos de resposta também sofreram uma diminuição. No caso dos dados do catálogo houve uma redução de 29,5 segundos e no caso das imagens de 59,6 segundos.

Em relação às métricas dos serviços, estas apresentam comportamentos expectáveis, tanto em relação aos micro-serviços Frontend, como Catalogue. É possível observar que existiu uma distribuição da carga pelas respetivas réplicas, indicando que todas receberam pedidos, sendo verificável pelo valor de transferência de bytes por segundo.

Tabela 5.2: Valores do acesso ao catálogo, com replicação na *cloud*.

Tipo de valor	Média (s)	Desvio padrão (s)	Máximo (s)
Tempo de resposta	2,6	4,1	27,9
Duração do grupo	11,6	13,3	60,4

Tabela 5.3: Diferença dos valores do acesso entre os testes do catálogo com replicação na *cloud* e sem replicação.

Tipo de valor	Média	Desvio padrão	Máximo
Tempo de resposta	-6,6 s (-254%)	-9,7 s (-237%)	-29,5 s (-106%)
Duração do grupo	-20,1 s (-173%)	-25,2 s (-189%)	-59,6 s (-99%)

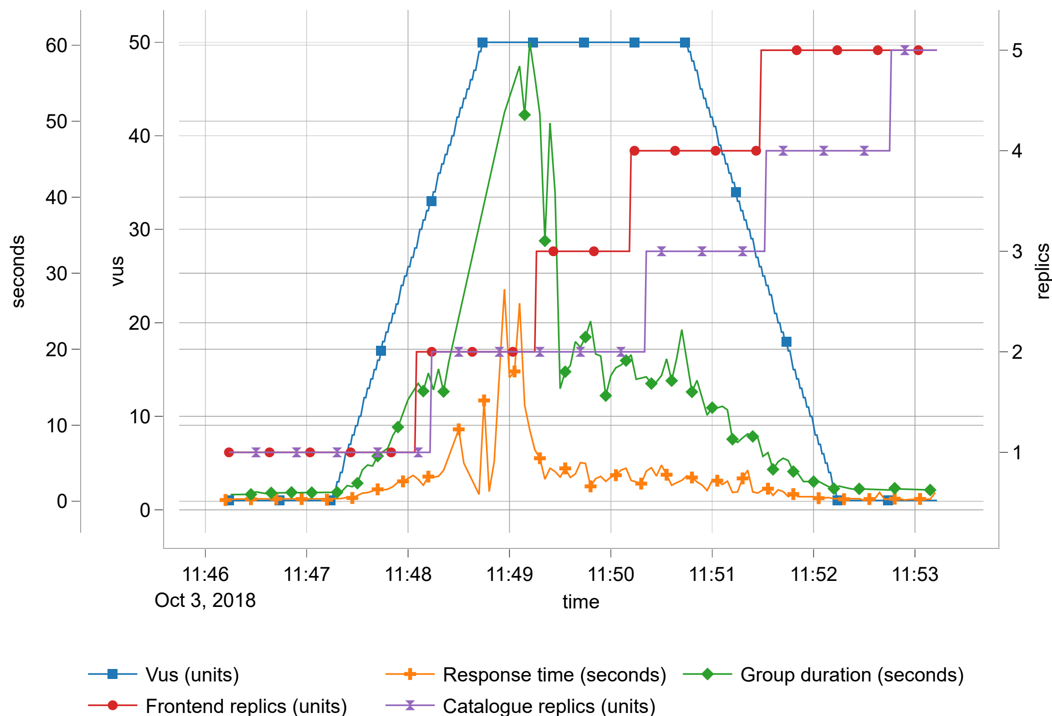


Figura 5.7: Teste de carga ao acesso do catálogo dos produtos, com replicação na *cloud*.

Foram também analisados os custos inerentes à replicação dos micro-serviços, sendo estes relativos ao tamanho transferido e ocupação em disco pelo micro-serviço, e tempos de transferência e de inicialização, apresentando-se estes valores na Tabela 5.4.

Tabela 5.4: Custos da replicação dos micro-serviços Frontend, Catalogue e Catalogue DB.

Micro-serviço	Tamanho transferido	Tamanho em disco	Tempo a transferir	Tempo a iniciar	Tempo total
Frontend	155 MB	260 MB	7 s	15 s	22 s
Catalogue	66 MB	103 MB	3 s	7 s	10 s
Catalogue DB	124 MB	371 MB	11 s	16 s	27 s

Em relação ao serviço Frontend, existem dois cenários em termos de tempo de replicação:

1. Se a imagem do micro-serviço não se encontrar em disco, o tempo total para a replicação é de 22 segundos;

2. Se a imagem do micro-serviço já estiver em disco, o tempo de replicação corresponde ao tempo necessário para iniciar o serviço, ou seja, 15 segundos.

No caso do serviço Catalogue, existe um conjunto maior de cenários, pois este depende de um serviço de base de dados:

1. Caso a imagem do serviço de base de dados não se encontre em disco:

   a) Se a imagem do micro-serviço Catalogue não se encontrar em disco, o tempo total de replicação engloba o tempo de transferência e de inicialização do serviço de base de dados e do micro-serviço, num total de 37 segundos;
   b) Se a imagem do micro-serviço Catalogue já estiver em disco, o tempo de replicação engloba o tempo de transferência e de inicialização do serviço de base de dados e apenas da inicialização do micro-serviço, totalizando 34 segundos.

2. Caso a imagem do serviço de base de dados já esteja em disco:

   a) Se a imagem do micro-serviço Catalogue não se encontrar em disco, o tempo total de replicação engloba o tempo de inicializar o serviço de base de dados e o tempo de transferir e inicializar o micro-serviço, num total de 26 segundos;
   b) Se a imagem do micro-serviço Catalogue já estiver em disco, o tempo total de replicação apenas engloba o tempo de inicializar ambos o serviço de base de dados e o micro-serviço, dando um total de 23 segundos.

3. Caso já exista em execução o serviço de base de dados no nó:

   a) Se a imagem do micro-serviço Catalogue não se encontrar em disco, o tempo total de replicação engloba o tempo de transferência e inicialização do micro-serviço, num total de 10 segundos;
   b) Se a imagem do micro-serviço Catalogue já estiver em disco, o tempo total de replicação apenas engloba o tempo de inicializar o micro-serviço, 7 segundos.

Os tempos obtidos parecem ser adequados para a reação do sistema a variações de carga, sendo no entanto preferível a transferência prévia dos serviços para uma menor duração do tempo total de inicialização dos mesmos.

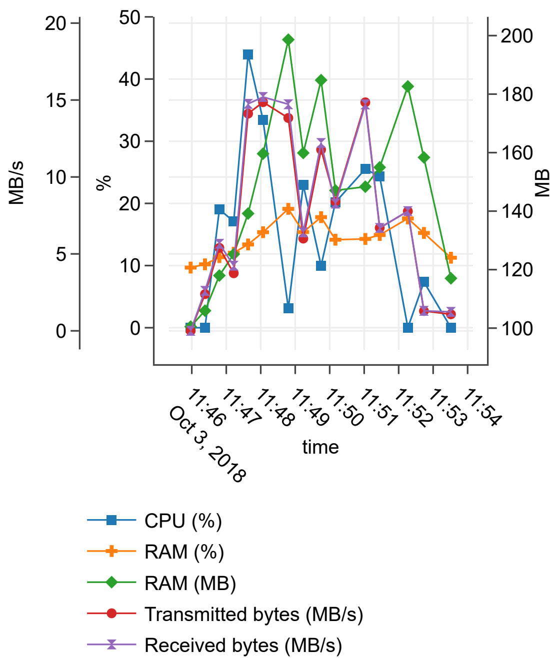


Figura 5.8: Métricas da 1ª réplica do serviço Frontend, durante o teste de carga ao acesso do catálogo dos produtos, com replicação na *cloud*.

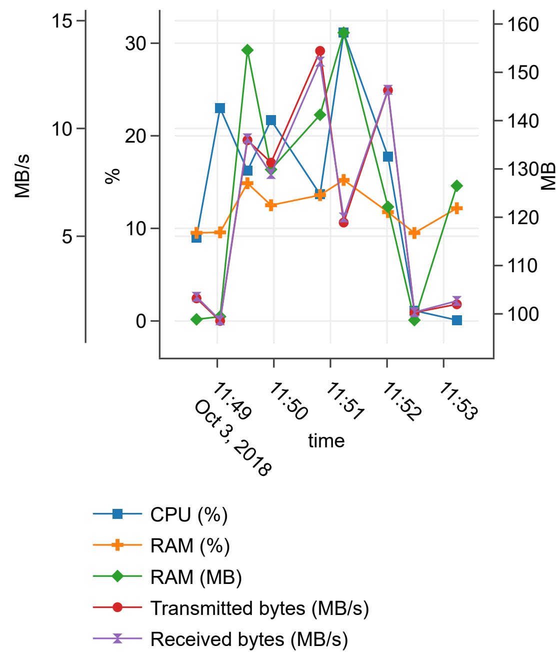


Figura 5.9: Métricas da 2ª réplica do serviço Frontend, durante o teste de carga ao acesso do catálogo dos produtos, com replicação na *cloud*.

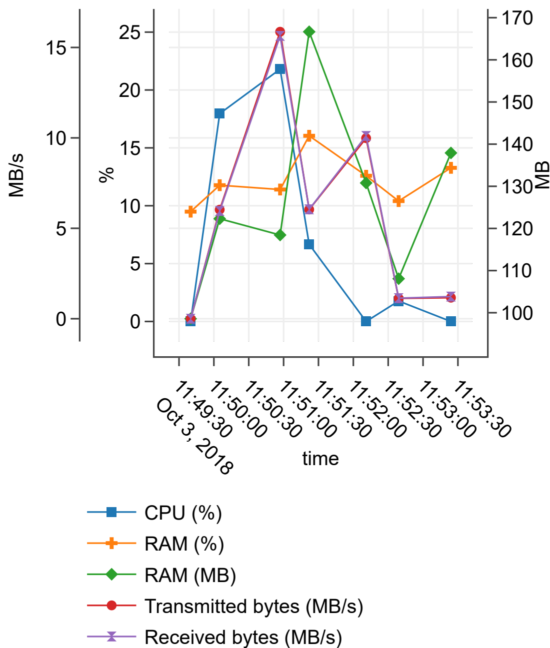


Figura 5.10: Métricas da 3ª réplica do serviço Frontend, durante o teste de carga ao acesso do catálogo dos produtos, com replicação na *cloud*.

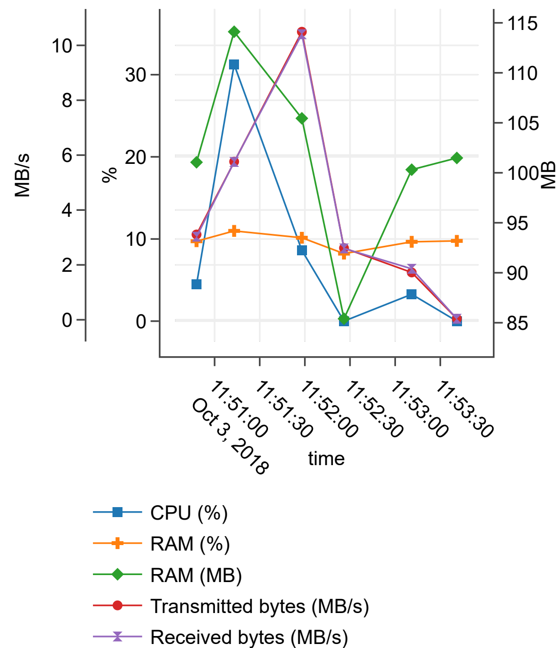


Figura 5.11: Métricas da 4ª réplica do serviço Frontend, durante o teste de carga ao acesso do catálogo dos produtos, com replicação na *cloud*.

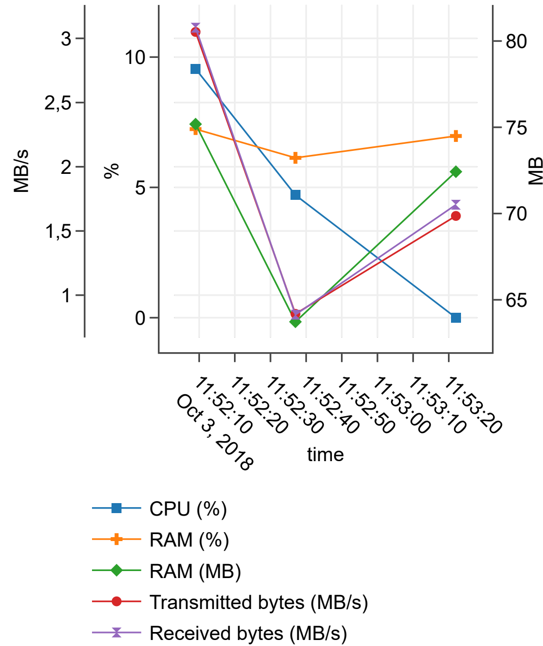


Figura 5.12: Métricas da 5ª réplica do serviço Frontend, durante o teste de carga ao acesso do catálogo dos produtos, com replicação na *cloud*.

