Pedro Correia

Licenciado em Engenharia Informática

Monitoração para suporte de Auto-Gestão em Aplicações de Micro-serviços

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática

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

Co-orientador: Vítor Manuel Alves Duarte,
Professor Auxiliar,
Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa

Júri

Presidente:	Prof. Drª. Carmen Pires Morgado
Arguente:	Prof. Dr. Luís Assunção
Vogal:	Prof. Drª. Maria Cecília Gomes

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Dezembro, 2019

Monitoração para suporte de Auto-Gestão em Aplicações de Micro-serviços

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

Aos meus pais e irmã.

Agradecimentos

Gostaria de expressar a minha sincera gratidão a todos aqueles que contribuíram direta ou indiretamente para a elaboração desta dissertação.

Aos meus orientadores, Professora Doutora Maria Cecília Gomes e Professor Doutor Vítor Duarte, por terem aceite realizar este trabalho comigo. Agradeço o apoio e disponibilidade constantes, bem como todo o conhecimento e ideias que me transmitiram.

A todos os professores da Faculdade de Ciências e Tecnologia com os quais eu tive oportunidade de aprender e de me tornar num melhor profissional.

A todos os meus amigos que me apoiam, que contribuem para a minha felicidade e com quem partilhei muitos bons momentos.

Ao meu pai, pela motivação e suporte incondicional, essenciais para a concretização deste objetivo. Agradeço-lhe por ser o meu melhor amigo.

À minha mãe, a quem guardo um abraço para celebrarmos um dia mais tarde. Até lá, continuo a agradecer-lhe com palavras o carinho, apoio e valores que me deu e continua a dar.

À minha irmã, pela força, ajuda, alegria que proporciona e por estar sempre presente.

A todos os meus restantes familiares que se disponibilizam para me ajudar a chegar mais longe.

Resumo

Com a computação *cloud*, os recursos computacionais tornaram-se facilmente acessíveis, facultando assim o rápido *deploy* de aplicações/serviços que beneficiam principalmente da escalabilidade oferecida pelos fornecedores de serviço. Devido ao aumento do volume de dados recebidos pelos sistemas *cloud*, observa-se um aumento da latência percebida pelas aplicações clientes, resultando na degradação da qualidade de serviço (QoS), incompatível com determinado tipo de aplicações (e.g. de tempo real/quase real). Como solução a este problema, surgiram os sistemas híbridos que combinam os recursos da *cloud* com os recursos de variados dispositivos entre a *cloud* e a *edge* – computação *fog*/*edge*. Estes recursos compreendem dispositivos muito heterogéneos, de diferentes capacidades, e que se encontram geograficamente distribuídos, próximos dos clientes de aplicações ou fontes de dados.

Os paradigmas das arquiteturas de *software* também evoluíram como efeito da necessidade de simplificar o desenvolvimento das aplicações e aumentar a sua escalabilidade. Com isto, adotou-se a arquitetura de micro-serviços, em que as aplicações monolíticas são divididas em serviços responsáveis por funcionalidades independentes. Dada a pequena dimensão e desacoplamento dos micro-serviços e avanços nas tecnologias de virtualização de recursos, é possível fazer *deploy* de *containers* que executam micro-serviços nos dispositivos no *continuum cloud*/*edge*. Este *deploy* deve considerar a variabilidade da carga dos nós computacionais, bem como o volume de dados gerados e/ou volume de acessos a partir de pontos particulares na periferia da rede, como forma de garantir uma boa gestão de recursos e a QoS necessária. O resultado é, no entanto, um sistema computacional heterogéneo, altamente dinâmico, apresentando uma gestão complexa e que requer um sistema de monitorização capaz de responder à variabilidade nas suas diferentes dimensões.

Esta dissertação apresenta um protótipo aplicacional de monitorização adaptativo, capaz de recolher e disponibilizar métricas com frequências variáveis e a diferentes níveis de um sistema distribuído composto por aplicações dinâmicas de micro-serviços executando sobre plataformas de nós heterogéneos no *continuum cloud*/*edge*. O sistema a monitorizar assume-se de larga escala, quer em número de componentes e sua dispersão geográfica, quer na variabilidade apresentada em termos da infraestrutura e aplicações.

O sistema de monitorização apresenta diferentes mecanismos de adaptação a esse dinamismo, com flexibilidade na seleção e frequência de métricas e na definição de mecanismos de alerta. Tal permite uma monitorização continuada, adequada aos requisitos, sem incorrer em sobrecarga adicional dos nós e da rede. As características dos seus componentes definem uma arquitetura de monitorização hierárquica, flexível e com propriedades de escalabilidade, adequada à complexidade do sistema alvo.

A avaliação realizada permitiu verificar o funcionamento e adaptabilidade do sistema de monitorização no contexto *cloud*/*edge* e de um número variável de micro-serviços. Foram também avaliadas diferentes dimensões de performance (desempenho), incluindo latências, uso de memória e CPU e tráfego na rede.

Abstract

With cloud computing, computing resources have become more accessible, thus enabling the rapid deployment of applications/services that benefit primarily from the scalability offered by service providers. Due to the increased volume of data received by cloud systems, there is an increase in latency perceived by client applications, resulting in degradation of the quality of service (QoS), incompatible with certain types of applications (e.g. real-time/near real-time). As a solution to this problem, hybrid systems have emerged that combine the capabilities of the cloud with the resources of various devices between the cloud and the edge - fog/edge computing. These resources include very heterogeneous devices of varying capacities that are geographically distributed near application clients or data sources.

The paradigms of software architectures also evolved as an effect of the need to simplify the development of applications and increase their scalability. With this, the architecture of microservices was adopted, in which the monolithic applications are divided into services responsible for independent functionalities. Given the small size and decoupling of microservices and advances in resource virtualization technologies, it is possible to deploy containers that run microservices on devices in the cloud/edge *continuum*. This deploy should consider the variability of computational node load, as well as the amount of data generated and/or access volume from particular points on the network periphery, as a means of ensuring good resource management and the required QoS. However, the result is a heterogeneous, highly dynamic computer system with complex management that requires a monitoring system capable of responding to variability in its different dimensions.

This dissertation presents an adaptive monitoring prototype capable of collecting and providing metrics with variable frequencies and at different levels of a distributed system composed of microservice applications running on heterogeneous node platforms in the cloud/edge *continuum*. The system to be monitored is large-scale, both in number of components and their geographical dispersion, as well as in the variability presented in terms of infrastructure and applications.

The monitoring system has different mechanisms of adaptation to this dynamism, with flexibility in the selection and frequency of metrics and in the definition of alerting mechanisms. This allows for continuous monitoring, appropriate to the requirements,

without incurring additional node and network overload. The characteristics of its components define a flexible, scalable, hierarchical monitoring architecture that fits the complexity of the target system.

The evaluation made it possible to verify the functioning and adaptability of the monitoring system in the cloud/edge context with a variable number of microservices. Different performance dimensions were also evaluated, including latencies, memory and CPU usage, and network traffic.

Índice

Lista de Figuras

Lista de Tabelas

Listagens

Siglas

API *Application Programming Interface*

CA *Collection Agent*

CPU *Central Processing Unit*

DNS *Domain Name System*

DoS *Denial of Service*

DSP *Digital Signal Processors*

FA *Federation Agent*

HTTP *Hypertext Transfer Protocol*

IaaS *Infrastructure as a Service*

IoT *Internet of Things*

IP *Internet Protocol*

IT *Information Technology*

JSON JavaScript *Object Notation*

P2P *Peer-to-peer*

PaaS *Platform as a Service*

QoE *Quality of Experience*

QoS *Quality of Service*

RA *Root Agent*

RAM *Random Access Memory*

REST *Representational State Transfer*

SaaS *Software as a Service*

SLA *Service Level Agreement*

SOA *Service-Oriented Architecture*

TCP *Transmission Control Protocol*

UI *User Interface*

URL *Uniform Resource Locator*

VM *Virtual Machine*

WAN *Wide Area Network*

Capítulo 1

Introdução

O aumento do volume de dados que, atualmente e no futuro, têm de ser armazenados e processados por muitas empresas obriga a que existam atividades constantes de manutenção e gestão da infraestrutura de *hardware*. Os custos necessários para tal, a velocidade e agilidade para a sua concretização, a segurança inerente aos dados, bem como as preocupações relacionadas com o aprimoramento da infraestrutura, são fatores que levaram à criação de um paradigma de computação, onde a capacidade computacional e de armazenamento são oferecidas como serviços, surgindo assim a computação *cloud*.

A computação *cloud* [23, 39, 50] refere-se às aplicações oferecidas como serviços acessíveis pela *Internet* e ao *hardware* e sistemas de *software* dos centros de processamento que as providenciam. Consiste num modelo que oferece suporte a aplicações, incluindo de *big data* e *analytics*, através da disponibilização de recursos computacionais, recorrendo à virtualização (servidores, armazenamento, *networking*) através da rede e de forma rápida, e à medida das necessidades de cada momento. Com a explosão de dispositivos móveis e no domínio de *Internet of Things* (IoT) e, consequente emergência de aplicações que tiram partido desses dispositivos, bem como o aumento de aplicações *bandwidth-intensive*, a *cloud* fica, por vezes, sobrecarregada com a quantidade de dados gerados que têm de ser processados e armazenados. A existência de aplicações sensíveis à latência, como por exemplo aplicações de jogos em tempo real ou um sistema de deteção e gestão de incêndios, inviabiliza as soluções que sejam apenas baseadas na *cloud*. A possibilidade de degradação da qualidade de serviço providenciada aos clientes refletida, por exemplo, em tempos de resposta elevados, constitui outro problema significante.

Os novos paradigmas de computação *fog*/*edge* estendem os recursos computacionais desde a *cloud* até aos dispositivos finais (*end devices*), e.g. através de *proxies*, *access points*, *routers*, entre outros. Estes paradigmas permitem complementar a computação *cloud* dando suporte ao armazenamento e pré-processamento de dados mais perto do local

onde são gerados, bem como à execução de aplicações mais próximas dos clientes. A definição inicial de computação na *edge* considera apenas os nós a um salto (*hop*) dos dispositivos finais, depois complementada com suporte da *cloud*. Dado que muitos recursos na *edge* são limitados, e por forma a evitar a sua contenção, a computação *fog* surgiu como forma de integrar os dispositivos no *continuum* desde a periferia da rede até aos recursos na *cloud*. Algumas definições mais recentes de computação *edge* [54] consideram também que a "*edge*" compreende uma periferia alargada com dispositivos no caminho até à *cloud*. Este trabalho considera também esta definição de *edge* mais lata e coincidente com a definição de *fog*.

Ao aproximar as computações aos clientes e fontes de dados, a computação *fog*/*edge* possibilita respostas mais rápidas dos serviços bem como a redução do volume de dados a transferir para os centros de dados da *cloud*. Com isto, diminui-se a congestão da rede e acelera-se o processo de tomada de decisão, sendo as aplicações sensíveis à latência processadas na periferia, enquanto que as aplicações tolerantes à latência (*delay-tolerant*) ou computacionalmente intensivas são processadas na *cloud*. Neste tipo de *frameworks fog* é possível o *deployment* de *containers* que executam micro-serviços [13, 43, 48] em dispositivos da periferia e que lhes proporcionam um alto grau de automatização, *deployment*, elasticidade e reconfiguração em tempo de execução.

O aparecimento da arquitetura de *software* de micro-serviços deu-se devido às dificuldades de manutenção e modificação das características de escalabilidade que as aplicações monolíticas apresentam. A ideia consiste em dividir a aplicação monolítica em vários serviços específicos, cada um responsável por uma capacidade de negócio e independentes entre si. Para além dos micro-serviços oferecerem as características desejadas do IoT de comunicação leve, *deployment* independente, pouca gestão central e tecnologias independentes de desenvolvimento, também oferece uma arquitetura ideal para aplicações com propriedades de escalabilidade, dinâmicas e distribuídas na *cloud*. Dado que estas aplicações se caracterizam por um número elevado de micro-serviços que comunicam entre si e que podem falhar, são usadas tecnologias de orquestração de *containers*, como por exemplo o Docker Swarm. Todavia, a granularidade fina dos módulos obriga o aumento de gestão e coordenação do sistema. São necessárias soluções automáticas responsáveis pelo *deployment*, controlo de ciclo de vida e comunicação das aplicações.

Técnicas de monitorização [1, 22, 52] mais flexíveis e que tenham em consideração plataformas emergentes, tornam-se imprescindíveis para concretizar os objetivos de gestão de recursos ao longo do *continuum cloud*/*edge*, bem como os objetivos das aplicações que deles dependem e também dos seus clientes. O aumento da complexidade da *cloud*, proporcional ao aumento de recursos e volume de tráfego, obriga ao uso de atividades de monitorização que recorrem à análise de métricas, para gerir os serviços de maneira eficiente e otimizá-los. A monitorização dos recursos físicos e virtuais relaciona-se diretamente com a disponibilidade, capacidade e outros requisitos de qualidade.

1.1 Problema

O objetivo a longo prazo do trabalho consiste em implementar um sistema de monitorização autónomo em que ambos os contextos de *edge* e *cloud* estão presentes e que considera as suas características. Estas infraestruturas são dinâmicas, e.g. por falha e adição de nós, sendo que nós móveis (e.g. *laptops*, *smartphones* e outros) não são considerados como constituintes das mesmas. É também altamente variável o desempenho das aplicações que são executadas na *edge*, dependendo de fatores como a qualidade de conexão da rede ou o número de pedidos a serem processados. É assim imprescindível fazer o rastreamento destas mudanças dinâmicas para garantir que os requisitos de qualidade de serviço são satisfeitos. O sistema de monitorização deve:

- Providenciar informação útil em tempo real, nos locais onde é necessária;
- Ser adaptável à elasticidade do sistema alvo e reagir rapidamente ao dinamismo dos recursos;
- Suportar ações de adaptação/reconfiguração dinâmica a nível da aplicação, e.g. ser capaz de monitorizar novos alvos de interesse na presença da migração e replicação de serviços;
- Ter uma arquitetura com propriedades de escalabilidade, de modo a disponibilizar novos monitores sempre e próximo de onde sejam necessários, de forma a responder aos pontos anteriores.

O grande número de nós da *edge*, dispersos entre si, que executam micro-serviços e com capacidades computacionais diferentes, pode tornar as abordagens de monitorização e orquestração apenas baseadas na *cloud*, menos eficientes ou mesmo inviáveis face à escalabilidade necessária. A consequência principal deste tipo de abordagem está no aumento de tempo de reação a mudanças na *edge* (a nível computacional e de acessos dos utilizadores) que é percebida na *cloud* e que pode interferir com a qualidade de serviço providenciada aos utilizadores finais.
A abordagem do trabalho já desenvolvido no Departamento de Informática FCT/UNL [32] que constitui a base para esta dissertação, oferece uma solução que tira partido dos nós da *edge*. A sua arquitetura está representada, de forma simplificada, na figura 1.1. A camada da *edge* suportando a execução de micro-serviços, é também usada para tarefas de monitorização, através do uso de monitores locais com capacidades de coleta, processamento, filtragem de dados e mecanismos de alerta. Este sistema de monitorização na *edge* expõe as métricas para o sistema de monitorização central. Desta forma, o tráfego de dados para a *cloud* é reduzido, enviando apenas os dados necessários, o que diminui substancialmente a carga da *cloud*.

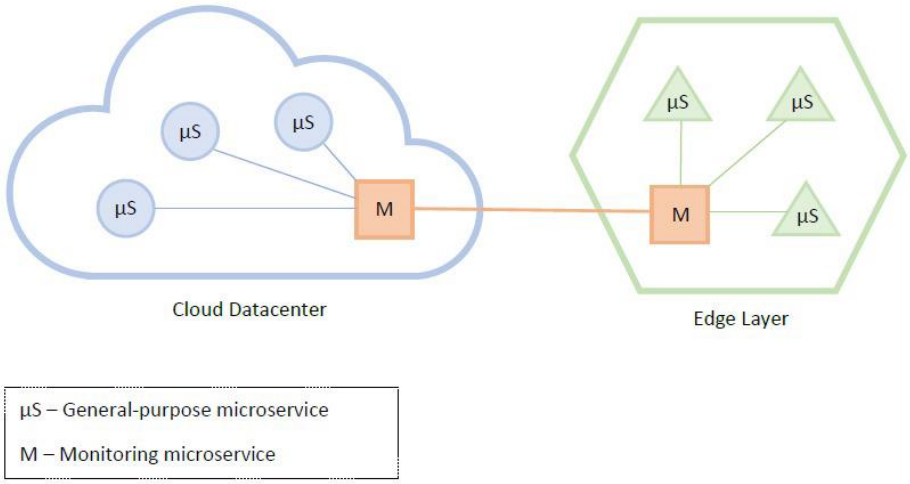

Figura 1.1: Arquitetura simplificada da solução anterior [32]

1.2 Objetivos

O objetivo deste trabalho prende-se com a extensão do sistema de monitorização anterior por forma a torná-lo flexível a novas parametrizações e adaptável a novos cenários de execução, tirando o melhor partido de uma infraestrutura *cloud/edge* que suporta a execução de aplicações adaptáveis baseadas em micro-serviços. Em particular, o novo sistema deve ter em consideração a heterogeneidade dos nós bem como a imprevisibilidade do ciclo de vida dos micro-serviços executados. O trabalho visa também complementar o trabalho existente de maneira a oferecer o máximo de informação relevante, assim como ser configurável dinamicamente. O sistema de monitorização deve poder ajustar-se e responder da melhor forma às exigências/necessidades das aplicações clientes, e que possam estar, ou não, diretamente relacionadas com as alterações da infraestrutura. Neste sentido, é necessário que o sistema atenda aos requisitos seguintes:

- Suporte políticas de adaptação de escalabilidade do sistema de monitorização em ambientes dinâmicos de larga escala. As características da solução de monitorização, como mecanismos de armazenamento de dados ou os recursos usados nas atividades de monitorização, podem afetar a escalabilidade de aplicações;

- Suporte ações de adaptação em resposta a alterações nas aplicações, como por exemplo, a migração de serviços, uma vez que os micro-serviços podem ser *deployed* e executados em diferentes *hosts* num intervalo de segundos;

- Obtenha informação acerca da infraestrutura e reaja rapidamente aos acontecimentos do sistema: a saída e entrada de nós, a disponibilidade, indisponibilidade e o estado dos mesmos;

- Altere a sua própria arquitetura dinamicamente. A complexidade resultante de orquestrar um número elevado de micro-serviços dispostos em diferentes localizações e dispondo de diferentes recursos, pode tornar ineficiente uma arquitetura baseada num monitor centralizado, resultando na degradação da qualidade de serviço;
- Recolha métricas a diferentes níveis (aplicação, *container* e *host*);
- Disponibilize opções para controlar a atividade de monitorização. Possibilitar que esta seja configurada à medida para cada alvo de monitorização (e.g especificar o intervalo de tempo entre recolhas de métricas consecutivas ou o tipo de métricas a serem coletadas no momento);
- Disponha de um mecanismo dinâmico de avaliação de regras com base nas métricas.

1.3 Contribuições

As principais contribuições do trabalho são:

- Análise e desenho das novas funcionalidades que estendem a arquitetura existente (secção 3.2);
- Protótipo que implementa as funcionalidades de controlo e gestão da infraestrutura e dos monitores. Mais especificamente:
    a) Gere os serviços necessários para a monitorização no *cluster* pelo qual é responsável;
    b) Obtém o estado do sistema num dado momento: todos os serviços significativos para a monitorização e informação sobre estes (e.g. posicionamento, identificação, carga);
    c) Comanda a atribuição dos alvos de monitorização aos monitores;
    d) Avalia regras com base nas métricas que recolhe dos monitores;
    e) Disponibiliza técnicas de adaptação perante sobrecarga dos monitores (particionamento e replicação);
    f) Permite alterar a arquitetura dinamicamente (cria/remove/migra monitores).
- Melhoramento do protótipo responsável pela monitorização dos micro-serviços. As funcionalidades adicionadas foram:
    a) Monitorização de micro-serviços externos ao nó onde o monitor está alojado;
    b) Recolha de métricas a diferentes níveis;
    c) Mecanismo de regras dinâmico;
    d) Auto-monitorização.

- Avaliação e validação do protótipo, de forma a verificar a viabilidade da solução perante os objetivos definidos e demonstrar os seus benefícios face a diferentes cenários.

1.4 Organização do documento

Seguindo o capítulo da introdução, o capítulo 2 diz respeito ao estado da arte, onde se apresentam os conceitos relacionados com o problema. O capítulo 3 apresenta a solução proposta, demonstrando a arquitetura da solução e detalhando as funcionalidades de cada componente desenvolvido. O capítulo 4 aborda os detalhes de implementação considerados relevantes. O capítulo 5, diz respeito à avaliação e validação da solução, onde são descritos os métodos utilizados para testar os componentes e os seus respetivos resultados. Por fim, o capítulo 6 apresenta uma conclusão geral sobre a solução concebida e algumas ideias que possam vir a complementar o sistema num trabalho futuro.

Capítulo 2

Estado da Arte

O presente capítulo é dedicado ao desenvolvimento do estado da arte das tecnologias relevantes ao problema da monitorização de micro-serviços no *continuum cloud*/*edge*. Primeiro é endereçado o tema da computação *cloud*, seguido da computação *fog*/*edge*. Depois, aborda-se a arquitetura de micro-serviços. De seguida, como objetivo principal, é feito o estudo sobre a monitorização, que se divide em duas partes: *cloud* e *edge*. Por fim, aborda-se, mais detalhadamente, o tema da virtualização e da tecnologia Docker.

2.1 Computação *cloud*

Ao longo do tempo, foi possível concluir que, em certos casos, várias entidades podem beneficiar do uso de sistemas de grande escala para o processamento e armazenamento de dados, remotamente, através da *Internet*. O surgimento dos modelos de computação *network-centric computing* e *network-centric content*, *grid computing*, *utility computing* e *cloud computing*, deu-se com avanços tecnológicos, realçando o avanço da rede.
Para o utilizador final, individual ou coletivo, a necessidade de diminuir os custos de computação, bem como o desacoplamento à manutenção do sistema e *software* incentivou o uso de recursos computacionais distribuídos com o intuito de atingir um objetivo comum – computação *grid*/*grid computing* [39, 50].

Computação *grid* consiste num conjunto de computadores interligados pela rede (privada ou pública), que manipula grande quantidade de dados, onde cada nó usa os seus recursos de processamento, memória e armazenamento para executar tarefas específicas. Os dispositivos *grid* caracterizam-se por serem heterogéneos e dispersos geograficamente (desacoplados), apesar de se fazerem parecer com um "super computador virtual". Dado que o *distributed grid* é formado por recursos que pertencem a várias entidades (*administrative domains*), pode acontecer que haja participantes maliciosos. A heterogeneidade

dos nós e a inexistência de uma gestão central, torna a tolerância a falhas, a otimização de recursos e balanceamento de carga extremamente difíceis de alcançar. Para além destas desvantagens, a heterogeneidade exige a existência de plataformas *middleware* e de linguagens *cross-platform* [23, 39].

Computação *utility* refere-se a um modelo de negócio, onde existe um fornecedor de serviço (*service provider*) que disponibiliza serviços, por exemplo de computação, armazenamento ou aplicações, que são cobrados aos clientes. Ao contrário da computação *grid*, existe um fornecedor de serviço que opera e gere toda a infraestrutura e recursos. Computação *utility* constitui um dos *service models* mais populares devido a providenciar flexibilidade e boa economia. Uma vez sendo a computação *utility* um modelo de negócio e não uma tecnologia específica, a computação *cloud* constitui o caminho para a mesma adotado pelas grandes empresas de Information Technology (IT) tais como a Amazon e a Google.

Computação *cloud* refere-se aos serviços e aplicações que são executados numa rede distribuída usando recursos virtuais e que são acedidos por protocolos da *Internet*. A *cloud* caracteriza-se por ser homogénea, por partilhar a mesma segurança entre os diversos nós do sistema, a mesma gestão de recursos e outras políticas. Os recursos oferecidos aos clientes são virtuais e ilimitados, e as características físicas do sistema são abstraídas do utilizador [39, 50].

A U.S *National Institute of Standards and Technology* (NIST) divide a computação *cloud* em *service models* e *deployment models* [40], permitindo assim determinar os diferentes tipos de sistema *cloud* que existem. O *deployment model* especifica o propósito da *cloud*:

- ***Cloud* pública**: Os recursos da *cloud* são propriedade de um fornecedor de serviço e são oferecidos pela *internet*. Toda a gestão de *hardware*, *software* e outras infraestruturas é gerido pelo *cloud provider*. Estes recursos são partilhados entre as diferentes organizações clientes. O acesso aos serviços pode ser feito através de um *web browser*, ou aplicação móvel, por exemplo.

- ***Cloud* privada**: Recursos da *cloud* são usados exclusivamente por uma organização. Todos os serviços são acessíveis por uma rede privada e o *hardware* e *software* são apenas acedidos pela organização. Permite personalizar o ambiente de acordo com as necessidades, aumentar a segurança e possibilita escalabilidade recorrendo a *clouds* públicas.

- ***Cloud* híbrida**: Combinação de mais que um tipo de *cloud*. Assim, é possível beneficiar das vantagens de ambos. Aqui, é possível que os dados e aplicações sejam transferidos entre as *clouds* privadas e públicas. Como vantagens, o *cloud provider* pode manter dados sensíveis numa infraestrutura privada e beneficiar de recursos adicionais da *cloud* pública.

Os *service models* especificam o tipo de serviço que o fornecedor de serviço oferece:

- ***Infrastructure as a Service* (IaaS)**: O IaaS *service provider* gere toda a infraestrutura, sendo o cliente responsável por todos os aspetos do *deployment*: sistemas operativos, aplicações, interações de utilizadores com o sistema, segurança, entre outros. Disponibiliza acesso aos recursos: servidores, armazenamento e de rede.

- ***Platform as a Service* (PaaS)**: O PaaS *service provider* gere a infraestrutura, sistemas operativos e o *software*. O cliente pode usar ou fazer *deploy* de aplicações suportadas pelo fornecedor de serviço. O cliente é apenas responsável por gerir a aplicação *deployed*. Para além dos recursos computacionais disponibilizados, são oferecidas ferramentas para desenvolver, customizar e testar as aplicações.

- ***Software as a Service* (SaaS)**: Fornece um ambiente com aplicações, gestão da infraestrutura e *interface*. Os serviços são normalmente disponibilizados através de *interfaces*, sendo que a única responsabilidade do cliente é inserir e gerir os seus dados (e.g. Google Apps).

O paradigma de partilha de recursos da computação *cloud* apresenta novas oportunidades para os utilizadores e programadores. A metodologia *"pay-as-you-go"*, onde só são cobrados os recursos que são utilizados, proporciona características que alteram a economia das infraestruturas IT e permite novos tipos de acesso e modelos de negócio para *user applications*.

Aplicações relacionadas com ciência e engenharia, *data mining*, jogos e redes sociais podem beneficiar muito da computação *cloud*.

2.1.1 Características principais da computação *cloud*

A publicação pelo NIST [40] da definição da computação *cloud*, inclui as suas principais características:

- ***On-demand self-service***: Serviços são fornecidos "a pedido" sem necessidade de haver interação entre o cliente e o *cloud service provider*;

- **Amplo acesso à rede:** Recursos estão disponíveis na rede e são acessíveis através de mecanismos *standard* que promovem o uso de diversas plataformas heterogéneas tais como telemóveis, computadores portáteis e assistentes pessoais digitais;

- **Agrupamento de recursos**: Os recursos disponibilizados pelo *cloud service provider* são reunidos para servir vários clientes usando o modelo *multiple-tenant*. Neste modelo, uma única instância de *software* é executada num servidor e serve vários *tenants*. Um *tenant* é um grupo de utilizadores que têm acesso à instância com certos privilégios. Os diferentes recursos virtuais e físicos são dinamicamente atribuídos de acordo com os pedidos dos consumidores. Estes recursos incluem armazenamento, processamento, memória, largura de banda dos canais, máquinas virtuais e serviços *email*;

- **Rápida elasticidade**: Os recursos podem ser fornecidos de forma rápida e elástica, ou mesmo automática. Estes recursos para os consumidores parecem ilimitados e podem ser comprados em quaisquer quantidades em qualquer altura;

- **Serviço medido**: Os recursos podem ser medidos, controlados e reportados, proporcionando transparência para ambos o *cloud service provider* e o consumidor. Os serviços de computação *cloud* usam um mecanismo de medição capaz de controlar e otimizar o uso dos recursos. Isto implica que, por exemplo, a eletricidade é cobrada conforme o seu uso – *pay-as-you-go*. Tal como empresas de telecomunicações que vendem serviços de voz e dados, também os serviços IT são concedidos como serviços contratuais;

- **Usa virtualização**: Virtualização consiste na criação de um recurso virtual como, por exemplo, um servidor, sistema operativo ou armazenamento. Através da virtualização de servidores, é possível particionar um servidor físico de forma a ter várias instâncias de sistemas operativos a serem executados paralelamente num único computador.

2.1.2 Vantagens da computação *cloud*

Apresentam-se de seguida as principais vantagens [50] do uso da computação *cloud*:

- **Redução de custos**: Com a computação *cloud*, uma organização não é obrigada a reunir os recursos necessários para criar uma infraestrutura capaz para disponibilizar os seus serviços, eliminando-se assim o investimento inicial que seria necessário. Relativamente à manutenção, o custo também é reduzido. Estes serviços são *"pay-as-you-go"*;

- **Segurança**: O *cloud provider* tem pessoas dedicadas exclusivamente à monitorização da segurança, o que é menos provável de acontecer nos casos onde é usado *hardware* local;

- **Flexibilidade**: Por se confiar numa organização externa para cuidar do alojamento e da infraestrutura, todas as preocupações relacionadas com *upgrades* de computação ou armazenamento de dados desaparecem. Qualquer recurso necessário, é servido de forma instantânea, melhorando a eficiência da organização;

- **Mobilidade**: Permite o acesso à informação a partir de qualquer dispositivo, seja fixo ou móvel, em qualquer lugar, desde que exista ligação à *Internet*;

- ***Insights***: Muitos *cloud service providers* oferecem soluções e serviços que permitem a análise de dados (*cloud analytics*). Constitui então uma facilidade para as empresas fazerem rastreamento (*tracking*) dos mesmos, bem como analisar certas métricas. Isto também permite aumentar a eficiência da organização;

- **Controlo de qualidade**: Todos os documentos guardados apresentam um único formato. Todos os dados são consistentes e são mantidos registos de atualizações, evitando assim possíveis falhas humanas;

- **Recuperação**: Apesar de existirem situações incontroláveis que originam tempos de inatividade do sistema, os *cloud-based services* fornecem uma recuperação muito mais rápida em caso de emergência. Isto deve-se ao facto dos servidores virtuais serem independentes ao *hardware*, fazendo com que o sistema operativo, as aplicações e os dados consigam ser transferidos de forma segura para outro centro de dados usando ferramentas que tornam este processo automático;

- **Fiabilidade de armazenamento**: Toda a informação transferida para a *cloud* permanece segura através dos mecanismos que dispõe de replicação e tolerância a falhas. De outra forma, com *hardware* local, para se evitar a perda de dados de forma permanente, o cliente teria de igual modo implementar estes mecanismos e suportar os respetivos custos associados;

- ***Deployment* rápido**: Um sistema pode-se tornar totalmente funcional em minutos.