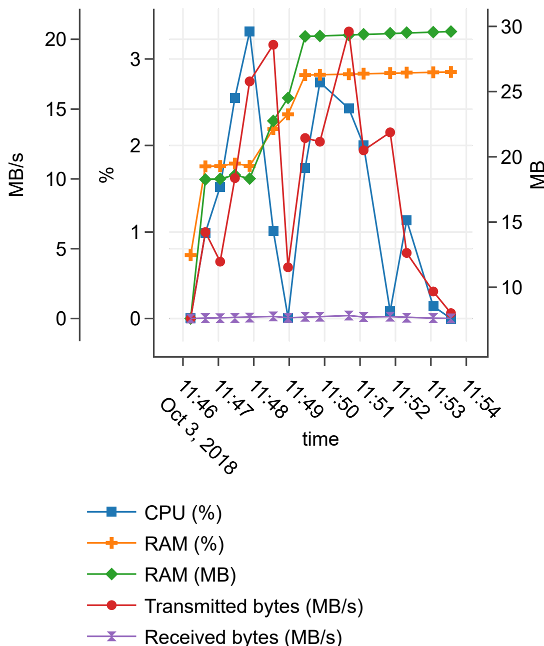


Figura 5.13: Métricas da 1ª réplica do serviço Catalogue, durante o teste de carga ao acesso do catálogo dos produtos, com replicação na *cloud*.

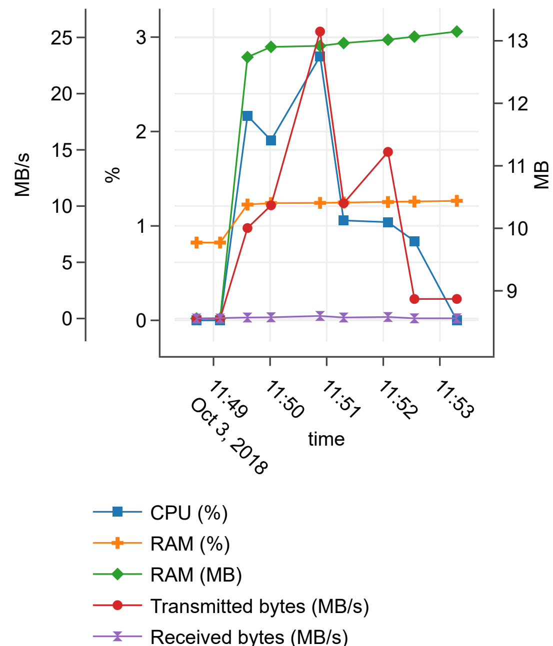


Figura 5.14: Métricas da 2ª réplica do serviço Catalogue, durante o teste de carga ao acesso do catálogo dos produtos, com replicação na *cloud*.

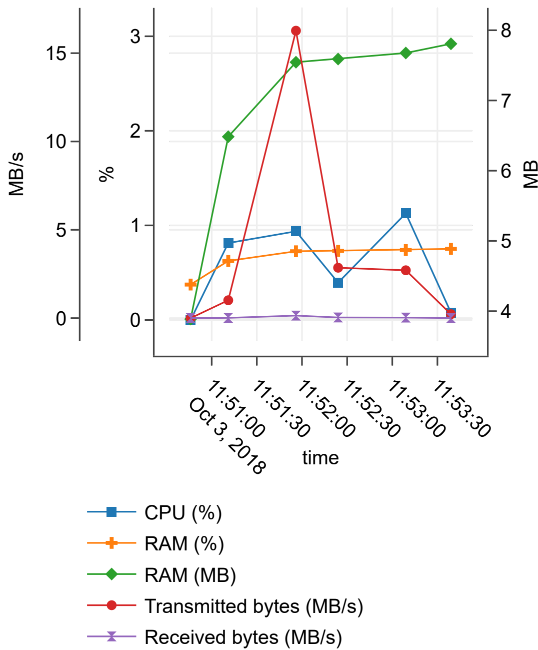


Figura 5.15: Métricas da 3ª réplica do serviço Catalogue, durante o teste de carga ao acesso do catálogo dos produtos, com replicação na *cloud*.

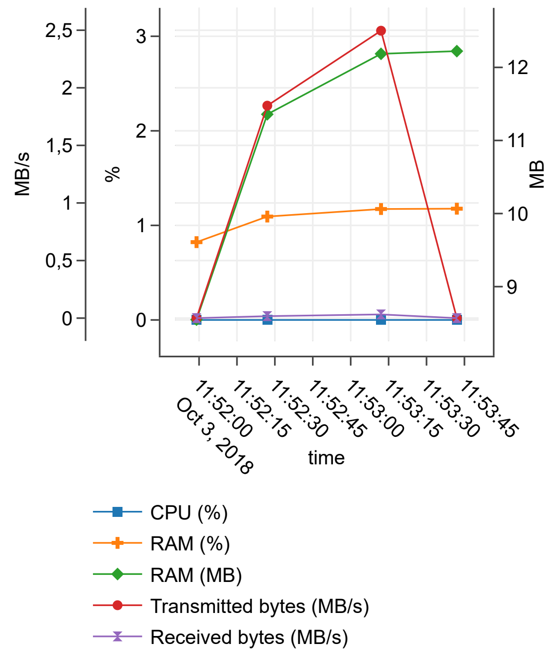


Figura 5.16: Métricas da 4ª réplica do serviço Catalogue, durante o teste de carga ao acesso do catálogo dos produtos, com replicação na *cloud*.

Teste com replicação na *cloud* e na *edge* Este teste está representado nas Figuras 5.17, 5.18 e 5.19, mostrando os valores médios, dos clientes de Portland e dos clientes de Londres, respetivamente. Pode observar-se o comportamento da aplicação quando estão disponíveis tanto nós na *cloud*, como na *edge*, de forma a mostrar os benefícios de uma infraestrutura que contemple nós mais próximos dos utilizadores. O teste segue a mesma lógica dos anteriores, sendo que para além dos nós da *cloud*, onde o seu número foi diminuído para 5 nós iniciais, estão também disponíveis 5 nós localizados em Londres para simular nós na *edge*. Neste caso, foram incluídos clientes de dois locais distintos, Portland e Londres, com taxa de acessos idêntica, de forma a perceber o comportamento do componente de gestão de micro-serviços quando existem clientes de múltiplos locais, concretamente no que diz respeito ao local onde são colocadas as novas réplicas.

Para além da configuração de regras para a replicação dos micro-serviços, foram também configuradas regras para a sua remoção, de forma a mostrar o comportamento do processo de reconfiguração neste caso. Foram configuradas as mesmas regras para o serviço Frontend como para o serviço Catalogue, onde a regra 5.2 diz respeito à replicação dos serviços e a regra 5.3 é referente à sua remoção.

$$TransmittedBytes/s \geq 2.5MB/s \Rightarrow Replicate \quad (5.2)$$

$$TransmittedBytes/s < 0.5MB/s \Rightarrow Stop \quad (5.3)$$

Neste teste, o serviço Catalogue foi replicado 5 vezes, ficando no final com 6 réplicas, e o serviço Frontend 4 vezes, ficando com 5 réplicas. As métricas dos serviços Frontend estão apresentados nas Figuras 5.21, 5.22, 5.23, 5.24 e 5.25. Em relação aos serviços Catalogue, estas são visíveis nas Figuras 5.26, 5.27, 5.28, 5.29, 5.30 e 5.31. As instâncias dos serviços foram replicadas e depois removidas de acordo a Figura 5.20.

Os valores dos serviços mostram que existiu uma distribuição dos pedidos por todas as réplicas de forma semelhante, com exceção da 6ª réplica do serviço Catalogue e da 5ª réplica do serviço Frontend, que apresentaram valores mais baixos. A diferença foi, em particular na transmissão de bytes por segundo, isto devido ao facto do número de pedidos ser menor na altura em que estas ficaram disponíveis.

As Tabelas 5.5, 5.6 e 5.7 mostram os valores do acesso ao catálogo, de forma média, por parte dos clientes de Portland e pelos clientes de Londres, respetivamente. Estes valores quando comparados com o teste com replicação apenas na *cloud*, conforme apresentado na Tabela 5.8, na perspetiva dos clientes de Londres, apresentam valores com grandes melhorias, onde a média do tempo de resposta desceu 2,1 segundos, o que representa uma taxa de variação de -420% e a média da duração do grupo uma diminuição de 7,1 segundos, representado uma taxa de variação de -158%.

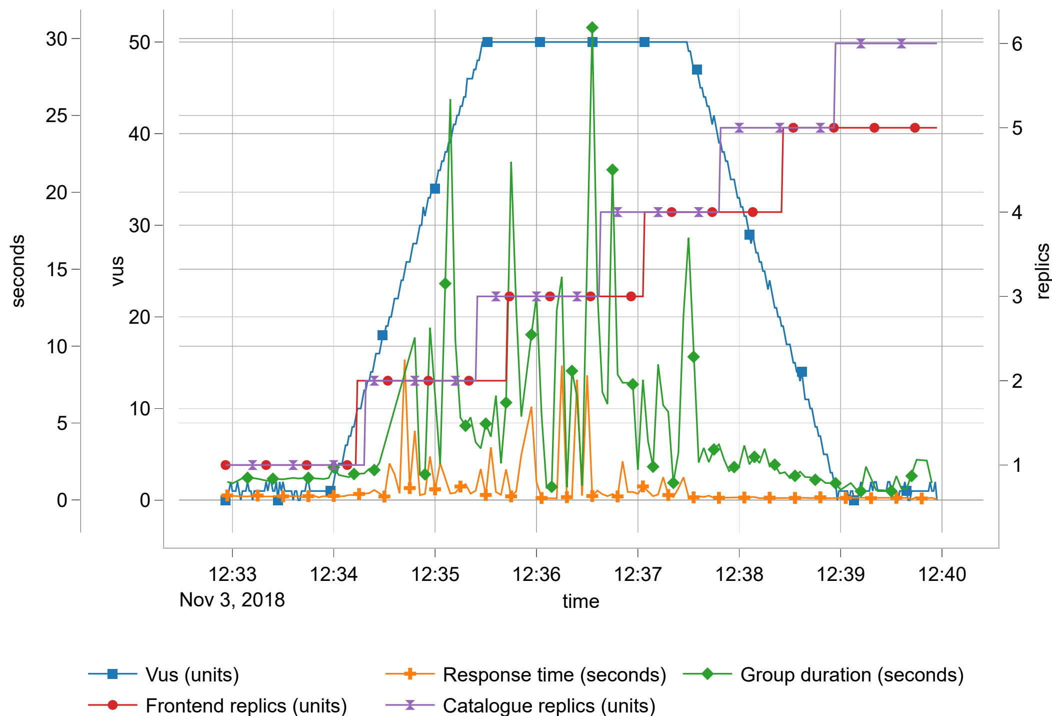


Figura 5.17: Teste de carga ao acesso do catálogo dos produtos, com replicação na *cloud* e na *edge* (média).

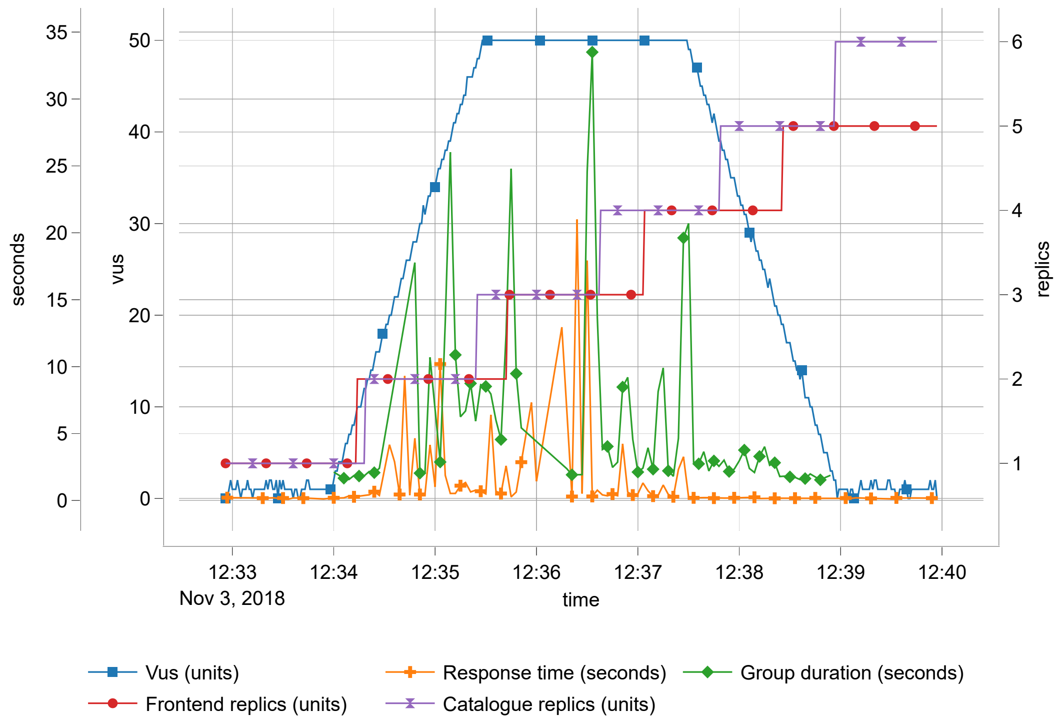


Figura 5.18: Teste de carga ao acesso do catálogo dos produtos, com replicação na *cloud* e na *edge* (Portland).