2.1.3 Desvantagens da computação *cloud*

Existem aplicações que, por outro lado, não beneficiam da *cloud* por diferentes razões. Enumeram-se de seguida algumas desvantagens [39, 50] do uso desta:

- **Customização**: Os serviços usados na *cloud* podem não ser tão customizáveis como desejável. A organização cliente tem de se restringir às aplicações e formatos da *cloud*. Aplicações executadas em *hardware* local podem ter mais funcionalidades;

- **Latência**: Existe latência inerente à conectividade *Wide Area Network* (WAN). A computação *cloud* não constitui a melhor solução para aplicações que impliquem grande movimento de dados entre sistemas;

- **Privacidade**: Todos os dados transferidos para a *cloud* deixam de estar sob controle da organização cliente. A organização fica suscetível a ataques de *hacking* e a roubo de dados sensíveis.

2.1.4 Desafios da computação *cloud*

Os desafios da computação *cloud* [2, 39, 50] são específicos a cada *service* e *delivery model*, apesar de todos terem origem na natureza da computação *utility* que se baseia na partilha e virtualização de recursos.

A segurança constitui possivelmente o maior desafio da computação *cloud*. O modelo SaaS está sempre suscetível a ataques de negação de serviço (DoS). Uma vez que a interação com o utilizador acontece apenas através de uma *interface*, os ataques não constituem

um grande desafio, já que é possível identificar os canais maliciosos. No modelo IaaS é mais preocupante devido à maior liberdade que é atribuída ao utilizador.

Relativamente à virtualização, todo o estado de uma máquina virtual (VM) é guardado num ficheiro que possibilita a migração e recuperação de dados. Contudo, é possível que uma VM maliciosa infete outros sistemas anteriormente validados caso, durante o processo de validação, a mesma tenha estado inativa.

Quanto à gestão de recursos, é necessário implementar várias políticas: controlo de admissão, alocação de capacidade, balanceamento de carga, otimização de energia e garantias de qualidade do serviço. Para tal, são usados controladores que necessitam de obter informação sobre o estado global do sistema, o que por vezes é impossível devido ao tamanho do mesmo. O desafio passa por utilizar uma informação parcial do sistema e atingir o mesmo resultado desejado.

2.1.5 Caso de estudo - Amazon Web Services (AWS)

A Amazon inicialmente criou uma infraestrutura para suportar os seus serviços de *e-commerce*. Depois, chegou à conclusão que a infraestrutura podia ser aumentada de forma a poder oferecer serviços de computação para outras empresas. Com isto, introduziu a Amazon Web Services[1], o primeiro fornecedor de serviços *cloud*, que se baseia no modelo IaaS.

Assim, começou por lançar em 2006 uma plataforma chamada EC2[2] que possibilitava a criação de instâncias em vários sistemas operativos. A instância era criada a partir da Amazon Machine Image e guardada no Simple Storage System (S3), serviço de armazenamento para guardar objetos e que suportava operações de *write*, *read* e *delete*.

Na EC2, cada máquina virtual funcionava como um servidor virtual privado e eram especificados os recursos máximos disponíveis, a *interface* de acesso e o custo por hora. Para além destes serviços principais, a Amazon também disponibiliza:

- **Elastic Block Store (EBS)**: Fornece *persistent block-level storage volumes*;
- **SimpleDB**: Base de dados não relacional que permite aos programadores guardarem e fazerem *queries* de *data items*;
- **Simple Queue Service (SQS)**: Constitui uma *message queue*. Permite a coordenação das atividades das várias instâncias EC2;
- **CloudWatch**: Infraestrutura de monitorização usada para coletar e fazer rastreamento de métricas com o fim de melhorar o desempenho e eficiência das aplicações;
- **Virtual Private Cloud (VPC)**: Permite fazer a gestão de serviços de segurança, *firewalls*, deteção de sistemas intrusos, entre outros;

[1] https://aws.amazon.com/
[2] https://aws.amazon.com/ec2/

- **Auto Scaling**: Explora a elasticidade da *cloud* e define a escala do grupo de instâncias EC2;
- **Elastic BeanStalk**: Interage com outros serviços disponíveis e lida com o *deployment*, provisionamento de capacidade, balanceamento da carga, *auto scaling* e funções de monitorização;
- **CloudFormation**: Cria uma *stack* que descreve a infraestrutura para uma aplicação.

Todos estes *cloud services* estão atualmente disponíveis em diferentes zonas do mundo. A Amazon cobra pelas instâncias EC2, pelo armazenamento EBS, pela transferência de dados, entre outros serviços [39].

2.2 Computação *edge*

Ao longo do tempo, é possível verificar a proximidade na interação entre as aplicações, dispositivos e os utilizadores humanos. Grande parte desta evolução, deve-se ao IoT e à procura de melhorar a distribuição de conteúdos na Internet. O IoT consiste na rede de dispositivos, univocamente identificados, que se encontram ligados à *Internet* e com capacidade de transferir dados ao longo da mesma, sem necessidade de existir intervenção humana para tal. Os dispositivos tanto podem ser sensores com a finalidade de tornar uma casa inteligente ou sistemas de grande escala como *smart cities*. Os ambientes de IoT geram grande quantidade de dados que são posteriormente processados e analisados. Uma vez que os *smart devices* são limitados nos recursos (bateria, computacionais, armazenamento e largura de banda), as aplicações de IoT são normalmente *deployed* na *cloud*, que garante recursos ilimitados a baixo custo, dado que os custos estão associados à sua utilização (modelo "*pay-as-you-go*").
Por outro lado, as aplicações de tipo *bandwidth-intensive*, como é o caso de *VoD*, 4K TV, *streaming* de vídeo e jogos *online*, constituem outras das principais causas da congestão da rede dada a quantidade de dados que produzem [8]. Também a solução usando apenas a *cloud*, pode tornar-se problemática para as aplicações que necessitam de respostas rápidas (*latency-sensitive*), como, por exemplo, o acesso a certas aplicações *web* e, ainda mais, aplicações de monitorização da área de saúde ou de emergência [31].
Isto deve-se ao facto que os centros de dados da *cloud* estão geograficamente centralizados e situados a uma grande distância dos dispositivos terminais. Para além do *delay* significativo que é provocado, a congestão da rede e a degradação da qualidade de serviço são outros problemas que advêm deste paradigma.

Como solução, é proposto o conceito de computação *edge*. A ideia consiste em dispor os serviços computacionais (processamento, armazenamento, *networking* e outros) para mais perto dos dispositivos terminais. Assim, aproximando os dados e a computação da periferia da rede diminui-se a sua congestão, acelera-se o processo de análise de dados e melhoram-se os tempos das respostas aos pedidos [37]. Adicionalmente, o paradigma

da computação *edge* permite obter informação acerca da localização (*context-awareness*) através da proximidade com os dispositivos terminais, que se revela essencial para providenciar recursos e serviços de forma eficiente.
A rede na periferia é normalmente constituída por dispositivos terminais (e.g. portáteis, telemóveis, *smart watches*, *set-up boxes*, sensores) e por dispositivos nas redes periféricas (e.g. *routers*, *wireless access points*, entre outros).

A computação *edge* original não está associada a nenhum serviço *cloud* (IaaS, PaaS, SaaS), estando sujeita a recursos limitados e a problemas de disponibilidade. Para contornar isto, os conceitos de *edge* e *cloud* foram conciliados. Há vários tipos de paradigmas [4, 5, 37] que estendem a camada *edge*, entre eles:

- **Computação *edge* móvel (MEC)**: Disponibiliza *edge servers* e *base stations* e pode ou não estar ligada a um centro de dados da *cloud*. As aplicações são *deployed* e as tarefas de processamento são executadas perto dos dispositivos terminais, resultando na redução da congestão da rede e diminuição de latência. Esta tecnologia oferece flexibilidade e *deployment* rápido de novos serviços. Os serviços oferecidos, bem como os recursos disponíveis, topologia da rede e *mobile edge applications* são geridos pelo *mobile edge orchestrator*. Este é responsável por providenciar informação em tempo real da rede e também informação dos dispositivos terminais ligados aos servidores.

- **Computação *cloud* móvel (MCC)**: Devido às limitações de recursos dos dispositivos móveis, MCC fornece servidores leves denominados *cloudlet* que são colocados perto da periferia da rede. Assim, o trabalho computacional de aplicações *compute-intensive* é movido dos dispositivos para estes servidores. Estes estão acessíveis a um passo, com alta largura de banda, disponibilizando assim pouca latência para as aplicações. A *cloudlet* assemelha-se a uma *cloud* de pequena escala, com recursos medianos, mas possivelmente ligada à *cloud*.

- **Computação *fog/edge***: Para além do uso dos componentes da periferia, os componentes da *core network* fazem também parte da infraestrutura computacional. Consiste numa arquitetura *n-tier*, oferecendo flexibilidade ao sistema, não podendo operar num modo *standalone*, precisando sempre da ligação com a *cloud*. Isto resolve contenção de recursos ao utilizar os recursos da *cloud*. Para além disto, a computação *fog* facilita o reconhecimento de localização, suporte de mobilidade, interações em tempo real, escalabilidade e interoperabilidade. A camada de computação *fog* é formada por um ou mais domínios que pertencem a um ou vários fornecedores. Estes domínios podem ter como componentes *core routers*, *regional servers*, WAN *switches*, entre outros.
  Beneficia também sobre a MCC e MEC já que nós *fog* podem ser inseridos mais próximos que *cloudlets* e *edge servers*. As comunicações entre os dispositivos terminais e a camada *fog* são realizadas através da *Local Area Network* (LAN), enquanto que

as comunicações com a *cloud* requerem WAN. Este paradigma pode estender *cloud based services* (IaaS,PaaS e SaaS) para a periferia da rede.

2.2.1 Vantagens da computação *edge*

As principais vantagens [5, 12] que a computação *edge* apresenta são:

- **Redução do tráfego na rede**: Dada a quantidade de dados e pedidos gerados pelos biliões de dispositivos ligados à *Internet*, torna-se ineficiente enviar todos estes dados não processados e pedidos para os serviços nos centros de dados da *cloud*. Com a inserção de várias camadas entre a periferia e a *cloud*, os dados e alguns pedidos podem passar por um processamento inicial, de filtragem e análise em nós próximos dos dispositivos terminais, reduzindo substancialmente o tráfico de dados para a *cloud*.
- **Adequação ao IoT**: Sistemas de IoT constituem um exemplo onde não é necessário o conhecimento global dos dados para servir certos pedidos. Por exemplo, faz sentido que o *Global Positioning System* (GPS) de um veículo inteligente, que indica os pontos de interesse na proximidade, seja servido por um nó da *edge* próximo da sua posição, em vez de um serviço noutro continente.
- **Latência baixa**: A distância aos *clouds centers* e consequente latência elevada, ou a indisponibilidade de serviço resultado de possíveis falhas de comunicação, tornam este tipo de computação incompatível com aplicações que requeiram menor latência na resposta, especialmente em aplicações em tempo real. Como exemplo, tem-se os carros autónomos que necessitam de informação atualizada que determina os comportamentos consoante o contexto. A computação *fog/edge* mitiga este problema com computação e respostas obtidas mais perto da periferia.
- **Escalabilidade**: Computação *edge* elimina parte da computação da *cloud*, limitando potenciais pontos de falha intermédios e pontos de contenção causados pelo envio de grandes quantidades de dados não processados.
- **Disponibilidade**: Torna a rede mais resiliente porque o sistema continua a operar mesmo quando os serviços *cloud* centrais estão inacessíveis.

2.2.2 Desafios da computação *edge*

A computação *edge* (incluindo *fog*) é um paradigma que faz uso de nós heterogéneos e desacoplados entre si. Estas características são as principais causadoras das dificuldades de *deployment* e desenvolvimento de aplicações na periferia da rede. Alguns dos principais desafios [11, 24, 53] que a computação *edge* enfrenta são:

- **Computação de propósito geral na *edge* (*General purpose computing*)**: Como a computação *edge* dispõe de um conjunto de nós entre os dispositivos da periferia e a

cloud – *access points*, *base stations*, *gateways*, *routers*, entre outros – o desafio está em rentabilizar os recursos destes nós, de forma a suportarem *general purpose computing*. Por exemplo, as *base stations* que incorporam *Digital Signal Processors* (DSP), podem não conseguir lidar com cargas de trabalho analíticas porque os DSP não estão desenhados para tal. A solução seria modificar os DSP por *general purpose* CPUs, o que requer grandes investimentos.

- **Descoberta dos nós da *edge***: Seleção de nós, dados os requisitos operacionais e o estado atual do sistema, feita por ferramentas de monitorização e *service brokers*. Estes usam técnicas, por exemplo de *benchmarking*, para mapear as tarefas aos recursos disponíveis de forma a melhorar o desempenho do sistema. Dada a heterogeneidade dos nós da *edge*, o desafio passa por não comprometer a experiência do utilizador, não aumentando a latência com o processo de descobrir a disponibilidade e capacidades dos recursos. A seleção dos nós recai sobre a boa proximidade entre os dispositivos terminais e os servidores da *edge* e a capacidade para servir o utilizador.

- **Particionar e distribuir tarefas**: A decisão de distribuir trabalho entre várias camadas pode levar ao aumento da latência e, portanto, precisa de ter em consideração várias métricas de latência, largura de banda, energia e custo. É essencial particionar o *workload* entre os diferentes nós da *edge* de forma eficiente e automática, sem especificar as capacidades ou localização dos mesmos.

- **Não comprometer a Qualidade do Serviço (QoS) e Qualidade da Experiência (QoE)**: Sistemas de gestão de recursos devem conseguir determinar dinamicamente que tarefas estão atribuídas à *cloud* ou a recursos da *edge* para que estas possam ser particionadas e agendadas de forma flexível, minimizando assim a latência e maximizando o *throughput*. Para tal, é necessário que estes sistemas tenham sempre informação acerca do uso dos nós da *edge* de forma a evitar a sobrecarga dos mesmos com *workloads*, garantindo a QoE.

- **Segurança**: Enquanto que na *cloud* os serviços estão centralizados, na computação *edge* os vários domínios podem pertencer a várias entidades que oferecem diferentes garantias de segurança, criando vulnerabilidades no sistema. Para além disso, quando a camada da *edge* é usada para *middleware* para proteger a *cloud* de dados maliciosos, acontece que existem nós da *edge* que não possuem recursos suficientes para suportar encriptação local eficiente, o que provoca o aumento da latência. Outro dos problemas deve-se à suscetibilidade a ataques dos componentes tradicionais da rede, usados pela computação *edge*. Finalmente, a implementação de técnicas de autenticação e autorização é dificultada pelos ambientes heterogéneos. A implementação destes mecanismos para garantir a integridade dos dados pode afetar a QoS.

- **Modelos de programação e arquiteturas**: Falta de flexibilidade e escalabilidade de *frameworks* para ambientes de computação *edge*. A *membership* dos nós *edge* é altamente dinâmica em termos de entrada, saída e falhas de nós, pelo que é incompatível com configurações estáticas.

2.2.3 Caso de estudo - Body Sensor Network (BSN)

Sistema de monitorização de saúde [31] que usa sensores com tamanho reduzido e com baterias de longa duração. Para obter eficiência no uso da energia, os dados transferidos apresentavam baixa resolução exceto quando detetada alguma atividade considerada urgente. Neste caso, para além da resolução dos dados aumentar, a taxa de transmissão de dados também aumentava. O utilizador possuía então um conjunto de sensores heterogéneos entre si – acelerómetro, eletrocardiograma, sensor de temperatura e humidade – que enviavam dados para uma *gateway*. Os sensores eram *deployed* em conjunto com uma *fog gateway* e comunicavam usando tecnologia *wireless* (ZigBee[3]). A *gateway* constituía um nó *fog* com capacidades computacionais (recorria a *containers* para disponibilizar serviços) e de rede, enviando dados para servidores remotos (*cloud*). Para além disso era utilizado um classificador de *machine learning* para treinar um modelo que detetava atividades humanas.

O *workflow* é constituído pelas fases de coleta e tratamento de dados, treinar e testar o modelo e *deployment*. A coleta de dados consiste no envio de dados não processados dos sensores para a *gateway*. Tratamento é a fase onde existe um pré-processamento (etiquetação, segmentação e imputação de dados) por parte da *gateway* dos dados recebidos. Treinar o modelo e testar é a fase onde o modelo é treinado e testado nos servidores remotos da *cloud*. No *deployment* real, eram ativados os eventos de emergência e a performance da rede era testada. O modelo pode, contudo, ser treinado na *gateway* na periferia da rede, garantindo menor latência e resposta rápida.
Foram realizados testes e chegou-se à conclusão que um sistema sem adaptabilidade e inteligência oferecia piores resultados em relação ao *delay*, *throughput* e confiabilidade.

2.3 Arquitetura orientada a serviços

A arquitetura orientada a serviços (SOA) consiste num estilo arquitetural que usa serviços como elementos para desenvolver aplicações. Os serviços disponibilizados pelas aplicações podem ser acedidos por outros componentes através da rede. Esta arquitetura tem como principais características a modularidade, encapsulamento, desacoplamento e separação de tarefas. A arquitetura SOA divide-se em 3 camadas [43]:

1. **Composição de serviços:** Engloba os cargos e funcionalidade para a consolidação de vários serviços num único serviço composto, usado por *service aggregators* ou

[3] https://zigbee.org/

service clients. Os serviços devem incluir capacidades de coordenação, conformidade, monitorização e QoS;

2. **Descrição e operações básicas:** Engloba serviços básicos, suas descrições, e operações básicas (publicação, *discovery*, *selection* e *binding*);

3. **Gestão:** Providencia suporte à gestão de plataformas de serviço, *deployment* de serviços e aplicações.

Um serviço consiste na implementação de uma funcionalidade de negócio. Os serviços são oferecidos por empresas e comunicam pela *Internet* fornecendo uma infraestrutura de computação distribuída. Caracterizam-se por serem componentes de *software* independentes à implementação, ou seja, que separam a *interface* da sua implementação, autocontidos e desacoplados. Podem ser descobertos dinamicamente e resultar da agregação de outros serviços (serviços compostos).
A implementação de uma arquitetura SOA usa geralmente *Web Services*. Estes sistemas suportam interação interoperável entre duas máquinas através da rede. A interoperabilidade é obtida através da utilização de padrões da *internet* tanto para a sua identificação (*Uniform Resource Identifier*) como para a comunicação. *Simple Object Access Protocol* (SOAP) e *Representational State Transfer* (REST) constituem as tecnologias mais usadas nas comunicações, que usam *Extensible Markup Language* (XML) para representar dados. *Web Services Description Language* (WSDL) para a descrição da *interface* e *Universal Description, Discovery and Integration* (UDDI) para o processo de *service discovery* [7, 36, 42].
SOA surgiu para combater os desafios que as aplicações monolíticas apresentam. A arquitetura monolítica consiste no desenho de aplicações de *software single-tiered*, ou seja, não usando modularidade. Todo o código do sistema está num *codebase* que é compilado e que produz um único artefacto. Os componentes do *software* estão interconectados e interdependentes entre si, partilhando a mesma memória e recursos, sendo a aplicação responsável por várias tarefas. Apesar desta arquitetura ter benefícios, como a facilidade nas fases do desenvolvimento inicial, de testes e *deployment*, também apresenta desafios [48] que se revelam incompatíveis com determinados requisitos de sistemas atuais:

- **Tamanho do *codebase***: Quanto maior a quantidade de código fonte usado para construir um sistema, maior será a dificuldade de adaptação ao código, especialmente para novos membros. Para além disso, as ferramentas do ambiente de desenvolvimento, bem como o *runtime container* usado podem ficar sobrecarregados, afetando a produtividade.

- **Atualizações menos frequentes**: Uma vez que todos os componentes são executados num único *container*, faz com que uma única mudança num dos componentes implique que todos os outros sejam *redeployed*. Isto faz com que todos os *developers* sejam afetados com todas as mudanças que ocorram, constituindo um obstáculo

para situações onde várias equipas são responsáveis por diferentes partes. A agilidade é reduzida e os tempos de resposta a pedidos de mudança são demorados.

- **Adoção de novas tecnologias**: A adoção de uma nova tecnologia num determinado componente irá afetar todo o sistema, resultando em possíveis erros. O tempo e custo de resolução são fatores essenciais que se perdem com esta arquitetura.
- **Escalabilidade**: Dividir o sistema por diferentes equipas, quando o tamanho do sistema assim o exige, torna-se impossível dada a dependência dos componentes, que requer a coordenação das equipas para qualquer alteração que ocorra.

Micro-serviços constitui o padrão arquitetural que se relaciona com a SOA, principalmente ao nível do desacoplamento de componentes. Contudo, distingue-se do mesmo uma vez que, enquanto que a SOA é uma iniciativa *enterprise-wide* para criar código reutilizável através da disponibilização de serviços com o objetivo de criar novas aplicações mais rapidamente, a arquitetura de micro-serviços direciona-se à aplicação de forma a torná-la ágil, escalável e resiliente [13].

2.3.1 Micro-serviços

De forma a contornar as limitações que a arquitetura monolítica apresenta e porque a complexidade dos sistemas cada vez é maior, surgiu a arquitetura de micro-serviços. Esta arquitetura consiste na composição de vários serviços específicos, onde os processos de cada serviço comunicam através de mecanismos *light-weight*. O objetivo é providenciar alta sustentabilidade e escalabilidade ao *software* criado. Os serviços são definidos por funcionalidades de negócio e são desacoplados entre si, melhorando assim a modularidade. Estes serviços são *deployed* de forma independente e executados numa máquina virtual ou *container* [13].
Apesar dos micro-serviços terem como base a SOA, as duas arquiteturas apresentam diferenças substanciais. Na SOA, a reutilização consiste no objetivo primário, enquanto que na arquitetura de micro-serviços, a reutilização pode resultar em dependências. SOA usa tipicamente comunicações síncronas, incompatíveis com o desempenho e independência desejadas dos micro-serviços. Na arquitetura de micro-serviços, ao contrário da SOA, pode ocorrer duplicação de dados dada a independência dos dados [29].

2.3.1.1 Características principais dos micro-serviços

As características principais da arquitetura de micro-serviços [34, 49] são:

- **Tamanho pequeno**: Os micro-serviços necessitam de ser focados num domínio específico. Necessitam de ter gestão do seu código fonte e *delivery pipeline* para desenvolvimento e produção (*deployment*);
- **Desacoplamento**: Característica essencial para tornar possível o *deployment* de um micro-serviço, sem necessidade de coordenação com quaisquer outros;

- **Neutralidade de tecnologias**: As tecnologias escolhidas em cada componente são definidas estritamente pela adequação da tecnologia à tarefa, independentemente das tecnologias usadas por outros componentes. A comunicação entre os vários componentes para o correto funcionamento do sistema dá-se através de *Application Programming Interfaces* (APIs) de linguagem neutra, por exemplo, REST;
- **Contexto limitado**: Cada um dos componentes não conhece qualquer detalhe de implementação de outros;
- **Organização dadas as funcionalidades de negócio**: Em vez de se separar a aplicação em várias partes e atribuí-las a equipas, por exemplo, equipas de *User Interface* (UI), equipas de servidor ou equipas de base de dados, a arquitetura de micro-serviços divide a aplicação por serviços que oferecem uma determinada funcionalidade de negócio – algo que a empresa faz que gera valor e que possa ser feito de forma independente a outros serviços. Isto requer equipas com membros especializados em todas as áreas;
- **Produtos**: Esta arquitetura evita o modelo de atribuir a responsabilidade pela manutenção do *software* à empresa cliente. Pelo contrário, a responsabilidade pertence à equipa que o desenvolveu. Cria uma relação contínua com o objetivo de perceber como o *software* pode ajudar os utilizadores a melhorar a funcionalidade de negócio;
- **Gestão de dados descentralizada**: Ao contrário do que acontece nas aplicações monolíticas que só têm uma base de dados e todas as componentes acedem aos dados nela contidos, na arquitetura de micro-serviços, cada funcionalidade de negócio, que corresponde a um micro-serviço, tem a sua própria base de dados. O acesso aos dados de outros componentes é possível através da *interface* do componente e não diretamente sobre a sua base de dados.

2.3.1.2 Vantagens dos micro-serviços

As principais vantagens [13, 42] que a arquitetura de micro-serviços apresenta são:

- **Heterogeneidade de tecnologias**: A possibilidade de usar várias tecnologias em cada um dos componentes de forma a melhorar o desempenho conforme a necessidade. Adicionalmente, a forma como são guardados os dados também é flexível, por exemplo, usar *graph-oriented database* ou *document-oriented data store*. A arquitetura de micro-serviços, com o desacoplamento entre componentes, permite a adoção de novas tecnologias mais rapidamente visto ser um processo localizado em que quaisquer outros componentes não serão afetados. Esta característica possibilita obter uma melhor perceção do impacto das mudanças na aplicação;
- **Resiliência**: O facto dos componentes estarem desacoplados e serem independentes entre si, faz com que se um componente do sistema falhar, o sistema continue a operar;

- **Escalabilidade**: Permite identificar os serviços que necessitam de propriedades de escalabilidade, permitindo fazer uma gestão de recursos de acordo com a carga em cada serviço. Por exemplo, é possível que cada serviço seja *deployed* e executado em máquinas diferentes, e que sejam criadas várias réplicas de um serviço que esteja sobrecarregado. Esta gestão focalizada em cada componente constitui uma solução menos dispendiosa comparativamente às soluções monolíticas, que necessitam de escalar o sistema como um todo;

- ***Deployment* rápido**: *Deployment* dos serviços aquando de uma mudança é independente do resto do sistema. Para além do código ser *deployed* mais rapidamente, o risco associado à mudança é menor por não interferir com o resto do sistema. Quando um erro ocorre, a origem do problema é encontrada mais rapidamente, fazendo-se um *rollback* para versões anteriores corretas. Para além disso, o processo de adicionar novas funcionalidades também é facilitado pelas mesmas razões;

- **Gestão organizacional**: Facilita o processo de perceber o sistema por parte de novos programadores e permite melhorar a distribuição de recursos humanos pelo sistema. Assim, devido ao facto do *codebase* atribuído a cada equipa ser menor, bem como o tamanho da equipa específica a cada serviço, aumenta a produtividade;

- **Reusabilidade**: Capacidade de reutilizar funcionalidades;

- **Facilidade de substituição:** O facto dos serviços terem granularidade fina e, portanto, *codebases* pequenos, torna os processos de reescrita ou substituição de componentes fazíveis, reduzindo o risco e o custo associado.

2.3.1.3 Desafios dos micro-serviços

As diferentes decisões que têm de ser tomadas no processo de criar um micro-serviço estão representadas na forma de taxonomia na figura 2.1. São encontrados desafios e problemas em aberto para alguns dos aspetos neste processo [20, 25, 48]:

- **No desenho:** As comunicações na rede e a complexidade de distribuição de tarefas são fatores que aumentam proporcionalmente ao número de micro-serviços, sendo importante encontrar o nível ideal de granularidade dos mesmos;

- **Na implementação:** Apesar da comunicação assíncrona permitir melhor desempenho que a síncrona, desacoplamento e tolerância a falhas, propriedades essenciais à arquitetura de micro-serviços, esta, por vezes, acaba por não ser implementada dada a facilidade de implementação e compreensão que a comunicação síncrona garante. A divisão de dados por micro-serviços constitui outro desafio;

- **No *deployment*:** Dado que as aplicações são geralmente executadas na *cloud*, os criadores das aplicações têm de se reger pela configuração específica do fornecedor

de serviços *cloud*. A solução será a criação de mecanismos padrão para o *deployment* e gestão de micro-serviços;

- **Em tempo de execução:** É provável que os micro-serviços venham a ser maioritariamente executados numa arquitetura *serverless* (cujo *service model* é *Function as a Service* (FaaS)), que é baseado em funções que são despoletadas por eventos. Escalabilidade, recursos, *deployment*, entre outros são fatores externos ao cliente. Este modelo de serviço apresenta alguns desafios: atribuição automática de computação para funções *serverless*, encontrar o nível ideal de granularidade e métodos para guardar o estado. A confiabilidade também constitui um desafio dado os mecanismos de integração, por exemplo *message passing*, serem não confiáveis. A sustentabilidade, validação e testes são áreas ainda pouco exploradas;

- **Na segurança:** O facto do sistema ser constituído por muitos componentes distribuídos, cria alvos menos robustos e portanto mais suscetíveis a ataques;

- **Nos aspetos organizacionais:** Deve ser usada a metodologia *DevOps* para adotar boas práticas nas equipas. *DevOps* consiste na combinação de práticas e ferramentas, de forma a construir, testar e lançar *software* de forma rápida e confiável;

- **Na monitorização:** Garantir QoE ao utilizador através da monitorização do sistema. Dado que os micro-serviços são executados em *containers* ou VMs, pode-se tornar difícil, por exemplo, medir o desempenho das várias interações desde o pedido do utilizador até ao fim da transação.