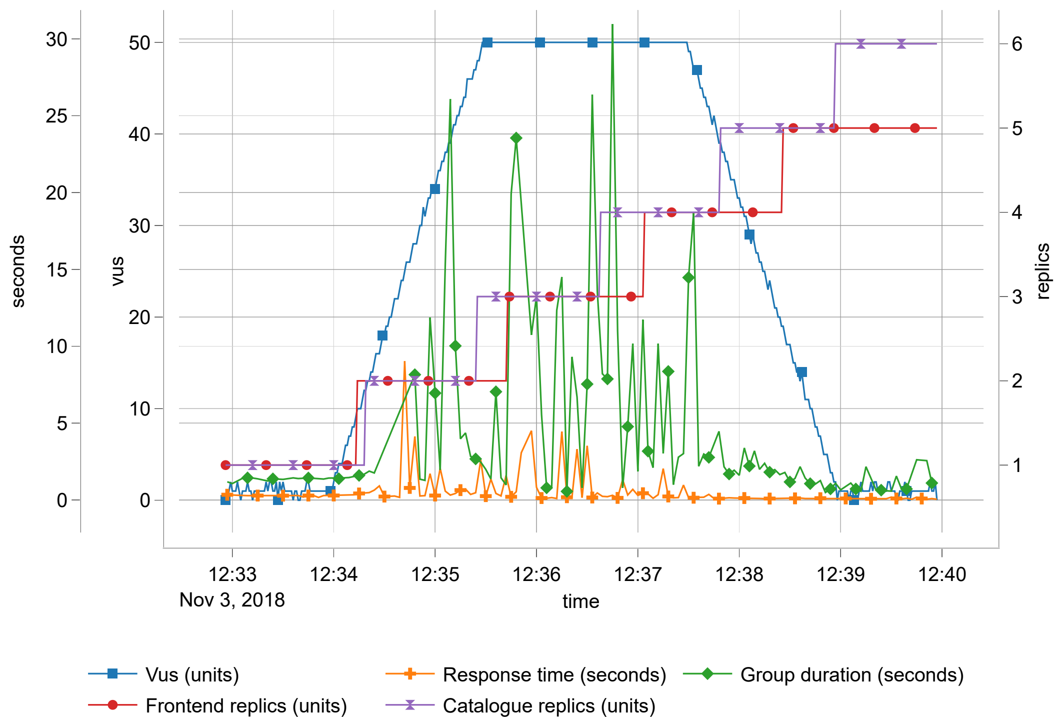


Figura 5.19: Teste de carga ao acesso do catálogo dos produtos, com replicação na *cloud* e na *edge* (Londres).

Tabela 5.5: Valores do acesso ao catálogo, com replicação na *cloud* e na *edge* (média).

Tipo de valor	Média (s)	Desvio padrão (s)	Máximo (s)
Tempo de resposta	0,7	1,6	9,1
Duração do grupo	4,6	5,4	30,7

Tabela 5.6: Valores do acesso ao catálogo, com replicação na *cloud* e na *edge* (Portland).

Tipo de valor	Média (s)	Desvio padrão (s)	Máximo (s)
Tempo de resposta	1,3	3,1	21,0
Duração do grupo	5,8	6,4	33,5

Tabela 5.7: Valores do acesso ao catálogo, com replicação na *cloud* e na *edge* (Londres).

Tipo de valor	Média (s)	Desvio padrão (s)	Máximo (s)
Tempo de resposta	0,5	1,1	9,0
Duração do grupo	4,5	5,9	31,0

Tabela 5.8: Diferença dos valores do acesso entre os testes do catálogo com replicação apenas na *cloud* e com replicação na *cloud* e na *edge* (Londres).

Tipo de valor	Média	Desvio padrão	Máximo
Tempo de resposta	-2,1 s (-420%)	-3,0 s (-273%)	-18,9 s (-210%)
Duração do grupo	-7,1 s (-158%)	-7,4 s (-125%)	-29,4 s (-95%)

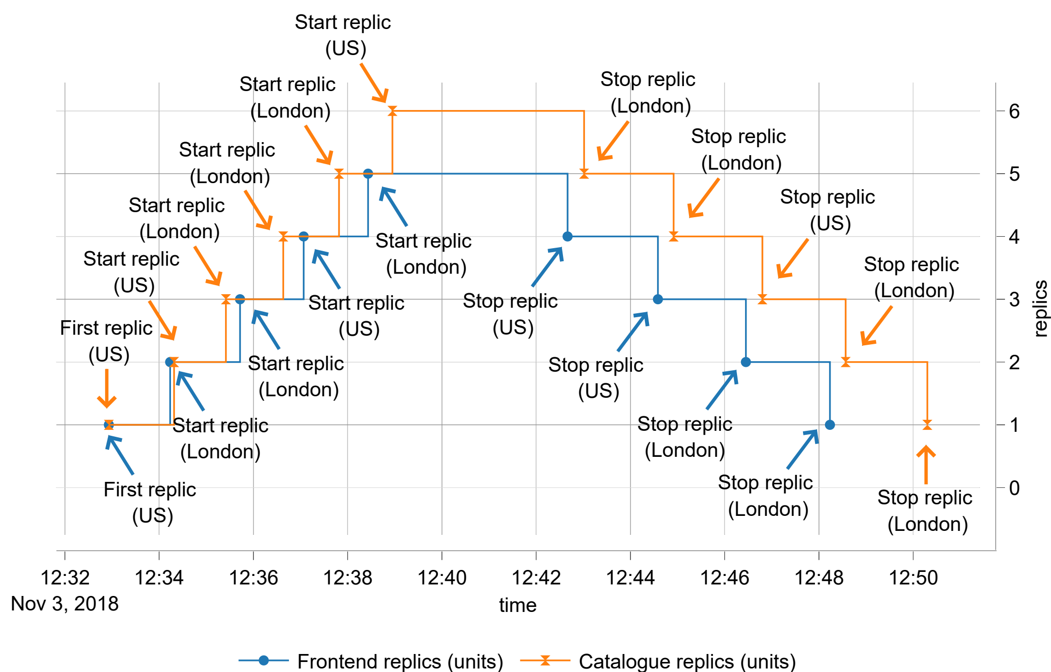


Figura 5.20: Variação das réplicas no teste de carga ao acesso do catálogo dos produtos, com replicação na *cloud* e na *edge*.

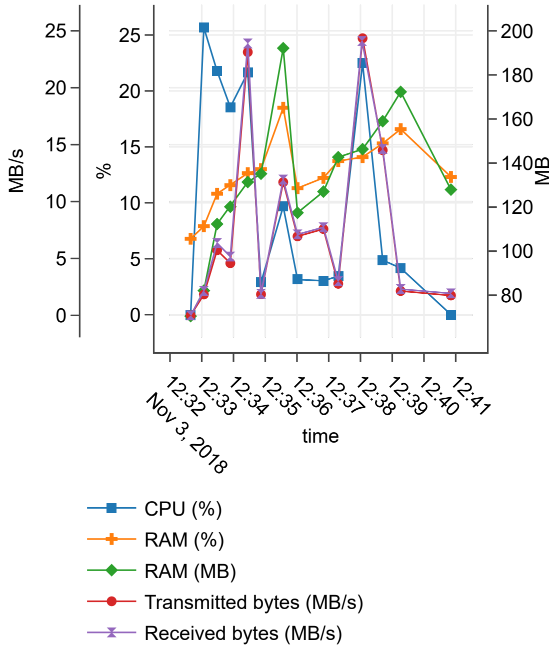


Figura 5.21: Métricas da 1ª réplica do serviço Frontend, durante o teste de carga ao acesso do catálogo dos produtos, com replicação na *cloud* e na *edge*.

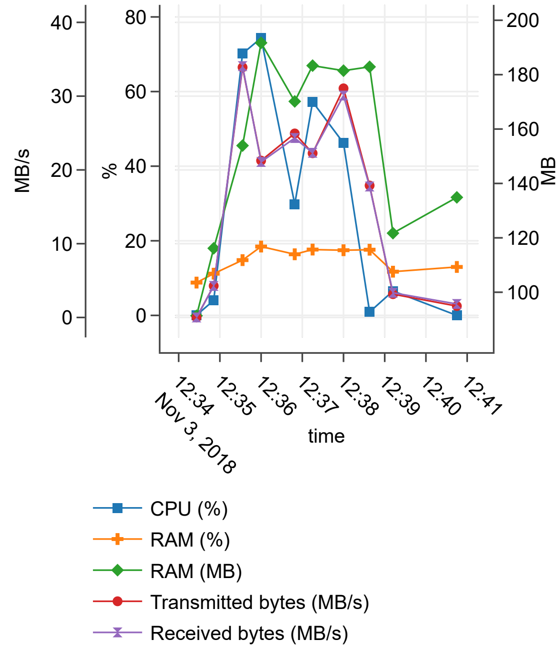


Figura 5.22: Métricas da 2ª réplica do serviço Frontend, durante o teste de carga ao acesso do catálogo dos produtos, com replicação na *cloud* e na *edge*.

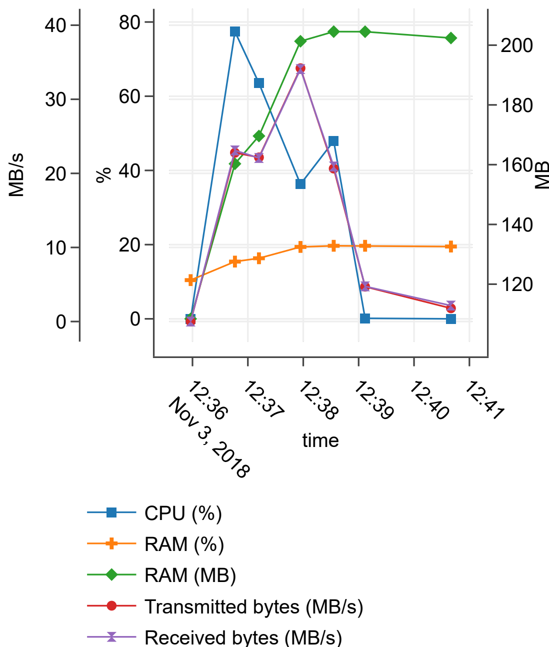


Figura 5.23: Métricas da 3ª réplica do serviço Frontend, durante o teste de carga ao acesso do catálogo dos produtos, com replicação na *cloud* e na *edge*.

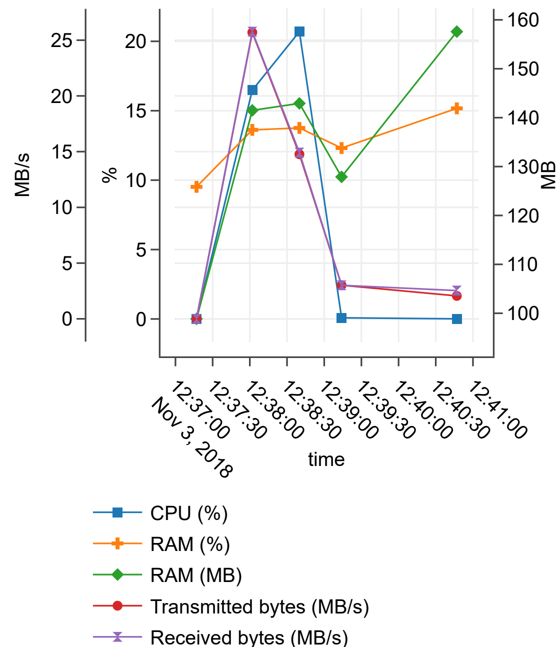


Figura 5.24: Métricas da 4ª réplica do serviço Frontend, durante o teste de carga ao acesso do catálogo dos produtos, com replicação na *cloud* e na *edge*.

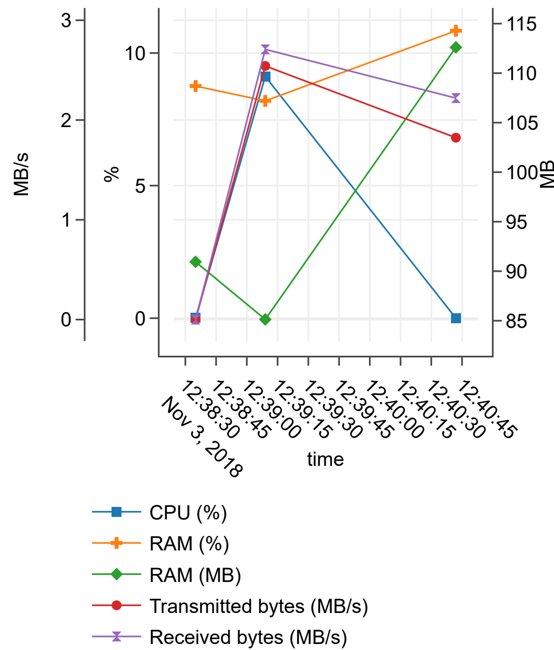


Figura 5.25: Métricas da 5ª réplica do serviço Frontend, durante o teste de carga ao acesso do catálogo dos produtos, com replicação na *cloud* e na *edge*.

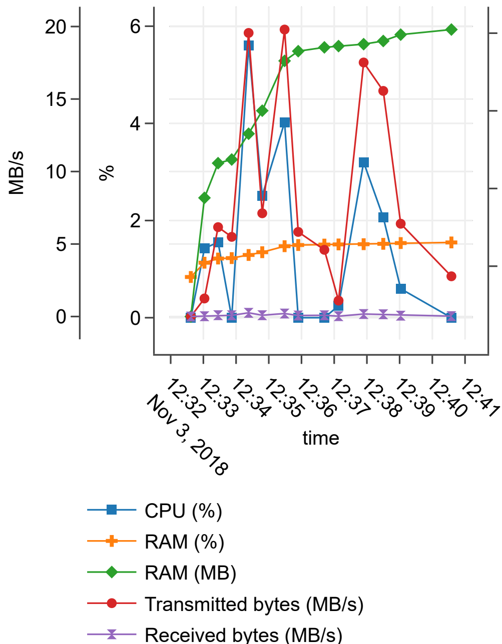


Figura 5.26: Métricas da 1ª réplica do serviço Catalogue, durante o teste de carga ao acesso do catálogo dos produtos, com replicação na *cloud* e na *edge*.