2.3.1.4 Caso de estudo - IBM Bluemix Console

Contexto: Bluemix [13] constitui uma plataforma de *cloud* desenvolvida pela IBM. Bluemix Console é o nome da aplicação, inicialmente monolítica (figura 2.2). O Bluemix estava dividido em duas grandes partes com objetivos diferentes: um *website* com conteúdo pesquisável e um *dashboard* acessível por autenticação. Com o aumento do tamanho e complexidade das partes constituintes, o custo de manutenção bem como as dificuldades de *deployment* e de compreensão pelas equipas também aumentaram.
Dado o estado do sistema e as dificuldades na sua evolução, a IBM decidiu transitar para uma arquitetura de micro-serviços. Os serviços da Landing Page e Solutions foram os primeiros a ser implementados. As tecnologias usadas foram Node.js com a *framework* Express e Dust.js como motor de renderização.
Solução: Nesta transição, surgiu o problema das *"common areas"*, ou seja, de como lidar com as partes necessárias por todos os serviços. Como solução, criaram-se micro-serviços dedicados a servir elementos comuns. Para tal, usou-se o protocolo *Hypertext Transfer Protocol* (HTTP) para as comunicações através da rede e OAuth2 para a autenticação.
O processo de criação dos micro-serviços deu-se de forma repetitiva, usando mecanismos DevOps, a partir do momento da resolução dos elementos partilhados – composição da

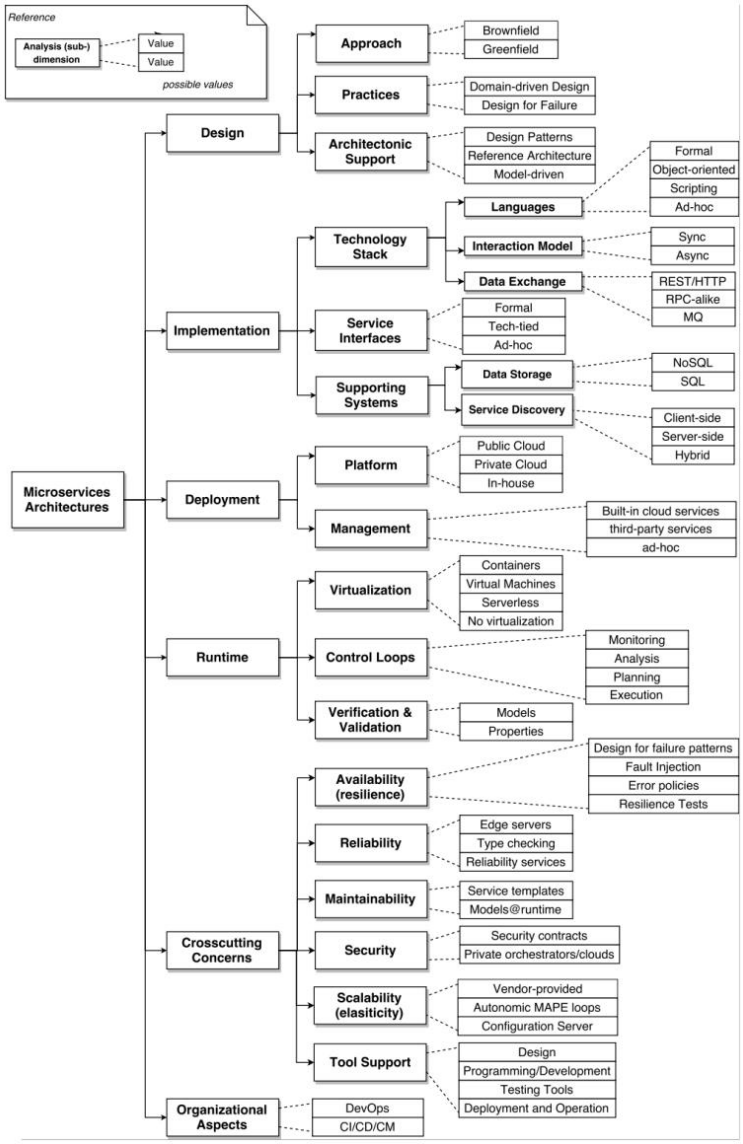

Figura 2.1: Taxonomia da arquitetura de micro-serviços [25]

(UI), autenticação e *reverse proxy*.
Cada micro-serviço corresponde a uma funcionalidade (figura 2.3), com hipótese de serem otimizados para determinadas tarefas (servindo conteúdo, assistindo utilizadores com serviços, *containers* e VMs de modo dinâmico).
Na fase de *deployment* e *runtime*, o Bluemix é executado no Bluemix (*self-hosting*), já que é compatível com o ambiente PaaS. Algumas das melhorias notadas com a mudança de arquitetura foram:

- *Auto-scaling*, ou seja, a remoção e adição de instâncias dinamicamente;
- A não necessidade de ferramentas de monitorização, dado que o Bluemix tem serviços para tal; e

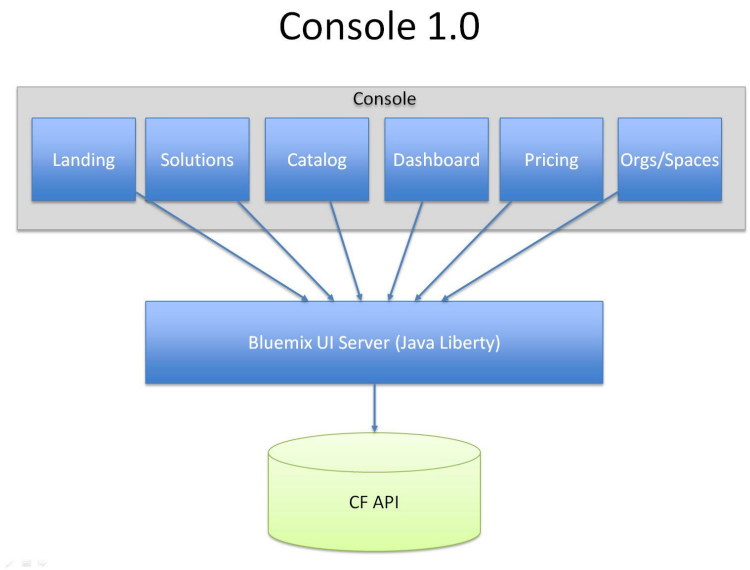

Figura 2.2: Arquitetura monolítica [13]

- A não necessidade de gerir o *message broker* devido à existência de serviços de *messaging* (IBM MQ Light) no catálogo de serviços Bluemix.

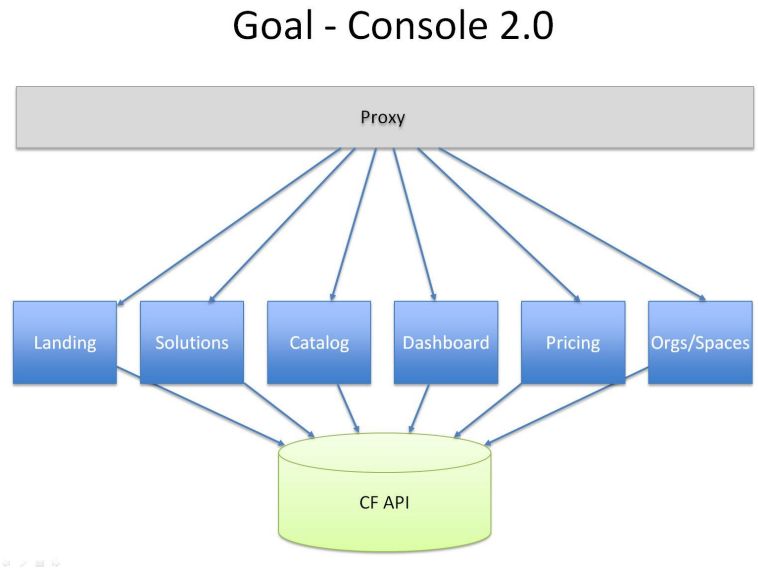

Figura 2.3: Arquitetura em micro-serviços [13]

2.4 Monitorização

O paradigma de computação *cloud* cada vez é mais usado dado as vantagens que oferece na oferta de serviços. Um sistema *cloud* de grande escala que recorre à virtualização de recursos requer mecanismos de gestão capazes de lidar com a sua complexidade e distribuição. É necessário garantir as características desejadas de elasticidade, escalabilidade,

recursos *"on-demand"*, disponibilidade, uso eficiente de energia, entre outros. Tal só é possível se existir um sistema de monitorização preciso, de granularidade fina e seguro.
O objetivo de um sistema de monitorização na *cloud* é monitorizar/gerir, através da análise de métricas, várias atividades da *cloud*. O uso destes sistemas aumenta, na perspetiva do *cloud provider*, a eficiência na gestão dos recursos disponíveis, no desempenho e na área da faturação dos serviços. Por sua vez, na perspetiva do utilizador de serviços *cloud*, o recurso à monitorização garante que o *Service Level Agreement* (SLA) é cumprido.
As sondas de monitorização, que realizam *polling*, podem ser integradas nas diferentes camadas da *cloud*:

1. **Instalação**: Infraestrutura física constituída pelos centros de processamento de dados e componentes da rede;
2. **Rede**: Ligações intra *cloud* e entre *cloud* e utilizador;
3. ***Hardware***: Componentes físicos e equipamento de rede;
4. **Sistema Operativo**: *Softwares* executados no *host* e respetivas VMs;
5. ***Middleware***: Camada entre o sistema operativo e a aplicação;
6. **Aplicação**: Aplicação executada no sistema *cloud*;
7. **Utilizador**: Aplicações executadas fora do ambiente da *cloud* (*browsers*, por exemplo).

As métricas dividem-se maioritariamente em duas categorias: de computação e de rede. As métricas de computação oferecem informação acerca das plataformas onde são executadas as aplicações: *throughput* do servidor, velocidade de CPU, troca de páginas de memória por segundo, tempo de resposta, entre outros.
De entre as métricas de rede, o *round trip time*, *jitter*, largura de banda disponível e perda de pacotes constituem alguns exemplos [1, 22, 52].

A monitorização apresenta vários benefícios para as várias atividades da computação *cloud* [1]:

- No planeamento de recursos, ajuda a determinar os recursos e capacidade necessárias de uma aplicação/serviço e o *workload* (quantidade de trabalho a ser processado por um determinado nó);
- Na gestão de recursos e capacidade usando recursos virtuais para lidar com a volatilidade dos mesmos e migração de *workloads* entre servidores, bem como das condições instáveis de rede para garantir disponibilidade;
- Na gestão do centro de processamento de dados, que controla as métricas desejadas do *hardware* e *software*, posteriormente processadas para o provisionamento de recursos, resolução de problemas, entre outros;

- Na gestão de SLAs, usada para certificar conformidade com atividades de auditoria e na realização de SLAs mais realistas e dinâmicos;

- Em termos de faturação, determinando os custos proporcionais ao uso dos serviços de acordo com o tipo de serviço (IaaS, SaaS, PaaS) e ao modelo de faturação do mesmo. As métricas tanto podem ser o número de utilizadores, como a utilização de CPU ou o número de VMs usadas. O cliente também beneficia com a monitorização certificando-se do uso e tendo uma forma de comparação com outros *cloud providers*. Isto é possível através dos componentes de apresentação de dados disponibilizados ou integrados nos sistemas de monitorização;

- Na resolução e deteção de problemas, já que implementando serviços de monitorização em vários nós de um sistema distribuído permite localizar muito mais rapidamente a origem do problema num dado componente de uma camada (observando um problema de desempenho ou *crash*, por exemplo);

- Na gestão de desempenho, importante na perspetiva do consumidor que faz *hosting* de serviços em nós de uma infraestrutura onde o desempenho é variável. Caso seja utilizada uma *cloud* pública, a variabilidade do desempenho e disponibilidade constituem dificuldades. Podem ser ultrapassadas com a análise de métricas e consequente tomada de decisões como, por exemplo, alojar o serviço em várias *clouds*, determinando o ambiente de execução conforme o desempenho medido, garantindo assim alta disponibilidade.

Diferentes topologias podem ser utilizadas nos sistemas de monitorização, como se pode observar na figura 2.4.
Na topologia centralizada, os *collection agents* (CAs) ou sondas de monitorização, comunicam diretamente com o *root agent* (RA) que implementa a lógica, enviando-lhe os dados de monitorização coletados. Caracteriza-se por ser fácil de gerir, mas não ser escalável.
Na topologia em camadas (hierárquica), os CAs transmitem os dados coletados para os *federation agents* (FAs), que processam e guardam os dados, que por sua vez comunicam com o RA ou vice-versa. Caracteriza-se por ser escalável.
A topologia *peer-to-peer* (P2P) consiste numa abordagem descentralizada onde cada agente pode desempenhar qualquer um dos cargos de CA ou FA. É tolerante a falhas, mas adiciona complexidade ao sistema.
Finalmente, a topologia híbrida consiste na combinação da topologia em camadas (organização hierárquica) com a P2P. Herda as vantagens de ambas as topologias [33, 52, 56].

2.4.1 Características da monitorização

As características principais que os sistemas de monitorização devem possuir são [1, 22, 47]:

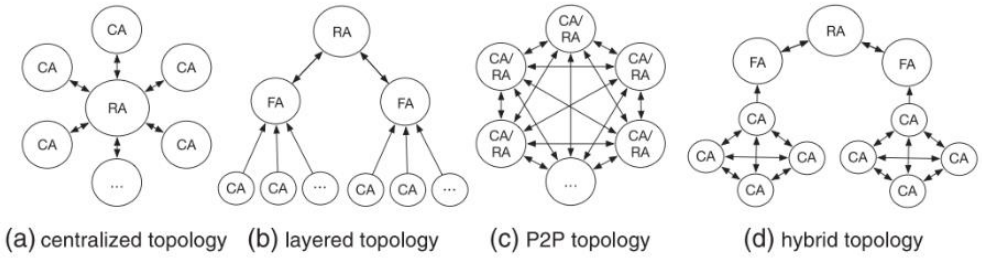

Figura 2.4: Topologias dos sistemas de monitorização [56]

- **Escalabilidade:** A infraestrutura da *cloud* pode ser constituída por muitos nós e consequentemente, sondas que geram grande quantidade de dados. O sistema deve então ser capaz de continuar a monitorizar (coletar, transferir e analisar) de forma eficiente, face ao aumento do volume de dados. De forma a reduzir o mesmo, podem-se usar técnicas de agregação que combinam várias métricas numa só (por exemplo, usando métricas das várias camadas ou usando algoritmos de *machine learning*) ou de filtragem, que evitam a propagação de dados não relevantes.

- **Elasticidade:** Lidar com infraestrutura dinâmica. De forma a alterar os recursos consoante a necessidade, é essencial a monitorização correta que acompanhe a criação e destruição de recursos virtuais. Para gerir a virtualização, podem-se usar, por exemplo, *active checks* (execução remota de comunicação do tipo *pull*) e *passive checks* (comunicação do tipo *push* onde os *hosts* físicos notificam acerca das VMs que estão a ser executadas).

- **Adaptabilidade:** Adaptar-se à carga computacional e da rede da *cloud* de forma a não se tornar invasivo, ou seja, que a carga computacional gerada pelas medições ativas, em conjunto com a coleta, processamento, transmissão e armazenamento de dados, não afete as operações normais da *cloud*. O desafio passa por manter o *trade-off* entre precisão e invasão.

- **Pontualidade:** A resposta a eventos, como por exemplo, o provisionamento de recursos, ou a migração de serviços, deve ser dada no tempo apropriado. Caso contrário, a existência de monitorização seria desnecessária. A existência de eventos complexos, que exigem a agregação de informação de diferentes nós remotos, constituem dificuldades em garantir respostas adequadas.

- **Autonomia:** Dado os recursos serem providenciados num modelo a pedido "*on-demand*", é importante que o sistema consiga reagir a mudanças, falhas ou degradação de serviço, de forma automática. Dada a escala da infraestrutura da *cloud*, conseguir esta gestão automática face a mudanças exige elasticidade e pontualidade no processamento de dados e na propagação das respostas aos atuadores. Para obter autonomia, são necessárias capacidades de análise e a definição de políticas que determinam os comportamentos. As camadas constituem um obstáculo.

- **Extensibilidade:** Com a evolução da *cloud* e dado o uso de novas tecnologias, as ferramentas de monitorização devem ser extensíveis para se adaptarem a novos ambientes. Um sistema de monitorização pode ser compreensivo se suporta ambos os tipos de recursos (físicos ou virtuais), vários tipos de dados de monitorização e vários tipos de utilizador. Caso não suporte, mas seja possível adicionar módulos para tal, o sistema caracteriza-se por ser extensível (característica atingível a partir de desenhos modulares). É intrusivo caso requeira a modificação da *cloud*, o que não é desejável.

- **Robusto:** Para não comprometer as atividades que beneficiam do sistema de monitorização, este deve ser resiliente, confiável e disponível, de forma continuar a operar mesmo na presença de falhas, a executar as funções e a servir os pedidos ininterruptamente dos utilizadores.

- **Precisão:** Deve ser preciso, isto é, que o valor medido seja o mais próximo da realidade, para garantir que as atividades resultado do processamento dos dados de monitorização se justifiquem. Também, na ausência de precisão, podem ser obtidas conclusões erradas na deteção de problemas.

- **Interoperabilidade:** Capacidade para partilhar informação de monitorização entre componentes da *cloud* heterogéneos para fins cooperativos.

- **Manter histórico:** Ter a capacidade de manter um histórico de dados. Com recurso a este, é possível identificar, por exemplo, a origem de um determinado problema a longo prazo.

2.4.2 Desafios e dificuldades

Dada a complexidade da infraestrutura da *cloud*, os sistemas de monitorização ainda enfrentam vários desafios [1, 22, 26, 52]:

- **Eficiência:** Com o aumento dos recursos e utilizadores da *cloud*, é necessário aumentar a eficiência de algoritmos e técnicas na gestão dos dados de monitorização. As operações do sistema de monitorização de filtrar, agregar, correlacionar, armazenar e outras, devem respeitar certos requisitos de forma aos recursos computacionais e de comunicação necessários não interferirem com o desempenho da *cloud*.

- **Camadas da *cloud*:** As camadas usadas para promover separação e modularidade constituem limitações na análise de dados e das respetivas ações, resultado do processamento dos mesmos. As camadas impedem acesso às camadas inferiores por parte dos utilizadores, bem como das camadas superiores por parte dos fornecedores.

- **Custos e consumo de energia:** Reduzir o consumo de energia e custos computacionais e de comunicação, satisfazendo as propriedades desejadas. Adicionalmente,

devem ser usadas determinadas métricas para otimizar o consumo de energia nos centros de processamento de dados. Por exemplo, os dados relacionados com a temperatura podem ser usados de forma a aumentar o tempo de vida do *hardware*.

- **Cobrança de serviços:** Há necessidade, por parte dos fornecedores, de mudarem o modelo de cobrança de serviços, de taxa fixa onde os serviços são cobrados por hora ou mês, por um modelo baseado no uso. O desafio consiste em obter um conjunto de métricas para estas novas políticas de faturação. A coleta de métricas para vários usos pode resultar em duplicados.

- **Escalabilidade:** Desafio em lidar com a quantidade de dados coletados e enviados para a base de dados de monitorização, provenientes dos recursos físicos que contêm VMs com agentes de monitorização integrados, coletando possivelmente múltiplas métricas.

- **Problemas nativos da *cloud*:** O facto da rede da *cloud* ser dinâmica e as VMs e *containers* serem iniciados e terminados de forma rápida, exige que o sistema de monitorização seja robusto, *agent-less* e que torne possível a monitorização de VMs.

2.5 Monitorização de aplicações baseadas em micro-serviços *deployed* na *edge*

Dada a escala e complexidade da infraestrutura onde os micro-serviços são *deployed* e executados de forma independente em diferentes *hosts*, o paradigma da computação *edge* apresenta vários obstáculos à monitorização nestes ambientes.
Uma vez que a computação *edge* abrange os recursos computacionais desde os dispositivos terminais até à *cloud* (*gateways*, *routers*, *switches*, centros de processamento) e que são heterogéneos entre si, faz com que a deteção da origem da degradação de serviço, a gestão do sistema e análise do desempenho, sejam tarefas difíceis. Requer então um sistema capaz de executar de forma automática as principais atividades de monitorização: coleta de informação de monitorização, gestão de dados (armazenamento e alojamento dos dados), processamento dos dados e disseminação e apresentação dos mesmos [51].

Cada uma das atividades do desenho de um sistema de monitorização é representado por um ou mais modelos de decisão. Estas decisões resultam em soluções com características e preocupações diferentes e por consequência, diferentes implicações nos sistemas em que atuam. A implementação de cada modelo, conforme a decisão, necessitará de componentes específicos [27].

1. **Geração de dados:** A decisão recai sobre o tipo de dados a ser coletados: dados relativos à infraestrutura onde os serviços são executados (e.g. *hosts* disponíveis, consumo de memória por *host*); métricas especificas ao serviço (e.g. *throughput*, tempo de resposta, consumo de memória por serviço) ou métricas de interações de

serviços (e.g. tempo de comunicação entre serviços).
Para analisar a infraestrutura, pode-se implementar um agente em cada *host* de um micro-serviço, que deteta novas instâncias quando são criadas mas que não oferece informação sobre o ambiente nem serviços em tempo de execução; mais específico, instalar agentes em cada plataforma de execução de serviços (e.g. Tomcat), podendo existir mais do que um agente por *host*; ou, instalar um agente por serviço coletando métricas do serviço mas não fornecendo informação sobre o ambiente onde o mesmo é executado.

2. **Gestão de dados:** Decisões acerca de como guardar, o tipo de alojamento (*hosting*) e a estratégia de distribuição dos dados. Relativamente a guardar os dados, pode-se optar por armazenamento centralizado, onde os dados são guardados num só sítio. Tem como vantagens o facto de necessitar de menor gestão e a facilidade no armazenamento de dados, apesar de ser mais suscetível a *bottlenecks*, constituindo um único ponto de falha. Na abordagem descentralizada, o armazenamento dos dados é distribuído pelos vários *hosts*, e tem como desvantagens a possibilidade de perda de dados com a indisponibilidade de um armazenamento local, apresentando, contudo, maior escalabilidade que a opção centralizada.
   Quanto ao *host* dos dados, a decisão recai em *self-host* ou em SaaS. No primeiro, os dados são *hosted* pela empresa que os detém, garantindo segurança, mas acarretando com as responsabilidades de escalamento. Relativamente ao SaaS, a empresa usa os serviços de um fornecedor de serviços *cloud*: apesar de não haver a preocupação acerca da infraestrutura, o *delay* introduzido no envio dos dados e o tráfego gerado podem ser prejudiciais.
   Na distribuição dos dados, há requisitos de existência de agregação local dos dados, de forma a reduzir o tráfego da rede, e de análise de monitorização em tempo real. Os tipos de comunicação possíveis são *pull* e *push*. *Pull*, onde o monitor recolhe a informação de um armazenamento local e guarda-a num armazenamento central, possibilitando agregação prévia, mas correndo o risco de perda de informação caso haja um *crash* no *host*. *Push*, onde os agentes enviam os dados diretamente para um armazenamento central. Apesar de possibilitar a análise em tempo real, o desenho do sistema tem de ser obrigatoriamente centralizado e não permite agregação.

3. **Processamento de dados:** Para o processamento de dados, devem ser feitas escolhas para a captura das interações dos serviços, o nível de agregação dos dados armazenados e na definição dos *thresholds*.
   Para endereçar a informação sobre as interações, pode-se recorrer a *tracing* a vários níveis, dependendo do grau de modificação permitido pelos componentes monitorizados (e.g. usando *timestamps*) ou *call monitoring* para fazer *logging* de interações, disponibilizando apenas informação acerca do número de chamadas realizadas e recebidas do serviço.
   Para guardar os dados, pode-se optar por guardar na forma nativa, ocupando mais

espaço, ou guardar de forma agregada, processando os dados de forma a reduzir o espaço de armazenamento ocupado, mas perdendo detalhes que podem ser essenciais, por exemplo, na análise da causa origem.
Para o *threshold*, pode-se optar por *thresholds* estáticos que são configurados manualmente, ou *thresholds* calculados, valores aprendidos pelo sistema de monitorização com base em dados mais antigos.

4. **Disseminação e apresentação dos dados:** Relativamente à disseminação de dados a um utilizador, existem escolhas para como o utilizador acede aos dados (e.g. *web*, aplicação móvel, aplicação *desktop*), se a informação pode ser adaptada consoante o utilizador e as suas necessidades e se a informação de diferentes sistemas *back-end* pode ser integrada (um sistema de monitorização é menos reutilizável, contudo, o uso de vários aumenta a complexidade).
   Finalmente, os dados apresentados podem ser respeitantes a métricas específicas aos serviços; uma apresentação de uma *IT-landscape* que disponibiliza relatórios de dados de monitorização de infraestrutura (e.g. *hosts* e centros de processamento disponíveis); métricas relacionadas com a infraestrutura (e.g. o uso de CPU de um *host*); mapa dos serviços que estão a ser executados que dá uma visão geral dos serviços que estão a ser executados e das suas interações; ou as chamadas feitas a um serviço, que permite descobrir a causa de falhas.

2.5.1 Caso de estudo - Prometheus

O Prometheus [45] é um sistema de monitorização que colige métricas, avalia expressões, apresenta resultados e que dispõe de mecanismos de alertas. As suas características principais são: (1) possuir um modelo de dados em que os dados *time series* são identificados pelo nome da métrica e pares chave/valor (e.g. api.server.http.requests.totalmethod= ”POST”,status=”500”, instance=”<sample1>”); (2) possuir uma linguagem de consulta PromQL para suportar o modelo de dados; (3) usa o tipo de comunicação *pull* para coletar dados via HTTP; (4) nós que são monitorizados (*targets*) são descobertos através de mecanismos de *service discovery* ou através de configuração; (5) nós autónomos sem dependências em armazenamento distribuído.
O sistema adapta-se tanto para a monitorização de arquiteturas centralizadas como para arquiteturas orientadas a serviços altamente dinâmicas. Revela-se, contudo, não viável para soluções que requeiram precisão máxima (e.g. soluções de faturação), já que o sistema pode preferir perder uma pequena quantidade de dados para continuar a operar e a apresentar as estatísticas disponíveis. A arquitetura do Prometheus pode ser vista na figura 2.5. Os principais componentes são:

- **Prometheus *server***: Colige métricas de um ou mais *targets* ou do Pushgateway de forma periódica, armazenando-as de seguida numa base de dados *time series* (TSDB). Contém algoritmos de agregação;

- **Client-libraries**: Bibliotecas em várias linguagens para expor métricas internas via HTTP;
- ***Exporters***: Monitorizam sistemas *third-party* de modo a converter as métricas existentes no modelo de dados compatível com o Prometheus;
- **Pushgateway**: Suporta *"short-lived-jobs"*. Estes *jobs* utilizam comunicação do tipo *push* para comunicar as suas métricas. As métricas são *cached* e expostas ao Prometheus *server*;
- **Alertmanager**: Lida com os alertas despoletados pelo Prometheus *server*, notificando aplicações clientes. É responsável por encaminhar os mesmos para plataformas de notificações *third-party* com o qual está integrado (e.g. *email*, PagerDuty, OpsGenis). Implementa os mecanismos de:
  - **Agrupamento**: Capacidade de categorizar os alertas recebidos que resulta no envio de um menor número de notificações. Como exemplo, assume-se que vários serviços reportam um alerta de *"connection timed out"* relativamente à base de dados que partilham. Na ausência deste mecanismo, o número de alertas despoletados seria equivalente ao número de serviços existentes e consequentemente ao número de notificações que o Alertmanager reencaminhava para o cliente. Contudo, o envio de várias notificações iguais nada acrescenta de relevante comparativamente ao envio de uma única notificação. Na perspetiva do utilizador, também é benéfico receber apenas o número de notificações necessário por questões de legibilidade;
  - **Inibição**: Mecanismo para suspender notificações de alertas perante a ocorrência de outro tipo de alertas. Por exemplo, é possível configurar para que, após ser enviada uma notificação referente a um alerta que indicava que um serviço estava inacessível, suspender o envio de notificações que tenham origem em outros alertas a respeito do mesmo;
  - **Desativação**: Mecanismo para desativar o envio de notificações temporariamente.

O Graphite[4] constitui outra solução para monitorização mais simples [3]. As principais diferenças nas várias atividades são:

- **Na coleta de dados**: O Graphite não coleta dados. Para tal, usa soluções integradas como por exemplo fluentd ou collectd. Estes últimos sistemas são *daemons* que reúnem métricas de várias fontes, armazenando-as ou disponibilizando-as através da rede. O Prometheus apresenta uma solução mais completa compreendendo mecanismos de coleção, armazenamento, visualização e exportação.

[4] https://graphiteapp.org/

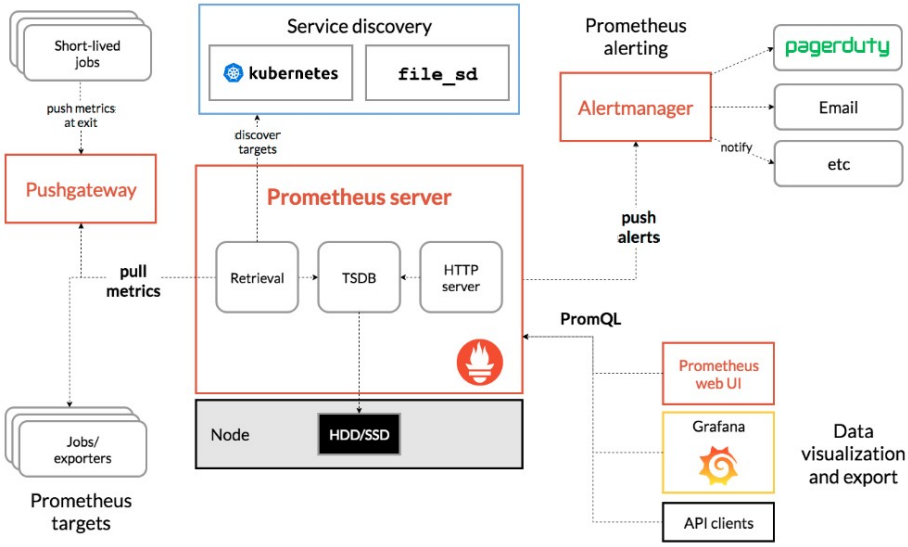

Figura 2.5: Arquitetura do Prometheus [45]

- **No modelo de dados**: De forma a simplificar os mecanismos de filtragem, agrupamento, e *matching*, o Prometheus codifica as dimensões através de pares chave/valor. Os nomes das métricas no Graphite são componentes separados por pontos que codificam as dimensões (e.g. stats.api.server.tracks.post.500). Adicionalmente, enquanto que o Prometheus preserva as instâncias como uma dimensão, o Graphite agrega os dados das várias instâncias. Por exemplo, se houver mais do que uma instância do servidor e o objetivo for medir o número de pedidos às instâncias do *api-server* com o código "500", o Graphite irá juntar os resultados de cada instância, enquanto que o Prometheus expõe os resultados para cada uma delas, facilitando assim a rápida deteção de instâncias com problemas.

- **No armazenamento:** No Graphite, os dados *time series* são armazenados em ficheiros diferentes num disco local e vão sendo substituídos pelos mais recentes. O Prometheus possibilita configurar o tempo em que os dados persistem no disco local, onde os novos dados são anexados aos existentes.

- **No alarme e rastreamento de eventos:** O Graphite não suporta diretamente alertas ao contrário do Prometheus. O Graphite consegue rastrear eventos, enquanto que o Prometheus oferece liberdade para implementar o rastreamento de eventos, através da linguagem PromQL.

2.5.2 Caso de estudo - FogMon

O FogMon [6] é uma ferramenta de monitorização distribuída e leve criada para ser executada em infraestruturas heterogéneas entre a *cloud* e a *edge*. As motivações para o desenvolvimento deste trabalho foram essencialmente o crescimento do número das

aplicações de IoT e, consequentemente, o aumento da quantidade de dados que estas introduzem na rede. Por um lado, as soluções que tiram proveito apenas da *cloud* dispõe de recursos ilimitados, contudo, apresentam resultados de latência elevada e congestão na rede. Em contraste, as soluções orientadas apenas para a perspetiva da *edge*, acontece que os nós podem não satisfazer os requisitos da aplicação dada a instabilidade que apresentam em termos de recursos e falhas.

Esta ferramenta é capaz de obter e agregar dados relacionados com o *hardware* dos nós que alojam o sistema (e.g. CPU, memória, disco), de QoS, de rede (e.g. latência e largura de banda entre os nós do sistema) e informação acerca dos dispositivos IoT ligados.