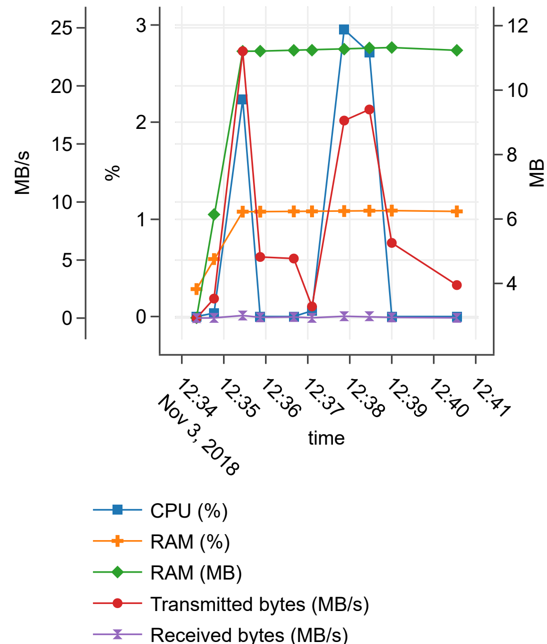


Figura 5.27: Métricas da 2ª réplica do serviço Catalogue, durante o teste de carga ao acesso do catálogo dos produtos, com replicação na *cloud* e na *edge*.

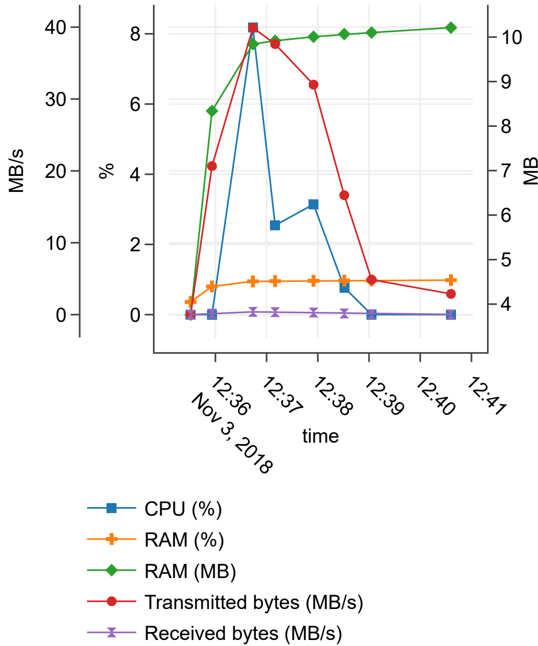


Figura 5.28: Métricas da 3ª réplica do serviço Catalogue, durante o teste de carga ao acesso do catálogo dos produtos, com replicação na *cloud* e na *edge*.

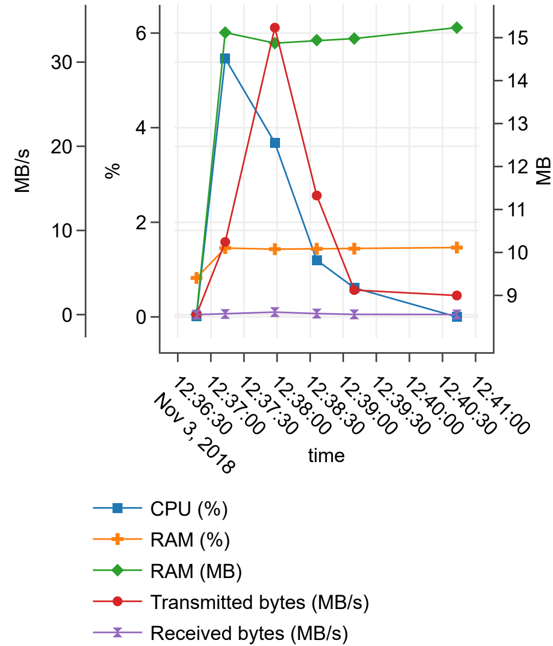


Figura 5.29: Métricas da 4ª réplica do serviço Catalogue, durante o teste de carga ao acesso do catálogo dos produtos, com replicação na *cloud* e na *edge*.

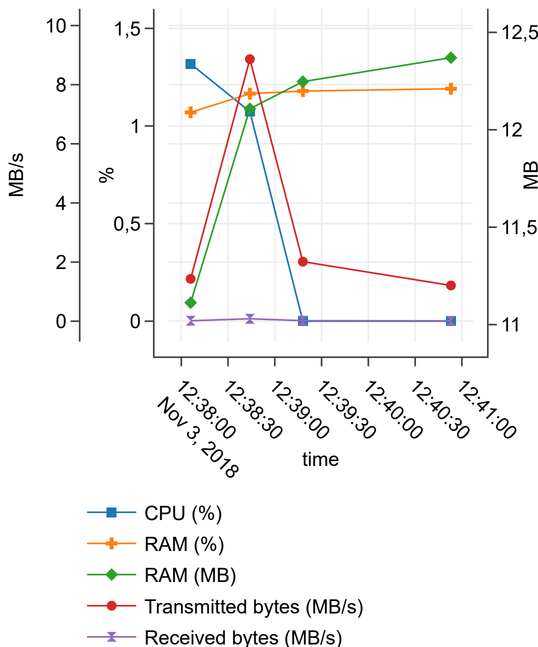


Figura 5.30: Métricas da 5ª réplica do serviço Catalogue, durante o teste de carga ao acesso do catálogo dos produtos, com replicação na *cloud* e na *edge*.

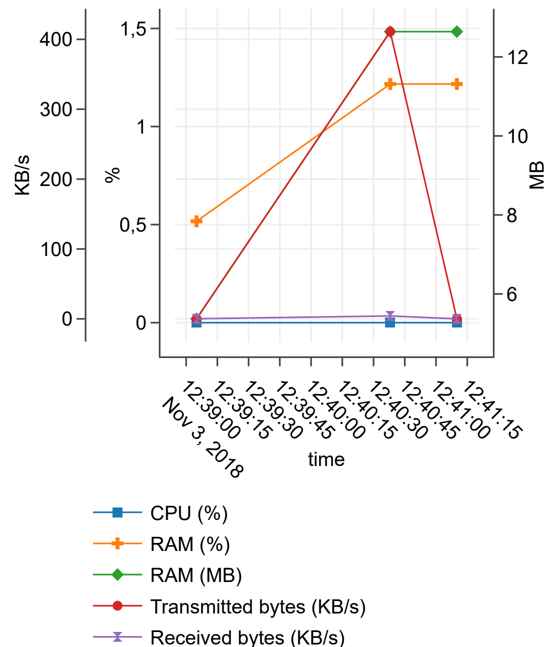


Figura 5.31: Métricas da 6ª réplica do serviço Catalogue, durante o teste de carga ao acesso do catálogo dos produtos, com replicação na *cloud* e na *edge*.

5.3.1.2 Login e registo de utilizadores

Para efetuar logins e registo de utilizadores são utilizados os serviços Frontend e User. São simulados tanto logins, como registos de utilizadores, com uma cota aproximada 50% cada um. O comportamento segue a linha do teste apresentado anteriormente; o utilizador acede ao micro-serviço Frontend e este encaminha o pedido para o micro-serviço User. Quando o serviço User efetua a operação, envia a resposta para o Frontend, que por sua vez a disponibiliza ao utilizador.

Nos testes seguintes, o valor do tempo de resposta (*response time*) diz respeito ao tempo que leva a ser feito o login ou o registo de um utilizador. Os testes têm a duração de 7 minutos, onde foram introduzidas alterações no número de utilizadores, aumentando gradualmente de 1 até 100 utilizadores, reduzindo-se depois de forma gradual até 1 utilizador. Cada utilizador efetua pedidos, entre 0 a 1 segundos (aleatoriamente), após a receção da resposta.

Teste sem replicação na *cloud* Este teste, representado na Figura 5.32, serve para mostrar o comportamento da aplicação quando não existe replicação dos serviços.

Nas Figuras 5.5 e 5.6, é possível observar os valores das métricas dos serviços Frontend e User, durante a execução deste teste.

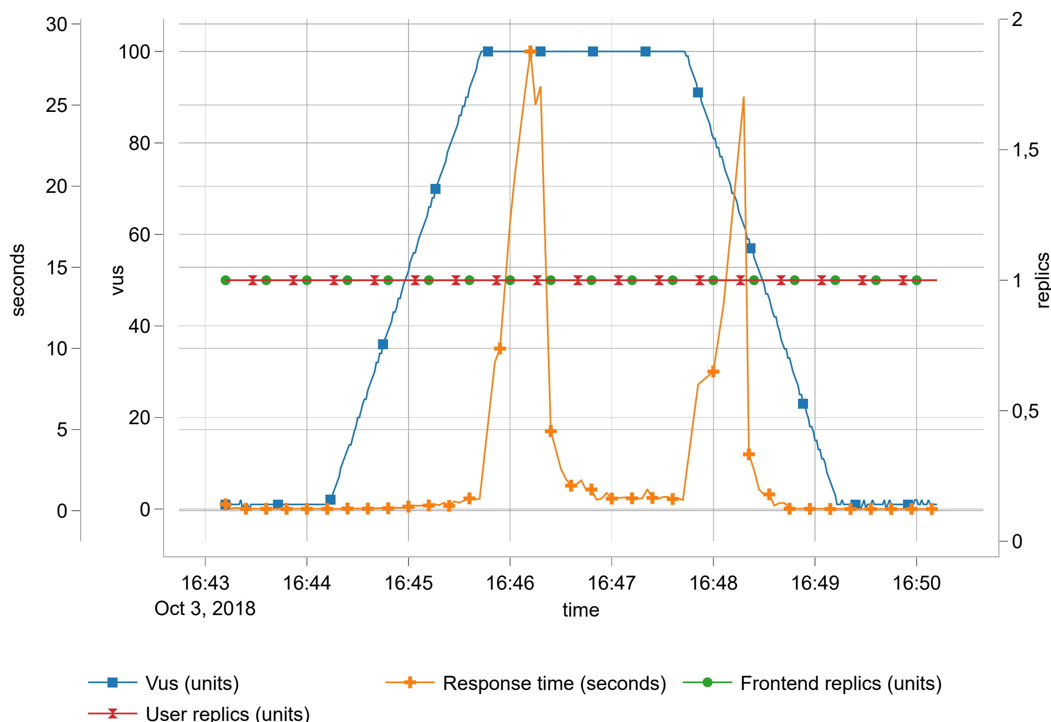


Figura 5.32: Teste de carga ao efetuar logins e registos de utilizadores, sem replicação.

Este tipo de operações, quando comparadas ao teste anterior (secção 5.3.1.1), são muito mais rápidas em termos de tempo de resposta, tendo por isso sido aumentado o número de utilizadores máximo que realizam o teste. É possível verificar que existem

dois picos bastantes elevados nos tempos de resposta, um na ordem dos 28 segundos, e outros perto dos 25 segundos. Este aumento acontece devido à acumulação de pedidos nos micro-serviços, levando a um maior tempo de processamento dos pedidos. Os valores do teste encontram-se na Tabela 5.9.

Tabela 5.9: Valores dos logins e registos de utilizadores, sem replicação.

Tipo de valor	Média (s)	Desvio padrão (s)	Máximo (s)
Tempo de resposta	2,4	6,1	28,3

Em relação aos valores das métricas dos micro-serviços, é possível observar que ambos os serviços de Frontend, na Figura 5.33, e User, visível na Figura 5.34, apresentam comportamentos algo semelhantes, apesar de terem valores distintos. Assistiu-se a um aumento da taxa de utilização de CPU, não de uma forma constante, isto é, com picos de utilização, tendo o serviço Frontend atingido perto dos 90%, e o serviço User cerca dos 6%. Uma outra métrica que registou um aumento nos dois serviços, foi a quantidade de RAM utilizada, onde é visível que no serviço Frontend existiram certas oscilações, aumentando mas com ligeiros decréscimos em alguns momentos, tendo atingindo no seu pico mais alto, cerca de 205 MB, ao passo que no serviço User este aumento foi constante, mas com valores muito inferiores, onde o seu pico mais alto foi de 13 MB. Em relação à transmissão e receção de bytes por segundo, apresentam valores bastante próximos, pois tanto o tamanho (bytes) dos pedidos, como a resposta têm valores baixos.

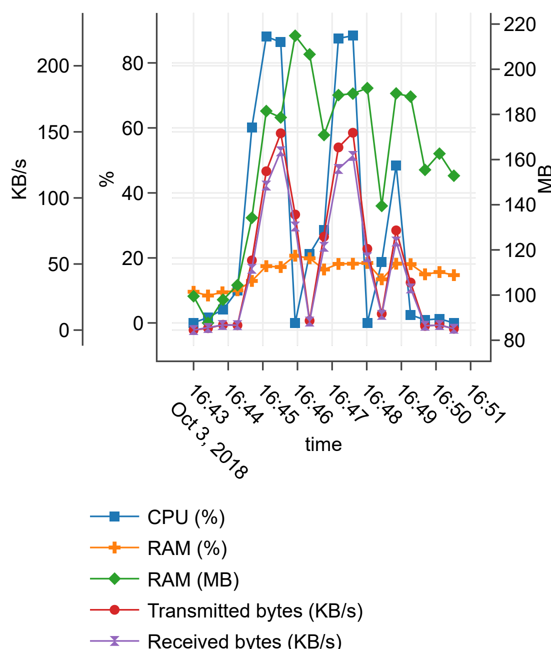


Figura 5.33: Métricas do serviço Frontend, durante o teste de carga ao efetuar logins e registos de utilizadores, sem replicação.

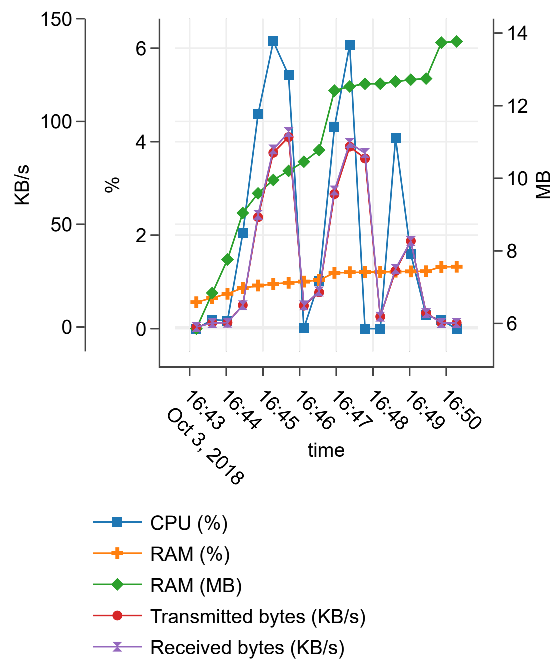


Figura 5.34: Métricas do serviço User, durante o teste de carga ao efetuar logins e registos de utilizadores, sem replicação.

Teste com replicação na *cloud* Este teste, representado na Figura 5.35, mostra o comportamento da aplicação quando estão configuradas regras para a replicação dos serviços Frontend e User.

As regras configuradas para os serviços Frontend e User foram decididas com base nos valores das métricas obtidas no teste 5.3.1.2. Para a replicação do serviço Frontend foi configurada a regra 5.4, e para o serviço User foi configurada a regra 5.5.

$$\%CPU \geq 55\% \Rightarrow Replicate \tag{5.4}$$

$$\%CPU \geq 3\% \Rightarrow Replicate \tag{5.5}$$

Durante a execução do teste o componente de gestão de micro-serviços replicou duas vezes o serviço User, ficando com 3 réplicas em execução, e replicou duas vezes o serviço Frontend, ficando também com 3 réplicas. As métricas dos serviços Frontend estão dispostas nas Figuras 5.36, 5.37 e 5.38. Em relação aos serviços User, estas são visíveis nas Figuras 5.39, 5.40 e 5.41.

Os valores do teste, na Tabela 5.10, apresentam uma redução significativa quando comparado com o teste sem replicação, podendo observar-se essa redução na Tabela 5.11. A média do tempo de resposta diminui 1,3 segundos, tendo o desvio padrão sofrido uma redução de 2,9 segundos, o que revela uma discrepância menor nos valores dos tempos de resposta. No valor máximo observado houve também uma melhoria, diminuindo 5,3 segundos.

O comportamento dos serviços foi de acordo com o expectável, havendo uma distribuição da carga pelas réplicas, verificável pelos valores dos recursos apresentados, contundo as terceiras réplicas de ambos os serviços apresentaram valores que indicam um menor volume de acessos.

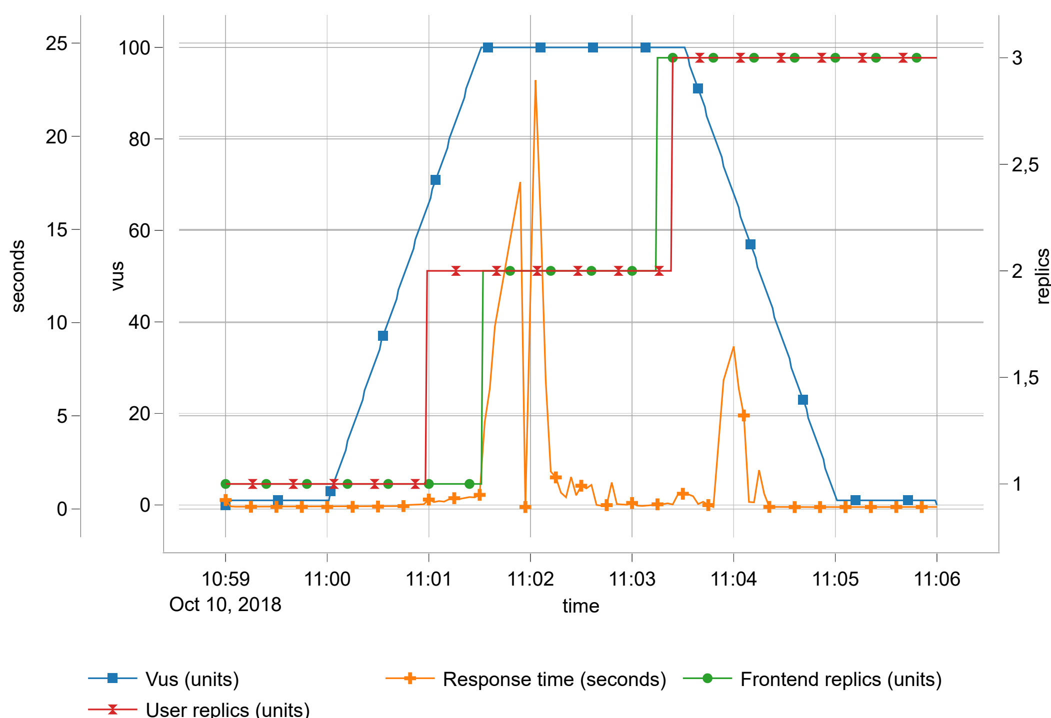


Figura 5.35: Teste de carga ao efetuar logins e registos de utilizadores, com replicação na *cloud*.

Tabela 5.10: Valores dos logins e registos de utilizadores, com replicação na *cloud*.

Tipo de valor	Média (s)	Desvio padrão (s)	Máximo (s)
Tempo de resposta	1,1	3,2	23,0

Tabela 5.11: Diferença dos valores do acesso entre os testes dos logins e registos de utilizadores com replicação na *cloud* e sem replicação.

Tipo de valor	Média	Desvio padrão	Máximo
Tempo de resposta	-1,3 s (-118%)	-2,9 s (-91%)	-5,3 s (-23%)

Os custos relativos à replicação dos micro-serviços, tamanho transferido e ocupação em disco do micro-serviço, e os tempos de transferência e de inicialização, encontram-se visíveis na Tabela 5.12. Em relação ao serviço Frontend, os valores são idênticos, aos apresentados na Tabela 5.4.

Relativamente ao serviço User, existem vários cenários que se podem colocar em termos de tempo de replicação:

Tabela 5.12: Custos da replicação dos micro-serviços User e User DB.

Micro-serviço	Tamanho transferido	Tamanho em disco	Tempo a transferir	Tempo a iniciar	Tempo total
User	23 MB	61 MB	1 s	2 s	3 s
User DB	132 MB	683 MB	21 s	4 s	25 s

1. Caso a imagem do serviço de base de dados não exista em disco:

   a) Se a imagem do micro-serviço User não se encontrar em disco, o tempo total de replicação engloba o tempo de transferência e de inicialização do serviço de base de dados e do micro-serviço, num total de 28 segundos;

   b) Se a imagem do micro-serviço User já estiver em disco, o tempo de replicação engloba o tempo de transferência e de inicialização do serviço de base de dados e apenas da inicialização do micro-serviço, totalizando 27 segundos.

2. Caso a imagem do serviço de base de dados já se encontrar em disco:

   a) Se a imagem do micro-serviço User não se encontrar em disco, o tempo total de replicação engloba o tempo de inicializar o serviço de base de dados e o tempo de transferir e inicializar o micro-serviço, num total de 7 segundos;

   b) Se a imagem do micro-serviço User já estiver em disco, o tempo total de replicação apenas engloba o tempo de inicializar ambos o serviço de base de dados e o micro-serviço, num total de 6 segundos.

3. Caso o serviço de base de dados já esteja em execução no nó:

   a) Se a imagem do micro-serviço User não se encontrar em disco, o tempo total de replicação engloba o tempo de transferência e inicialização do micro-serviço, num total de 3 segundos;

   b) Se a imagem do micro-serviço User já estiver em disco, o tempo total de replicação apenas engloba o tempo de inicializar o micro-serviço, 2 segundos.

Os tempos obtidos neste tipo de teste parecem ser apropriados para a reação do sistema a variações de carga, onde a transferência e inicialização da base de dados tem um maior impacto no tempo total de replicação.

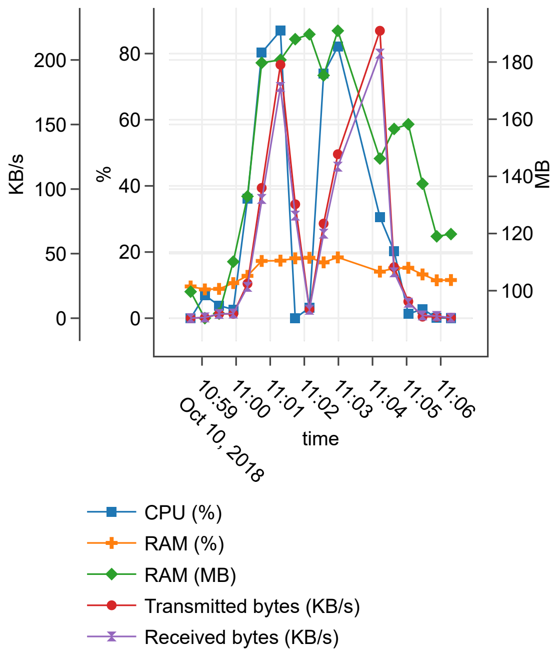


Figura 5.36: Métricas da 1ª réplica do serviço Frontend, durante o teste de carga ao efetuar logins e registos de utilizadores, com replicação na *cloud*.

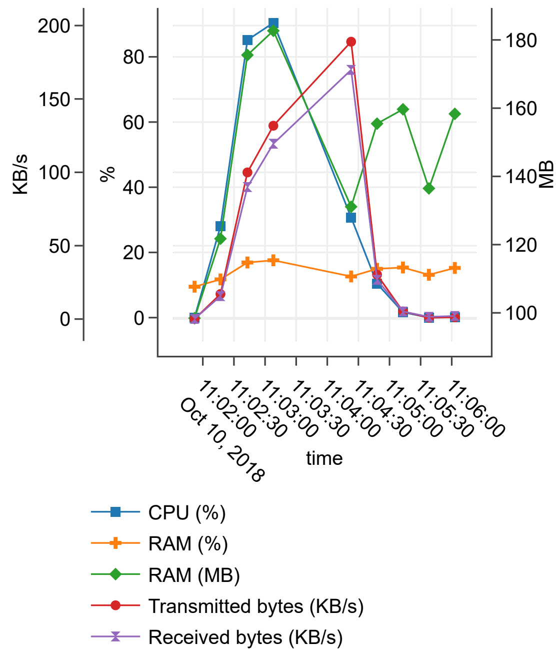


Figura 5.37: Métricas da 2ª réplica do serviço Frontend, durante o teste de carga ao efetuar logins e registos de utilizadores, com replicação na *cloud*.

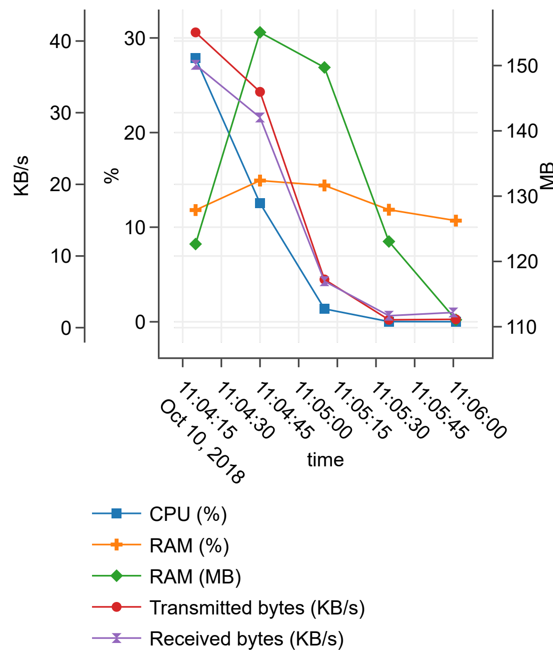


Figura 5.38: Métricas da 3ª réplica do serviço Frontend, durante o teste de carga ao efetuar logins e registos de utilizadores, com replicação na *cloud*.