Foi escrita em C++ e apresenta uma arquitetura de duas camadas *peer-to-peer*. A figura 2.6 apresenta um exemplo de topologia do FogMon. Dispõe também de protocolos *gossip* para tolerância de falhas, tanto de rede como dos nós. Este protocolo garante a disseminação dos dados para todos os membros de um grupo.

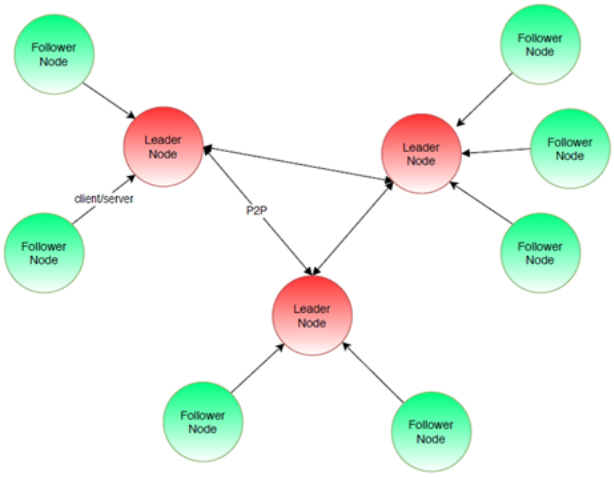

Figura 2.6: Exemplo de topologia do FogMon [6]

A ferramenta é constituída por dois tipos de agentes de *software*, que ditam o tipo de nós do sistema: *followers* ou *leaders*. Os nós *followers* estão associados a um único *leader* e são responsáveis por coletar a informação de monitorização. Os *leaders*, para além de monitorizarem o seu próprio nó, agregam, periodicamente, os dados que os *followers* recolheram. Os *leaders* estão organizados numa rede *overlay peer-to-peer* e partilham, entre si, os dados dos respetivos *followers* através do protocolo de comunicação *gossip*.

O FogMon utiliza a ferramenta Hyperic Sigar[5] para coletar informação sobre o *hardware*, o comando `ping` para medir a latência ponto-a-ponto entre os nós do grupo, as ferramentas Iperf[6] e Assolo[7] para obter informação da largura de banda e, finalmente, a biblioteca libserialport para a descoberta de dispositivos IoT.

[5] https://github.com/hyperic/sigar/
[6] https://iperf.fr/
[7] http://netlab-mn.unipv.it/assolo/

Quando se pretende juntar um novo *follower* ao ambiente monitorizado pelo FogMon, este precisa de saber o endereço de pelo menos um *leader*. Depois, este último retorna-lhe uma lista com todos os identificadores dos *leaders* disponíveis para que este possa ficar associado ao que estiver mais próximo. A partir deste momento, o novo nó inicia a atividade de monitorização. O agente de *software* é composto por duas *threads*: uma dedicada à recolha e comunicação dos dados e para receber atualizações dos *leaders* acerca de novos *followers*/*leaders*. Esta informação é, por sua vez, utilizada pela outra *thread* que monitoriza a rede e as ligações associadas aos nós do grupo.

Os *leaders* são responsáveis por receber os dados que os *followers* injetam de forma periódica. A coleta de métricas pode também ser despoletada pelos *leaders* (*pull*). Estes, periodicamente, partilham o seu relatório de dados com outro *leader* para prevenir possíveis perdas de dados. São também encarregues de eliminar os *followers* que não enviem os dados de monitorização dentro do período definido no mecanismo de *heartbeat*.

2.5.3 Caso de estudo - FMonE

O FMonE [46] constitui outra ferramenta de monitorização que implementa as funcionalidades necessárias que suportam as características dos dispositivos em infraestruturas *fog*. As motivações para o desenvolvimento deste trabalho foram, também, o facto da maioria das soluções existentes não ter em consideração os desafios e características que os dispositivos deste tipo de infraestruturas apresentam, nomeadamente:

- **Heterogeneidade**: Máquinas que compreendem o sistema dispõem de recursos diferentes;
- **Mobilidade**: Nós não são estáticos;
- **Conectividade**: Conectividade pode não ser contínua dada a mobilidade dos nós da *edge*;
- **Distribuição geográfica**: Nós podem estar separados por distâncias significativas.

De seguida, apresentam-se os requisitos considerados essenciais para o sistema:

- **Não necessidade de instalação**: Heterogeneidade dos nós dificulta o processo de instalação de dependências;
- **Agregação e filtragem de métricas**: Dados específicos e de grão-fino são mantidos na *edge* para análise enquanto que a informação agregada é enviada para a *cloud* para propósitos de visualização, por exemplo. Utilizador deve poder definir critérios para dados que possam ser excluídos;
- ***Back-end* flexível**: Dada a instabilidade dos nós, os dados devem ser armazenados nos nós considerados mais seguros, sendo que esta gestão deve ser flexível. Também

devem ser configuráveis os parâmetros do volume e velocidade da atividade de monitorização, dependendo do número de nós e do que se está a monitorizar;

- **Elasticidade**: Deve conseguir detetar a entrada e saída de nós na infraestrutura. Perante o primeiro caso, deve começar a monitorizar os mesmos e respetivos componentes o mais rápido possível;

- **Resiliência**: Visto que o ambiente está sujeito a falhas que podem levar à inatividade de agentes de monitorização, o sistema deve ser capaz de reiniciá-los automaticamente;

- **Atento à localização**: As escolhas de *deployment* dos agentes devem ser otimizadas tendo em conta a localização dos mesmos e dos respetivos *back-ends*. Assim, evitam-se latências desnecessárias no processo de monitorização. Adicionalmente, este requisito pode ser usado para obter informação mais detalhada acerca de uma dada região;

- **Monitorização à base de *plugins***: Os agentes monitores devem ter *plugins* que possibilitam a extração de diferentes métricas perante a especificação do tipo de componente a ser monitorizado (e.g. Docker *container*, MongoDB);

- **Não intrusivo**: A atividade de monitorização não deve interferir com o funcionamento normal das aplicações;

- **Agnóstico no *hardware* e sistema operativo**: Sistema deve ser capaz de ser executado em diferentes tecnologias independentemente do *hardware* e sistema operativo.

2.5.3.1 Arquitetura

A arquitetura está representada na figura 2.7. As partes constituintes, que visam cumprir os requisitos são:

- **FMonE** *Framework* que coordena toda a atividade de monitorização na infraestrutura. Está escrita em Python e comunica com o Marathon, orquestrador de *containers* que gere os agentes de monitorização e os *back-ends*, ambos executados em Docker *containers*. Para além do Marathon, são também executados, através do sistema DC/OS[8], o Docker Engine e o Apache Mesos em cada um dos nós para facilitar o *deployment* dos componentes. DC/OS foi usado pela facilidade de *deployment* e pelas funcionalidades favoráveis aos requisitos: alta disponibilidade, restrições de *deployment* e deteção de novos nós;

- **Pipeline** *Workflow* que é definido pelo utilizador sobre a disposição dos agentes e *back-ends* na infraestrutura. Este *workflow* é depois traduzido para um ficheiro

[8] https://dcos.io/

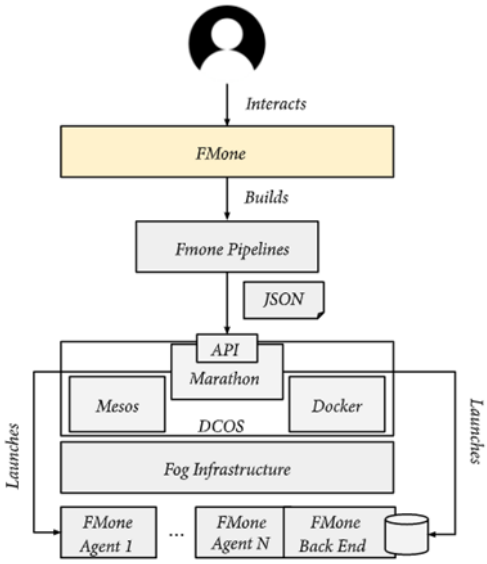

Figura 2.7: Arquitetura geral que suporta o FMonE [46]

JSON e enviado para o Marathon. Este é responsável por ler o mesmo e inicializar os *containers* nas localizações especificadas. Para além dos agentes e *back-ends*, o utilizador pode introduzir diferentes tipos de *plugins* que ativam a agregação ou recolha de algum tipo de métricas, por exemplo. Cada Pipeline pode estar associado a uma ou várias regiões;

- ***Back-end*** Componentes que armazenam as métricas de um ou mais agentes;
- **Agent FMonE** Responsável pela monitorização dos dispositivos. Este processo envolve três atividades: coletar, filtrar e publicar as métricas. O utilizador define os *plugins* usados para cada uma destas. Primeiro, o *Inplugin* que extrai as métricas dos componentes de forma periódica. Depois, o *Midplugin* que é usado para filtrar e agregar o conjunto de métricas que guarda em memória. Finalmente, o *Outplugin*, que transfere os dados filtrados para os *back-ends*.

A figura 2.8 mostra um exemplo de instanciação do sistema. Este está a ser usado para monitorizar três regiões usando dois *back-ends*. Na região B e C são executados *plugins* de agregação de métricas. Os utilizadores podem consultar as métricas armazenadas em qualquer um dos *back-ends*.

2.6 Virtualização

Antigamente, dada a lacuna de tecnologias que possibilitassem a execução de várias aplicações no mesmo servidor de forma segura, sempre que era necessário o *deployment* de uma nova aplicação, era também necessário comprar um servidor que a alojasse. Adicionalmente e também devido à falta de conhecimento acerca dos recursos computacionais

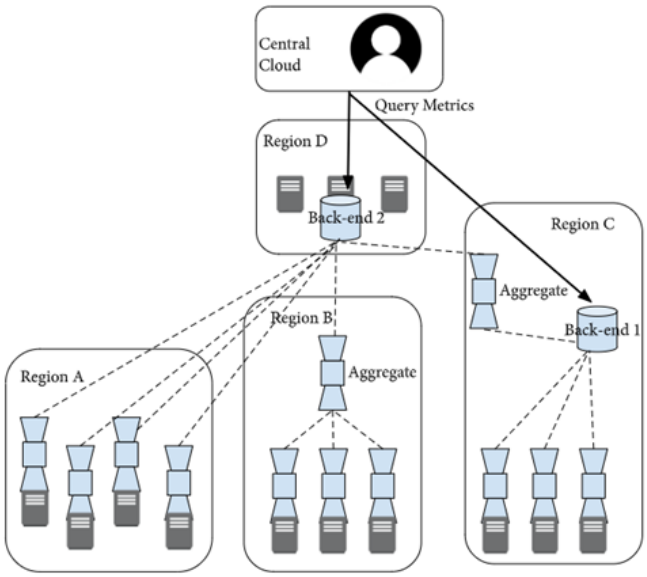

Figura 2.8: Exemplo de instanciação do FMonE [46]

necessários para o correto funcionamento da aplicação, os servidores comprados disponibilizavam, na maior parte das vezes, recursos com capacidade superior à que era utilizada. Presenciava-se então um desaproveitamento da infraestrutura e gastos de capital superiores aos necessários. De forma a resolver este problema, surgiram as máquinas virtuais, que consistem num tipo de virtualização [44].

Virtualização consiste no método de criar objetos virtuais de *software* ou *hardware* (e.g. sistema operativo, recursos, servidor) a partir de um único sistema de *hardware* físico. As partições possíveis de criar no disco rígido são um exemplo de virtualização lógica usada para facilitar a gestão dos dados.

A possibilidade de executar uma instância virtual de um servidor (VM) numa camada abstraída do *hardware* do *host*, soluciona o problema de ineficiência do uso de um servidor por aplicação. Da perspetiva das aplicações que são executadas em cada VM, o ambiente é igual a um servidor físico onde o sistema operativo, bibliotecas e programas são únicos e independentes ao sistema operativo do *host* [38]. As propriedades essenciais da virtualização são [55]:

Particionamento

- Execução de diversos sistemas operativos numa máquina física;
- Divisão de recursos do sistema entre máquinas virtuais.

Isolamento

- Fornecimento de isolamento de falhas e segurança no nível do *hardware*;
- Preservação do desempenho com controles avançados de recursos.

Encapsulamento

- Gravação do estado integral da máquina virtual em arquivos;
- Facilidade para mover e copiar máquinas virtuais (tão fácil quanto mover e copiar arquivos).

Independência de *hardware*

- Aprovisionamento ou migração de qualquer máquina virtual para qualquer servidor físico.

Na perspetiva do utilizador, executar aplicações construídas para diferentes sistemas operativos na mesma máquina física, e, na perspetiva do provedor de serviços, poder gerir as aplicações permitindo que o servidor seja usado de forma mais eficiente, são os motivos principais que fazem com que a virtualização seja usada.

O *software* responsável por gerir e partilhar o sistema base, criar e executar máquinas virtuais seguras e independentes entre si é denominado *Hypervisor* ou *Virtual Machine Monitor* (VMM). Liga-se diretamente ao *hardware* do *host* e tem como responsabilidades gerir e distribuir os recursos disponíveis pelas máquinas virtuais. Estas últimas instanciam um sistema operativo convidado que reside na *sandbox* da VM [30].

Os *hypervisors* podem ser classificados em vários tipos [10, 38]:

- **Tradicional**: São executados diretamente no *hardware* do *host*. Adequados para performance. A sua arquitetura está representada na figura 2.9 (a).
- **Hospedado**: Executados no sistema operativo do *host* tal como qualquer outro programa. São facilmente instalados e são capazes de usar componentes do sistema operativo base (e.g. scheduler, pager, I/O drives). A sua arquitetura está representada na figura 2.9 (c).
- **Híbrido**: Executado paralelamente ao sistema operativo base. A sua arquitetura está representada na figura 2.9 (b).

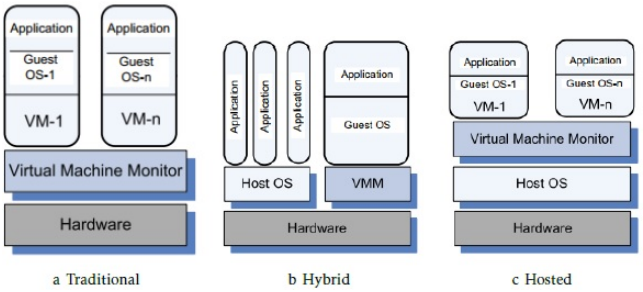

Figura 2.9: Tipos de VMMs [39]

O uso de VMs como forma de virtualização colmata o problema descrito acima de ineficiência ao possibilitar a execução de várias aplicações num único servidor. Contudo, cada VM necessita de um sistema operativo dedicado, que acarreta o consumo de CPU,

Random Access Memory (RAM) e armazenamento que poderia ser consumido pelas aplicações. Adicionalmente, a velocidade de inicialização, portabilidade ou necessidade de licenças, constituem outras das limitações desta tecnologia que levaram à criação de novos paradigmas. Surgiram assim os *containers*.

2.6.1 *Containers*

Um *container* consiste numa unidade de *software* que empacota uma aplicação com tudo o que é necessário para a sua execução, de forma a torná-la isolada do sistema operativo do *host* que aloja o *container*. Ao contrário do que acontece com as VMs, onde o sistema operativo convidado (*guest*) é executado por cima do sistema operativo da máquina que as aloja com acesso virtualizado ao *hardware*, todos os *containers* presentes num *host* partilham o mesmo e único sistema operativo. Necessitam apenas dos componentes necessários para a aplicação executar de forma fiável e isolada em diferentes ambientes de computação: código, *runtime*, ferramentas do sistema, bibliotecas, dependências, entre outros (como ilustrado na figura 2.10) [41, 44].