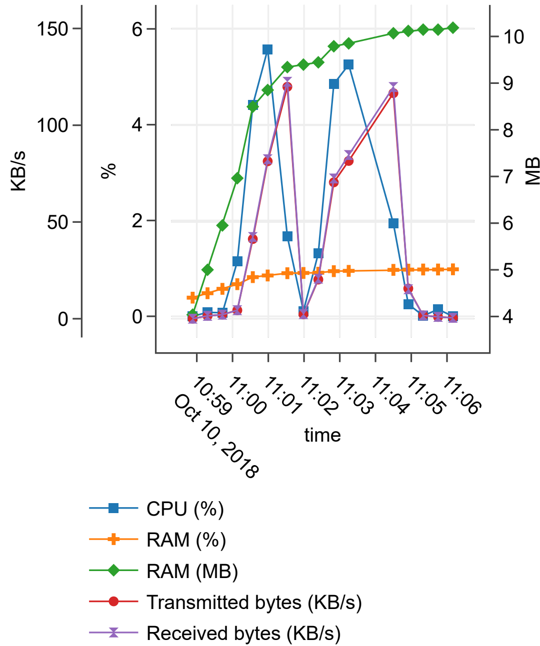


Figura 5.39: Métricas da 1ª réplica do serviço User, durante o teste de carga ao efetuar logins e registos de utilizadores, com replicação na *cloud*.

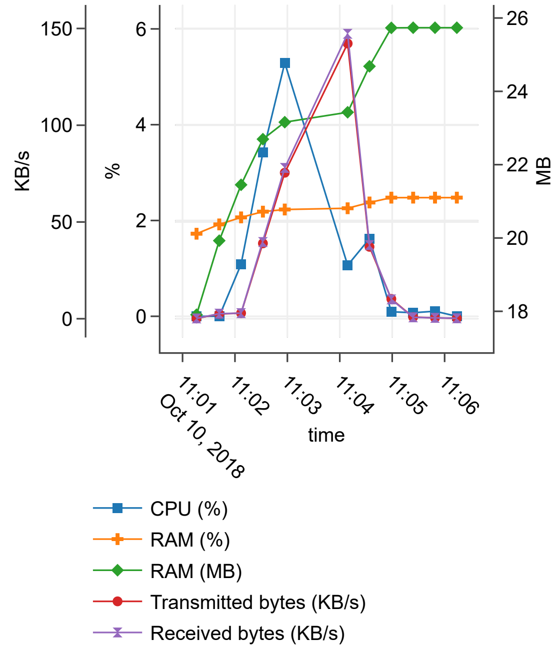


Figura 5.40: Métricas da 2ª réplica do serviço User, durante o teste de carga ao efetuar logins e registos de utilizadores, com replicação na *cloud*.

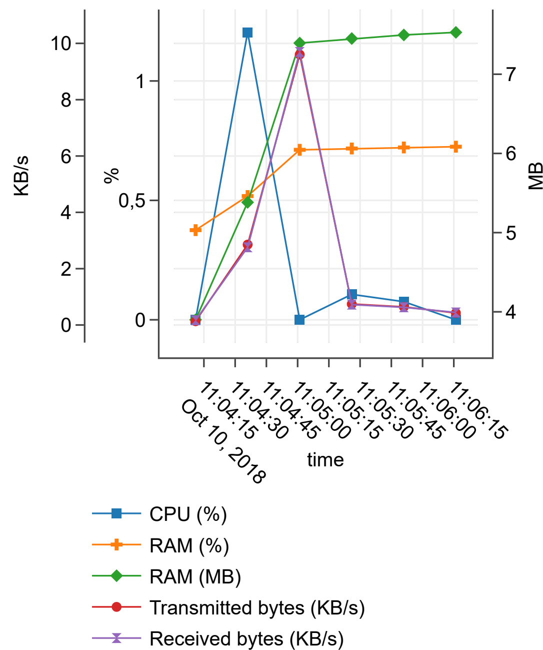


Figura 5.41: Métricas da 3ª réplica do serviço User, durante o teste de carga ao efetuar logins e registos de utilizadores, com replicação na *cloud*.

Teste com replicação na *cloud* e na *edge* Neste teste, representado nas Figuras 5.42, 5.43 e 5.44, mostrando os valores médios, dos clientes de Portland e dos clientes de Londres, respetivamente, mostra o comportamento da aplicação quando estão disponíveis tanto nós na *cloud*, como na *edge*, de forma a mostrar os benefícios de uma infraestrutura que contemple nós mais próximos dos utilizadores. Ao nível dos nós, estão disponíveis inicialmente 5 nós na *cloud* e também 5 nós localizados em Londres para simular nós na *edge*. Foram incluídos clientes de dois locais distintos, Portland e Londres, com taxa de acessos idêntica, para perceber o comportamento do componente de gestão de micro-serviços quando estão presentes clientes de diferentes locais, em concreto, onde são replicados os serviços.

Para além da configuração de regras para a replicação dos micro-serviços, foram também configuradas regras para a sua remoção, de forma a mostrar o comportamento do processo de reconfiguração neste caso.

Para o serviço Frontend foram configuradas as regras 5.6 e 5.7 para a replicação e remoção de réplicas respetivamente. Em relação ao serviço User foi configurada a regra 5.8 para a replicação e a regra 5.9 para a remoção de réplicas.

$$\%CPU \geq 50\% \Rightarrow Replicate \tag{5.6}$$

$$\%CPU < 2\% \Rightarrow Stop \tag{5.7}$$

$$\%CPU \geq 2.5\% \Rightarrow Replicate \tag{5.8}$$

$$\%CPU < 0.5\% \Rightarrow Stop \tag{5.9}$$

Neste teste, ambos os serviços User e Frontend foram replicados duas vezes, ficando com 3 réplicas no final cada um. Os valores das réplicas do serviço Frontend estão dispostos nas Figuras 5.46, 5.47 e 5.48. Em relação ao serviço User, estes são visíveis nas Figuras 5.49, 5.50 e 5.51. As réplicas foram replicadas e depois removidas de acordo a Figura 5.45.

Os valores das réplicas dos serviços mostram que existiu uma distribuição dos pedidos por todas as réplicas de forma semelhante.

As Tabelas 5.13, 5.14 e 5.15 mostram os valores, de forma média, por parte dos clientes de Portland e pelos clientes de Londres, respetivamente. Estes valores quando comparados com o teste com replicação apenas na *cloud*, de acordo o apresentado na Tabela 5.16, na perspetiva dos clientes de Londres, a média do tempo de resposta desceu 0,3 segundos, o que representa uma taxa de variação de -38% .

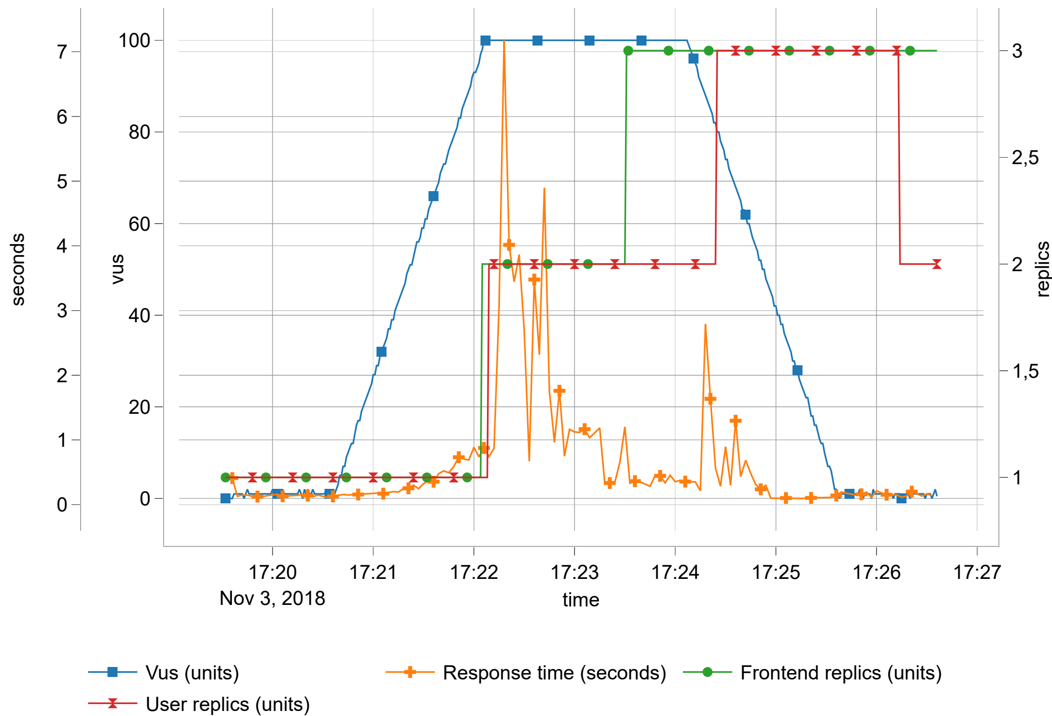


Figura 5.42: Teste de carga ao efetuar logins e registos de utilizadores, com replicação na *cloud* e na *edge* (média).

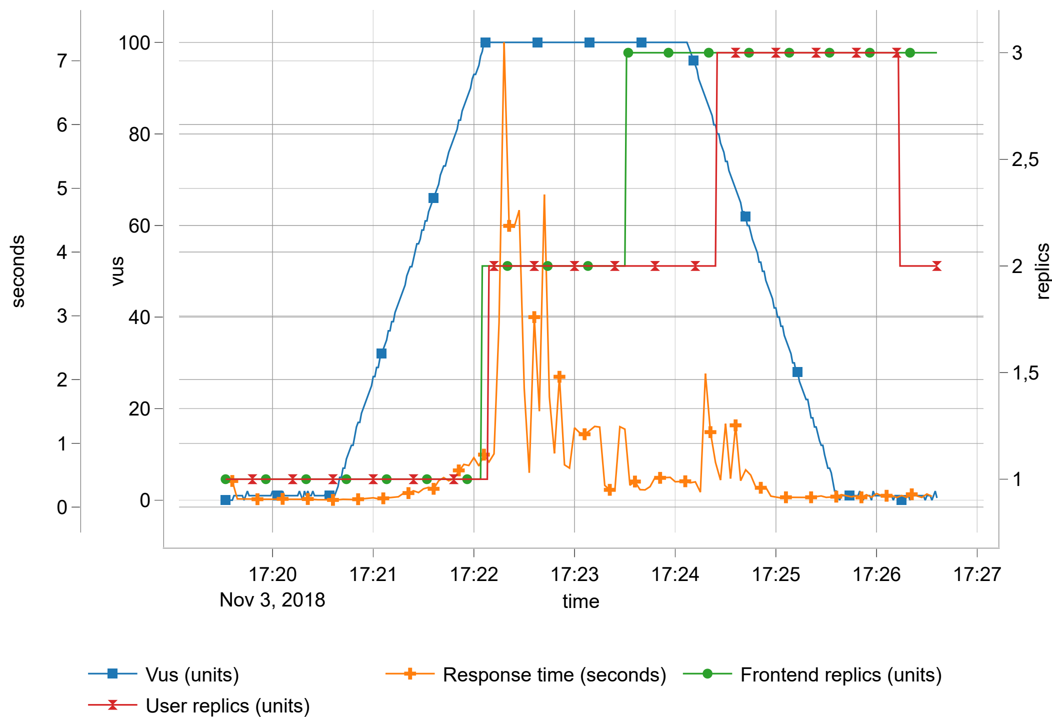


Figura 5.43: Teste de carga ao efetuar logins e registos de utilizadores, com replicação na *cloud* e na *edge* (Portland).

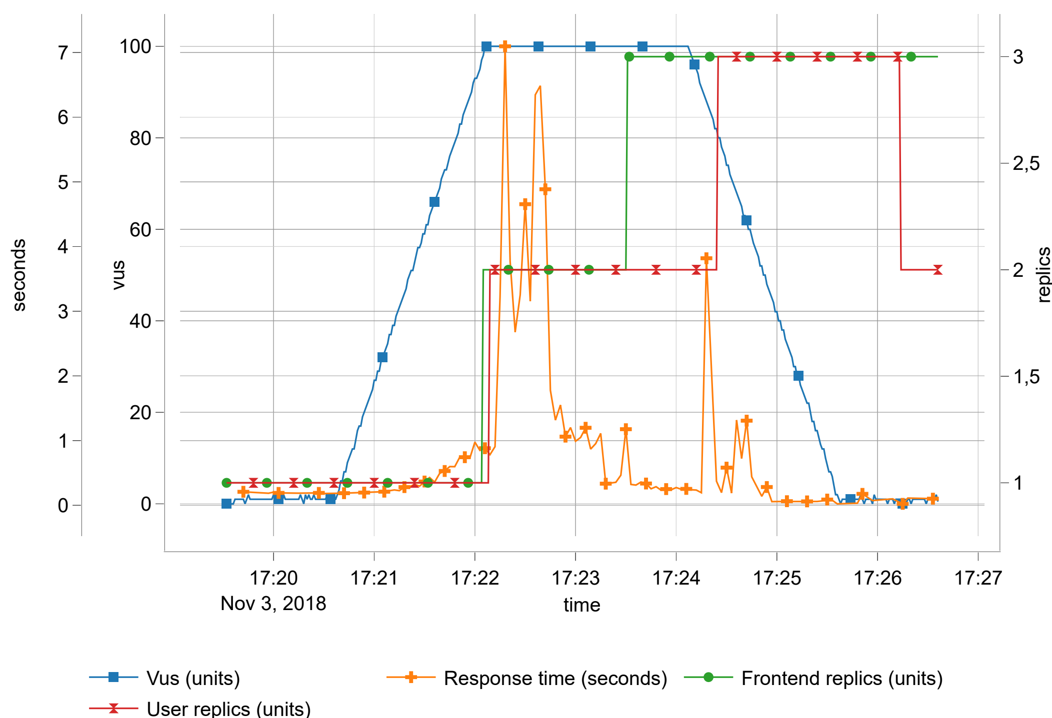


Figura 5.44: Teste de carga ao efetuar logins e registos de utilizadores, com replicação na *cloud* e na *edge* (Londres).

Tabela 5.13: Valores dos logins e registos de utilizadores, com replicação na *cloud* e na *edge* (média).

Tipo de valor	Média (s)	Desvio padrão (s)	Máximo (s)
Tempo de resposta	0,6	1,0	7,2

Tabela 5.14: Valores dos logins e registos de utilizadores, com replicação na *cloud* e na *edge* (Portland).

Tipo de valor	Média (s)	Desvio padrão (s)	Máximo (s)
Tempo de resposta	0,6	1,0	7,3

Tabela 5.15: Valores dos logins e registos de utilizadores, com replicação na *cloud* e na *edge* (Londres).

Tipo de valor	Média (s)	Desvio padrão (s)	Máximo (s)
Tempo de resposta	0,8	1,3	7,1

Tabela 5.16: Diferença dos valores do acesso entre os testes dos logins e registos de utilizadores com replicação apenas na *cloud* e com replicação na *cloud* e na *edge* (Londres).

Tipo de valor	Média	Desvio padrão	Máximo
Tempo de resposta	-0,3 s (-38%)	-1,9 s (-146%)	-15,9 s (-224%)

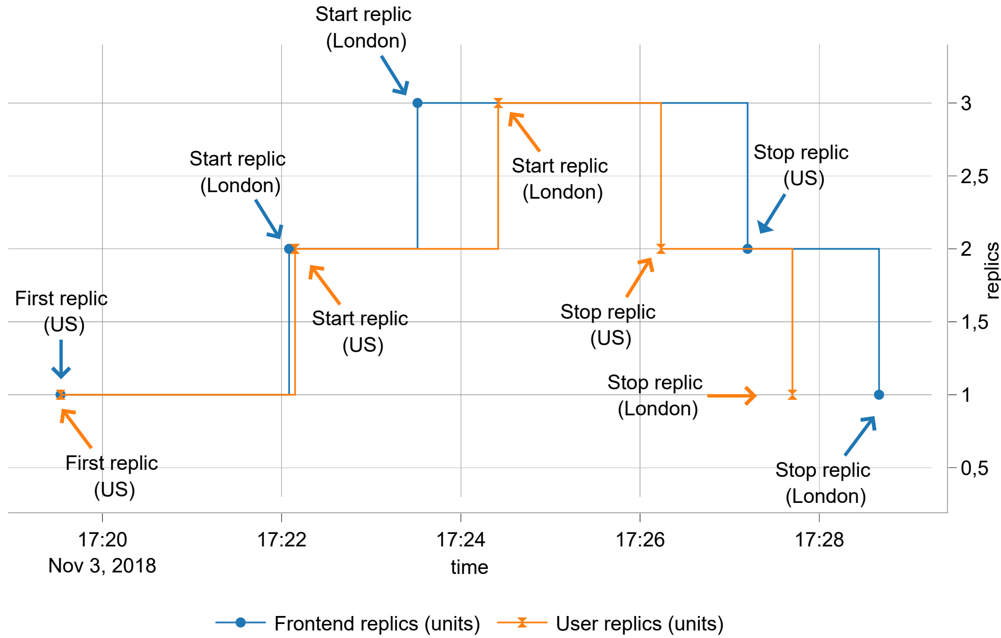


Figura 5.45: Variação de réplicas no teste de carga ao efetuar logins e registos de utilizadores, com replicação na *cloud* e na *edge*.

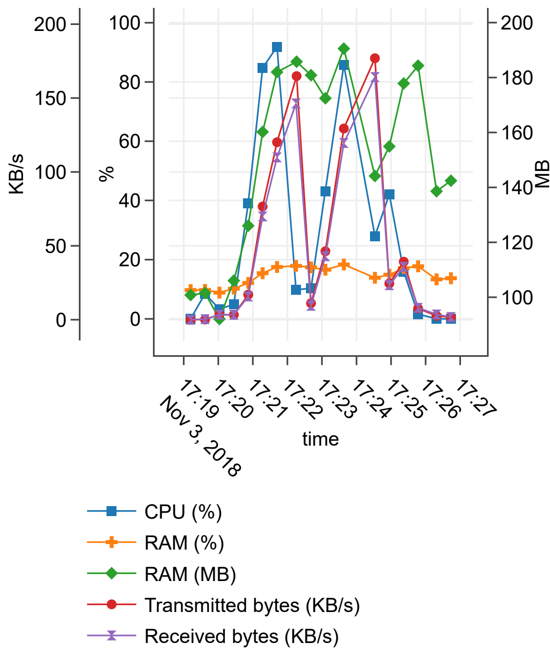


Figura 5.46: Métricas da 1ª réplica do serviço Frontend, durante o teste de carga ao efetuar logins e registos de utilizadores, com replicação na *cloud* e na *edge*.

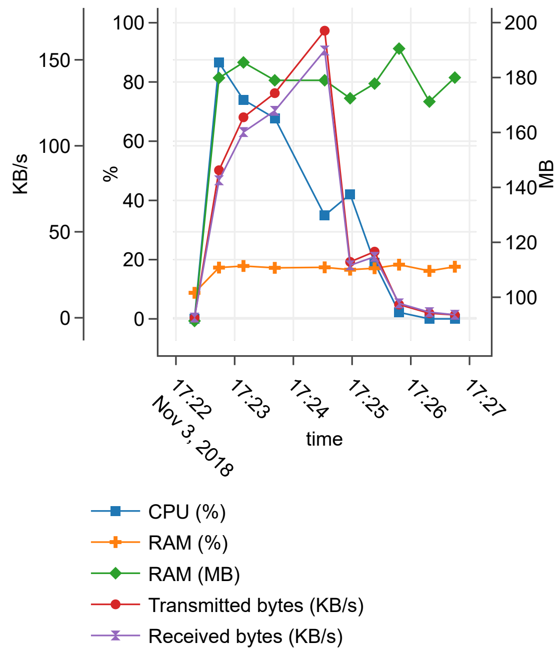


Figura 5.47: Métricas da 2ª réplica do serviço Frontend, durante o teste de carga ao efetuar logins e registos de utilizadores, com replicação na *cloud* e na *edge*.

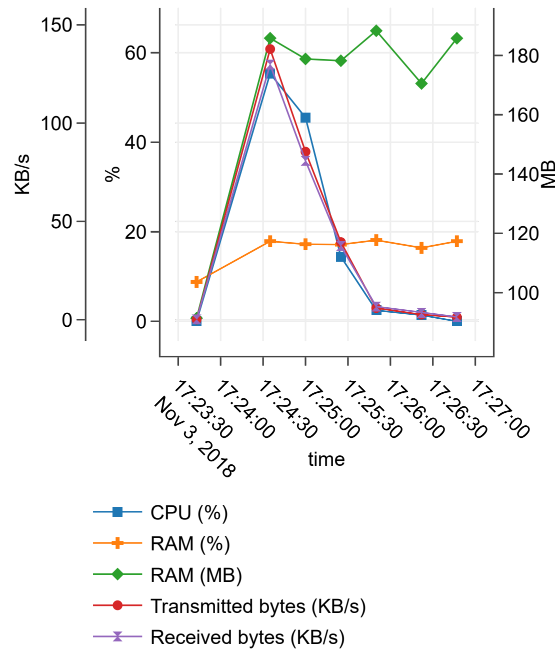


Figura 5.48: Métricas da 3ª réplica do serviço Frontend, durante o teste de carga ao efetuar logins e registos de utilizadores, com replicação na *cloud* e na *edge*.

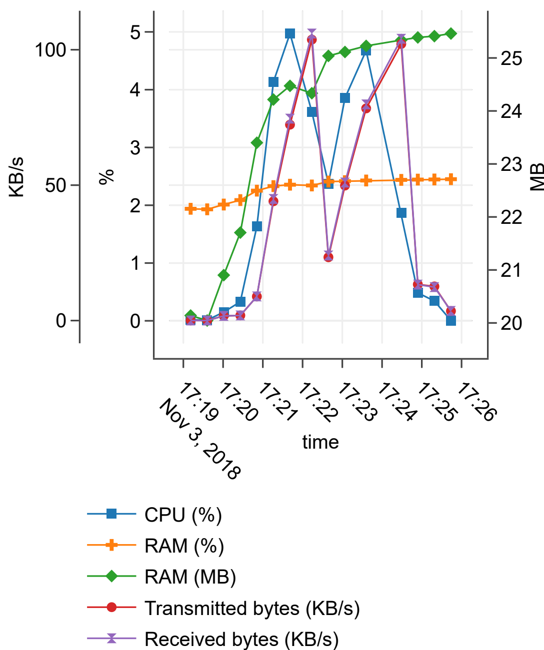


Figura 5.49: Métricas da 1ª réplica do serviço User, durante o teste de carga ao efetuar logins e registos de utilizadores, com replicação na *cloud* e na *edge*.

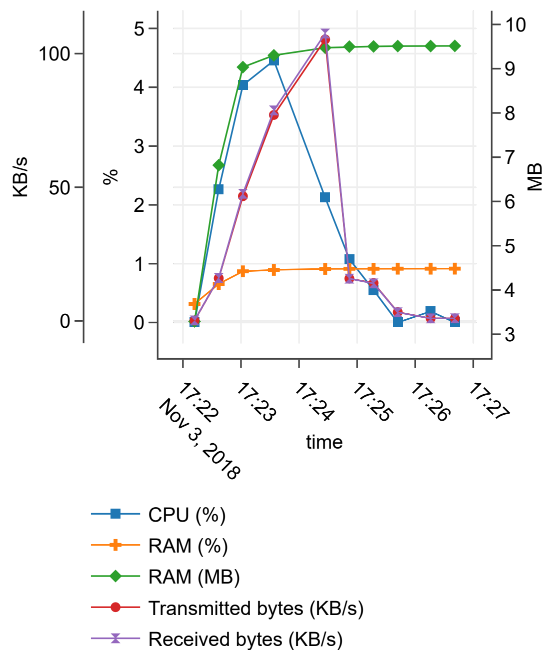


Figura 5.50: Métricas da 2ª réplica do serviço User, durante o teste de carga ao efetuar logins e registos de utilizadores, com replicação na *cloud* e na *edge*.

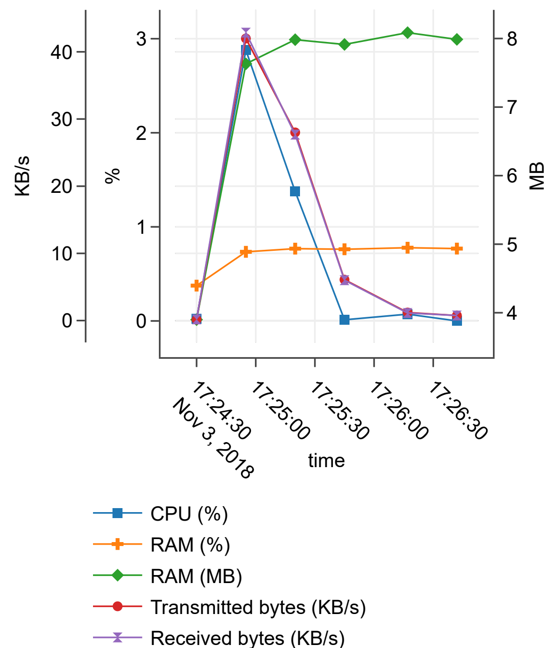


Figura 5.51: Métricas da 3ª réplica do serviço User, durante o teste de carga ao efetuar logins e registos de utilizadores, com replicação na *cloud* e na *edge*.

5.4 Discussão

Com a avaliação realizada ao sistema, através da utilização da aplicação *Sock Shop*, foi possível validar o funcionamento da replicação dos micro-serviços, um dos mecanismos principais do componente de gestão de micro-serviços para manter a QoS das aplicações.

Com os dois tipos de testes de carga realizados, Catálogo dos produtos, e Login e registo de utilizadores, foi possível verificar que, através da configuração de regras de replicação apropriadas para cada tipo de micro-serviço, é possível melhorar a QoS. Concretamente, ao nível dos tempos de resposta dos micro-serviços, e também remover réplicas de serviços quando não estão a ser necessárias, através da configuração de regra de remoção, isto, de uma forma automática e transparente tanto para os donos das aplicações, como para os utilizadores finais.

A utilização de nós da *edge* para complementar a *cloud*, é um ponto bastante importante, pois como foi confirmado com os testes realizados, através da sua utilização foi possível diminuir os tempos de resposta das operações. No caso do teste 5.3.1.1, os tempos de resposta diminuíram em 420%, em comparação à utilização de nós apenas na *cloud*, um valor bastante significativo e que indica que uma solução com infraestruturas híbridas pode ser bastante benéfica para o comportamento das aplicações, traduzindo-se em melhorias reais para os utilizadores.

Os custos de replicação, isto é, o tempo que demora a realizar uma replicação de um micro-serviço, são também bastante satisfatórios. Os tempos permitem que o sistema se

consiga adaptar às variações de carga, sendo preferível, em caso geral, a transferência prévia dos micro-serviços para minimizar este custo.

Capítulo 6

Conclusões e trabalho futuro

Este capítulo tem como objetivo apresentar algumas conclusões relativamente ao conteúdo abordado nesta dissertação, bem como referir alguns aspetos que possam ser melhorados ao nível da solução proposta.

6.1 Conclusões

Nesta dissertação foi apresentado um protótipo no contexto da gestão automática de aplicações de micro-serviços em ambientes *cloud*/*edge*, na dimensão particular de replicação/migração de micro-serviços em nós da infraestrutura heterogénea. O objetivo é reduzir a latência no acesso às aplicações e melhorar a qualidade de serviço (QoS) percebida pelos utilizadores, e reduzir o volume de dados a enviar dos nós da *edge* para os *data centers* na *cloud*. Tirar partido do número crescente de nós computacionais na *edge*, permite ainda contribuir para a redução dos custos computacionais e energéticos nos *data centers* da *cloud*, bem como contribuir para a privacidade dos dados, ao permitir que eles sejam processados na *edge*, próximo do local onde são gerados.

A larga maioria das soluções existentes, à altura do desenvolvimento deste trabalho, baseia-se sobretudo na replicação dinâmica de serviços na *cloud* não incluindo localizações na *edge*, ou aborda os benefícios da utilização de uma estrutura heterogénea *cloud*/*edge* a um nível mais teórico. Disponibilizar um sistema que permita a gestão automática de aplicações de micro-serviços tirando partido destes contextos heterogéneos é um objetivo mais alargado, para o qual este trabalho contribui.

O protótipo apresentado consiste num sistema simplificado com ênfase na execução transparente de micro-serviços na *cloud* e na *edge*, quer para os donos das aplicações quer para os utilizadores finais. O componente de gestão automática de micro-serviços possibilita a definição de regras e métricas a utilizar na migração/replicação dinâmica na

cloud e na *edge*, e melhorar a utilização/ocupação dos nós disponíveis para a execução dos micro-serviços. Por exemplo, o sistema permite a consolidação de recursos também para nós da *edge*, ao detetar nós que apresentem uma baixa utilização dos seus recursos. Os serviços aí existentes são migrados para um outro nó (próximo) disponível, considerando que não há implicações significativas na latência dos serviços percebida pelos utilizadores. O nó subutilizado é colocado em suspensão, levando a uma diminuição nos custos de infraestrutura.

A abordagem seguida para a implementação das decisões de migração e replicação de micro-serviços tem como base um conjunto de regras reconfiguráveis dinamicamente, quer ao nível das aplicações quer ao nível dos nós disponíveis na plataforma *cloud*/*edge*. No primeiro caso, a definição das regras é da responsabilidade dos donos das aplicações e incide sobre cada micro-serviço. As regras de um micro-serviço aceitam como parâmetros um conjunto de métricas, e implicam uma de três ações, a migração de uma réplica, a criação de uma nova réplica, ou a eliminação de uma réplica.

As métricas são obtidas através de componentes que estão dedicados à monitorização dos serviços em execução, obtendo os valores atuais de utilização dos recursos (por exemplo, CPU, RAM do *container* que suporta a sua execução), número de acessos ao serviço, etc. Periodicamente, o componente de gestão de micro-serviços executa as regras configuradas para cada um dos serviços em execução e obtém a decisão a aplicar. Porém, para cada tipo de serviço é apenas tomada uma decisão numa dada iteração do processo de reconfiguração. Ou seja, é obtida uma decisão para cada réplica, mas o resultado final é uma decisão conjunta, em que esta pode consistir na inclusão de uma réplica, migração de uma réplica, a eliminação de uma réplica, ou não realizar nenhuma ação, caso não seja necessário. A aplicação destas decisões leva então à reconfiguração dinâmica do sistema em termos do número de réplicas de cada micro-serviço e sua localização.

No segundo caso, os nós onde são executados os serviços também podem ter regras associadas e, neste caso, o seu resultado é a decisão sobre se se deve iniciar ou parar um nó. O processo de reconfiguração desenvolve-se de forma idêntica, sendo tomada uma decisão conjunta para todos os nós, isto é, adicionar um novo nó, a paragem de um nó, ou caso não seja necessário, não realizar nenhuma ação.

Para testar as funcionalidades de migração e replicação, bem como os seus benefícios reais para os utilizadores, isto é, a nível dos tempos de resposta dos serviços, foi utilizado como caso de estudo, a *Sock Shop*, um protótipo de uma aplicação web de comércio eletrónico de venda de meias composta por micro-serviços.

A avaliação permitiu comprovar que o sistema consegue decidir corretamente, com base nas regras definidas, quando deve replicar um dado serviço, havendo melhorias significativas para os utilizadores em termos de tempos de resposta dos serviços. Com a utilização de uma infraestrutura híbrida, composta pela *cloud* e pela *edge*, ainda é possível diminuir mais esses tempos.

Concluindo, os objetivos propostos inicialmente foram atingidos, ainda que não fossem implementadas todas as funcionalidades (a funcionalidade de migração de serviços),

revelando o potencial que o sistema pode ter na gestão automática de aplicações de micro-serviços em ambientes de *cloud* e *edge*, permitindo garantir a melhor qualidade de serviços aos utilizadores.

6.2 Trabalho futuro

O componente de gestão de micro-serviços foi desenvolvido, estando em fase de protótipo para testar a viabilidade dos conceitos abordados. Existem certos aspetos que podem ser melhorados, tanto a nível do próprio componente, como em relação aos outros subcomponentes de sistema. Estas melhorias incluem:

Componente de gestão de micro-serviços

- Implementar a funcionalidade de migração dos micro-serviços;
- Permitir aquando da decisão de replicação de um serviço, que sejam iniciadas várias réplicas simultaneamente, se assim se justificar;
- Incorporar prioridade nas regras em relação à preferência em replicar/migrar na *edge* ou na *cloud*;
- Utilizar um maior conjunto de parâmetros (CPU, espaço em disco, etc.) podendo estes ser especificados para cada nó, para verificar se um nó tem capacidade disponível para executar *containers*;
- Permitir iniciar novos nós na *cloud* em todas as regiões disponíveis, conforme o pretendido, e não apenas numa só;
- Permitir que as aplicações de micro-serviços sejam iniciadas em múltiplas regiões simultaneamente;
- Integração com outros fornecedores de *cloud*, como o Azure, para além da AWS, de forma a fornecer maiores garantias. Caso um falhe, será utilizado outro, ou mesmo dar a possibilidade de escolha aos donos das aplicações do fornecedor que prefere utilizar.

Subcomponentes Existe ainda a possibilidade de melhorar os componentes que dão suporte à execução dos micro-serviços, como o componente de registo de serviços, utilizado para a comunicação entre micro-serviços, e o componente de balanceamento de carga de serviços de *frontend*.

- **Componente de registo de serviços (*service registry*)**:
  - Possibilidade de adicionar mais réplicas deste componente no sistema, em tempo de execução, de forma que o tempo de consulta dos *endpoints* por parte dos micro-serviços seja o mais rápido possível.

- **Componente de balanceamento de carga**:
  - Melhorar a escolha das réplicas com base na origem do pedido, a carga e o tempo médio de resposta cada réplica.

Bibliografia

[1] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar e M. Steinder. "Performance Evaluation of Microservices Architectures using Containers". Em: *CoRR* abs/1511.02043 (2015). arXiv: 1511.02043. URL: http://arxiv.org/abs/1511.02043.

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

[3] A. A. Badawy, G. Yessin, V. K. Narayana, D. Mayhew e T. A. El-Ghazawi. "Optimizing thin client caches for mobile cloud computing: : Design space exploration using genetic algorithms". Em: *Concurrency and Computation: Practice and Experience* 29.11 (2017). DOI: 10.1002/cpe.4048. URL: https://doi.org/10.1002/cpe.4048.

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

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

[6] E. Casalicchio. "Autonomic Orchestration of Containers: Problem Definition and Research Challenges". Em: *10th EAI International Conference on Performance Evaluation Methodologies and Tools, VALUETOOLS 2016, Taormina, Italy, 25th-28th Oct 2016*. 2016. DOI: 10.4108/eai.25-10-2016.2266649. URL: https://doi.org/10.4108/eai.25-10-2016.2266649.

[7] E. Casalicchio e V. Perciballi. "Auto-Scaling of Containers: The Impact of Relative and Absolute Metrics". Em: *2nd IEEE International Workshops on Foundations and Applications of Self* Systems, FAS*W@SASO/ICCAC 2017, Tucson, AZ, USA, September 18-22, 2017*. 2017, pp. 207–214. DOI: 10.1109/FAS-W.2017.149. URL: http://doi.ieeecomputersociety.org/10.1109/FAS-W.2017.149.

[8] M. Champion. *How We Built Our Stack For Shipping at Scale*. Ed. por HubSpot. https://product.hubspot.com/blog/how-we-built-our-stack-for-shipping-at-scale. Consultado: Jan. de 2018. Mar. de 2015.

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

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

[11] Confluent, Inc. *Building a Microservices Ecosystem with Kafka Streams and KSQL*. https://www.confluent.io/blog/building-a-microservices-ecosystem-with-kafka-streams-and-ksql/. Consultado: Jan. de 2018. Nov. de 2017.

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

[13] A. Currie. *What is Cloud Native? - Containers Solutions*. https://container-solutions.com/what-is-cloud-native/. Consultado: Ago. de 2018. Mai. de 2017.

[14] A. V. Dastjerdi e R. Buyya. "Fog Computing: Helping the Internet of Things Realize Its Potential". Em: *IEEE Computer* 49.8 (2016), pp. 112–116. DOI: 10.1109/MC.2016.245. URL: https://doi.org/10.1109/MC.2016.245.

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

[16] N. Dragoni, S. Giallorenzo, A. Lluch-Lafuente, M. Mazzara, F. Montesi, R. Mustafin e L. Safina. "Microservices: Yesterday, Today, and Tomorrow". Em: *Present and Ulterior Software Engineering*. 2017, pp. 195–216. DOI: 10.1007/978-3-319-67425-4_12. URL: https://doi.org/10.1007/978-3-319-67425-4_12.

[17] A. Earnshaw. *Top Benefits of Continuous Delivery: An Overview*. https://puppet.com/blog/top-benefits-of-continuous-delivery-an-overview. Consultado: Jan. de 2018. Dez. de 2014.

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

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

[20] C. N. C. Foundation. *CNCF Cloud Native Definition*. https://github.com/cncf/toc/blob/master/DEFINITION.md. Consultado: Ago. de 2018. Jun. de 2018.

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

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

[23] Gajotres.net. *Top 8 Java RESTful Micro Frameworks – Pros/Cons*. https://www.gajotres.net/best-available-java-restful-micro-frameworks/. Consultado: Jan. de 2018. Out. de 2017.

[24] M. Garriga. "Towards a Taxonomy of Microservices Architectures". Em: *Software Engineering and Formal Methods - SEFM 2017 Collocated Workshops: DataMod, FAACS, MSE, CoSim-CPS, and FOCLASA, Trento, Italy, September 4-5, 2017, Revised Selected Papers*. 2017, pp. 203–218. DOI: 10.1007/978-3-319-74781-1_15. URL: https://doi.org/10.1007/978-3-319-74781-1_15.

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

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

[27] C. Harvey e A. Patrizio. *AWS vs. Azure vs. Google: Cloud Comparison*. Ed. por Datamation. https://www.datamation.com/cloud-computing/aws-vs.-azure-vs.-google-cloud-comparison.html. Consultado: Jan. de 2018. Dez. de 2017.

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

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

[30] H. Khazaei, R. Ravichandiran, B. Park, H. Bannazadeh, A. Tizghadam e A. Leon-Garcia. "Elascale: Autoscaling and Monitoring as a Service". Em: *CoRR* abs/1711.03204 (2017). arXiv: 1711.03204. URL: http://arxiv.org/abs/1711.03204.

[31] F. Klinaku, M. Frank e S. Becker. "CAUS: An Elasticity Controller for a Containerized Microservice". Em: *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering, ICPE 2018, Berlin, Germany, April 09-13, 2018*. 2018, pp. 93–98. DOI: 10.1145/3185768.3186296. URL: http://doi.acm.org/10.1145/3185768.3186296.

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

[33] Kubernetes. *Horizontal Pod Autoscaler*. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/. Consultado: Ago. de 2018. Ago. de 2017.

[34] J. Leitão, M. C. Gomes, N. Preguiça, P. Costa, V. Duarte, D. Mealha, A. Carrusca e A. Lameirinhas. "A Case for Autonomic Microservices for Hybrid Cloud/Edge Applications - (under submission)". Em: (2018).

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

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

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

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

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

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

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

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

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

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

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

[46] Rancher. *Playing Catch-up with Docker and Containers*. https://rancher.com/playing-catch-docker-containers/. Consultado: Ago. de 2018. Fev. de 2017.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[65] Sysdig. *2018 Docker Usage Report*. https://sysdig.com/blog/2018-docker-usage-report/. Consultado: Ago. de 2018. Mai. de 2017.

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

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

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

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

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

[71] N. Wang, B. Varghese, M. Matthaiou e D. S. Nikolopoulos. “ENORM: A Framework For Edge NOde Resource Management”. Em: *CoRR* abs/1709.04061 (2017). arXiv: 1709.04061. URL: http://arxiv.org/abs/1709.04061.

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

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

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

[75] S. Yi, C. Li e Q. Li. “A Survey of Fog Computing: Concepts, Applications and Issues”. Em: *Proceedings of the 2015 Workshop on Mobile Big Data, Mobidata@MobiHoc 2015, Hangzhou, China, June 21, 2015*. 2015, pp. 37–42. DOI: 10.1145/2757384.2757397. URL: https://doi.org/10.1145/2757384.2757397.

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