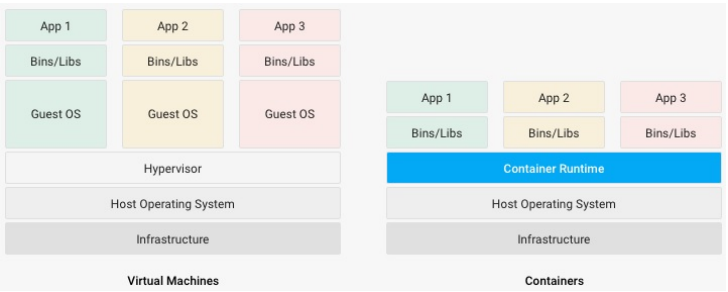

Figura 2.10: Diferenças na arquitetura das VMs e dos *containers* [9]

Os *containers* usam o comando chroot do Linux para isolar e partilhar o sistema de ficheiros entre os ambientes virtuais, e as funções do kernel cgroups para gerir e isolar os recursos de CPU, entre outros, e namespaces para a atribuição dos identificadores dos processos, artefactos de rede, *mountpoints* e outros [19].

Uma vez não ser necessária a utilização de um sistema operativo próprio no suporte aos *containers*, existe menor impacto sobre os recursos do sistema, tais como CPU, RAM e armazenamento. Reduz também a sobrecarga induzida pelas atualizações e manutenções do sistema operativo. O facto de todas as dependências estarem empacotadas, torna possível a abstração entre a aplicação e o ambiente onde realmente está a ser executada, que resulta num desacoplamento entre a aplicação e o sistema operativo da máquina, proporcionando alta portabilidade e consistência [9]. A tabela 2.1 sintetiza as diferenças entre as máquinas virtuais e os *containers* em várias áreas.

Numa arquitetura de micro-serviços, é comum o uso de um *container* por micro-serviço do sistema. Deste modo, a gestão e *deployment* de cada micro-serviço tornam-se independentes.

Tabela 2.1: *Containers* vs VMs [19]

Parâmetro	**VMs**	***Containers***
Sistema operativo	Cada VM tem o seu próprio sistema operativo e recursos de *hardware* virtuais isolados do *host*.	Todos os *containers* partilham o mesmo sistema operativo e kernel.
Comunicação	Feita através de dispositivos *Ethernet*.	Mecanismos *standard* IPC (*pipes*, *sockets*, entre outros).
Segurança	Sistema operativo constitui um ponto de ataque. Está isolado do sistema operativo do *host*.	Dado que os *containers* partilham o mesmo kernel, é possível que um *container* afete negativamente outro ao comprometer a estabilidade do kernel.
Performance	*Overhead* introduzido na tradução das instruções da máquina do sistema operativo da VM para o sistema operativo do *host*.	Performance equivalente à performance nativa do sistema operativo do *host*.
Isolamento	Garante isolamento total.	Dados podem ser partilhados entre *container* e servidor através de *mounts*.
Tempo de inicialização	Pode levar minutos a iniciar. Adicionados os tempos que envolve o processo de arranque do kernel que compreende *locating*, *decompressing*, e *initializing*, bem como o tempo necessário para a inicialização de *hardware* com o kernel *bootstrap*.	Demoram maioritariamente segundos a iniciar. O tempo de arranque é unicamente dependente do tempo que demora a aplicação a iniciar.
Armazenamento	VMs consomem muito mais armazenamento pois todo o kernel do sistema operativo e programas associados têm que ser instalados.	Ocupam menos espaço já que o sistema operativo é partilhado.
Custos	Custos de licenciamento e administração. VMs raramente usam todos os recursos ao seu dispor, resultando numa solução ineficiente que se traduz em desperdício de infraestrutura.	Um único sistema operativo suporta vários *containers*, o que reduz os custos de licenciamento.
Escalabilidade e portabilidade	Pouca escalabilidade dado o tamanho (GB) e os recursos consumidos da máquina onde são alojadas. Aplicação e sistema operativo estão sempre acoplados pelo que a migração da aplicação exige consequentemente a migração do sistema operativo.	Solução leve (poucos MB) e rápida de iniciar facilitam o *deployment* em grande escala (centenas ou milhares de aplicações por máquina).

2.6.2 Docker

A ferramenta Docker [28, 41, 44] consiste num *software* responsável por criar, gerir e orquestrar os *containers*. Tanto pode ser executado em ambientes Linux ou Windows. Para a execução de *containers*, o Docker usa o containerd, *daemon* com suporte para Linux e Windows, visto como um supervisor de *containers* que controla as operações durante o ciclo de vida dos mesmos. Para a criação de *containers*, é usado o runc, *runtime* que comunica com o kernel do sistema operativo para reunir as condições necessárias da criação de um *container*.

Quando se faz referência ao Docker que está a ser usado para executar e orquestrar os *containers* de uma dada máquina, refere-se mais propriamente ao Docker Engine. Apenas é possível fazer o *deployment* de um Docker *container* numa máquina que o tenha instalado. A sua arquitetura divide-se em dois componentes principais: o Docker Daemon que expõe uma API RESTful e o Docker Client que faz pedidos à mesma, agindo como intermediário entre o cliente e a API. O cliente consegue obter informação e dar instruções ao Docker Daemon enquanto que o Docker Daemon é responsável pelas imagens e *containers*. O Daemon é quem obtém as imagens tanto do Docker Registry, serviço que guarda imagens,

como de *registries* públicos que existam na *Internet* (e.g. Docker Hub) através de pedidos HTTP (como é representado na figura 2.11).

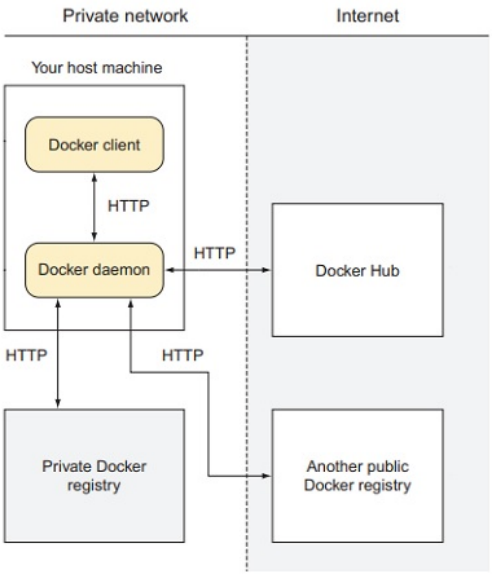

Figura 2.11: Arquitetura e comunicações [41]

A configuração padrão dita que a API disponibilizada do Docker Daemon é acessível através do *socket* do domínio "docker.sock", contudo, é também possível expor a API via *sockets Transmission Control Protocol* (TCP).

A instância de uma imagem é denominada de *container* quando executada no Docker Engine. Uma imagem é construída a partir de um conjunto de camadas *read-only*, em que cada camada representa um comando usado na construção de uma imagem Docker. Estes comandos são escritos num Dockerfile (documento de texto com todas as instruções, exemplificado na listagem 2.1, que quando compilado, resulta na imagem. Dentro da imagem está um sistema operativo reduzido, todos os ficheiros e dependências necessárias para executar a aplicação [14, 21].

Listagem 2.1: Exemplo de Dockerfile

```
1 FROM ubuntu:15.04
2
3 COPY . /app
4
5 RUN make /app
6
7 CMD python /app/app.py
```

Exemplo de Dockerfile. O comando FROM indica a imagem ubuntu, COPY copia ficheiros da diretoria do cliente para a diretoria no *container*, RUN constrói a aplicação usando o comando make e CMD especifica o comando a executar dentro do *container*.

2.6.2.1 Docker *services*

No contexto de uma aplicação de larga escala baseada em micro-serviços, um sistema é normalmente constituído por diferentes serviços, cada um com as suas responsabilidades (e.g. serviço da base de dados, serviço de *back-end*, serviço de *front-end*, entre outros). Neste âmbito, um serviço é um *container* em produção e constitui a imagem para um micro-serviço [15, 44]. Na criação de um serviço, é especificado a imagem do *container* a usar e um conjunto de opções, nomeadamente:

- Porta para tornar o serviço disponível publicamente;
- Rede à qual o serviço se deve juntar;
- Limites de CPU e memória RAM;
- Número de réplicas (*tasks*) que devem ser criadas;
- Políticas de reiniciação.

De forma a definir, executar e aumentar o nível de escala dos serviços, o Docker disponibiliza a opção de criar um ficheiro YAML onde é detalhado o comportamento dos *containers* em produção. Em qualquer altura em *runtime*, é possível reconfigurar as instâncias (e.g. aumentar o número de réplicas, alterar o limite no uso dos recursos) sem suspender a execução do(s) *container*(s) em questão nem interromper a atividade dos restantes da *stack* [16].

2.6.2.2 Docker Swarm

O Docker facilita a escalabilidade de aplicações disponibilizando a possibilidade de criar um *cluster* de nós virtuais ou físicos candidatos para alojar *containers* de serviços. Deste modo, em vez de se adicionarem mais recursos a uma máquina para esta ser capaz de alojar novas réplicas, adicionam-se mais máquinas ao sistema.

Um swarm é um conjunto/*cluster* de nós, podendo, cada um destes, ser do tipo *manager* ou *worker*. Os nós *managers* do *swarm* controlam a configuração e estado do mesmo, sendo os únicos capazes de criar e distribuir *containers* pelos *workers*, bem como executar comandos Docker específicos ao *cluster*. Os nós *workers* têm como único propósito executar *containers* [41].

Características e vantagens principais do Docker Swarm [17, 21]

- **Integrado no Docker Engine**: O Docker Swarm está integrado no Docker Engine, não sendo necessário qualquer *software* de orquestração adicional.
- **Escalabilidade**: O *swarm manager* lança ou remove *tasks*, automaticamente, conforme o número de *tasks* discriminadas no ficheiro de configuração, adaptando-se ao estado desejado.

- **Conformidade com estado desejado**: Caso o *swarm manager* presencie diferenças entre o estado atual do *swarm* e o estado desejado do mesmo, atua de forma a mantê-lo consistente. Por exemplo, quando é declarado que um serviço é composto por oito réplicas e duas falham ao longo tempo, o *swarm manager* vai voltar a executar dois *containers* nos nós disponíveis. Outro exemplo é quando os serviços são executados apenas em nós que tenham uma dada etiqueta: caso o nó deixe de ficar etiquetado, o *swarm manager* é responsável por remover as *tasks* ou vice-versa, automaticamente.

- **Rede *multi-host***: O Docker Swarm atribui, automaticamente, endereços aos *containers* de uma rede especificada *a priori* ou, caso contrário, à rede ingress criada quando é iniciado o *swarm* ou quando um nó se junta a um *swarm* previamente existente (explicado na secção 2.6.2.3).

- ***Service Discovery***: O *swarm* tem um servidor de *Domain Name System* (DNS) incorporado. Assim, dentro da rede do *swarm*, é possível que os *containers* comuniquem através do nome do serviço.

- ***Load balancing***: É possível expor as portas de um serviço e usar um mecanismo de balanceamento de carga externo. Relativamente à distribuição de *containers* pelos nós do *swarm*, o Docker Swarm oferece a possibilidade de configuração manual. Isto é, é possível usar restrições e etiquetas para controlar o posicionamento dos *containers* (e.g. nome da máquina, máquina que têm uma determinada etiqueta, entre outros). Quando não existem estas restrições, o *swarm manager* tenta distribuir os *containers* de forma a obter resiliência máxima, ou seja, se um serviço for constituído por mais do que uma réplica, o *swarm manager* irá posicioná-las em diferentes nós, caso os recursos sejam suficientes para suportar os *containers*.

- **Seguro**: Docker Swarm usa *Transport Layer Security* (TLS) para mecanismos de autenticação, autorização e encriptação, de forma a estabelecer as comunicações seguras entre *containers* no *swarm*.

2.6.2.3 Docker *network*

O Docker cria e configura a *interface* docker0 do sistema *host* dentro do kernel do Linux como uma *Ethernet bridge*. Esta pode ser usada tanto para a comunicação entre *containers* presentes no mesmo *host*, como para expor o serviço publicamente através do mapeamento de portas entre *container* e *host*. O Docker escolhe um endereço e uma sub-rede do intervalo privado definido pelo RFC 1918 que não é usado pelo *host* e atribui ao docker0. Todos os *containers* estão ligados ao docker0 por configuração padrão (figura 2.12). É possível criar uma rede *bridge* customizada e juntar apenas alguns *containers* à mesma, possibilitando a comunicação entre eles e isolamento para outros que existam. A figura 2.13 mostra uma rede *bridge* criada, denominada "localnet", que pode ser especificada no momento quando um *container* é executado [35, 44].

Contudo, a comunicação entre *containers* que estão em *hosts* diferentes obriga a publicar as portas de cada *container* e criar um mecanismo de *routing* entre eles, pelo que, para além de adicionar complexidade e não ser uma solução escalável (visto que dois *containers* não podem usar a mesma porta), as portas ficam expostas para o mundo, constituindo possíveis pontos de ataque [44].

O Docker oferece como solução as redes *overlay*. São redes internas privadas que permitem ligar diferentes nós com o Docker Engine instalado e fazer com que os *containers* que executam nos mesmos comuniquem entre si. É desta forma que os *swarm services* interagem de forma segura. Quando é criado um serviço e este não é ligado a uma rede *overlay* existente, este liga-se à rede *overlay* que é criada no Docker *host* quando o mesmo inicializa ou se junta a um *swarm* (rede "ingress"). O Docker trata também do *routing* dos pacotes desde o Docker Engine origem até ao Docker Engine destino. Para isto, o Docker usa túneis VXLAN de forma a criar redes *overlay* de camada 2 virtuais [18, 44]. Como exemplo, tem-se a figura 2.14, onde ambos os *containers* pertencentes a cada nó têm um endereço *Internet Protocol* (IP) na rede *overlay*.

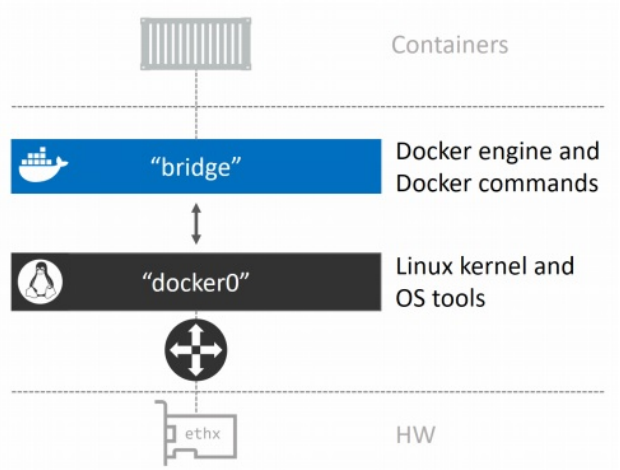

Figura 2.12: Configuração *bridge* [44]

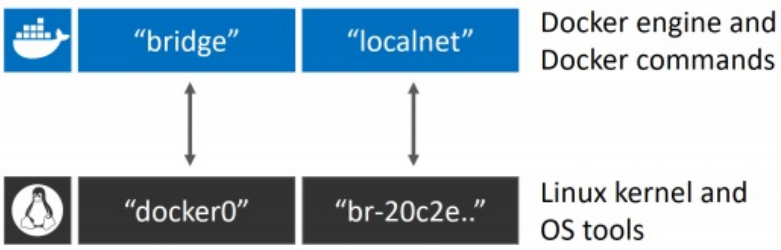

Figura 2.13: Configuração *bridge* customizada [44]

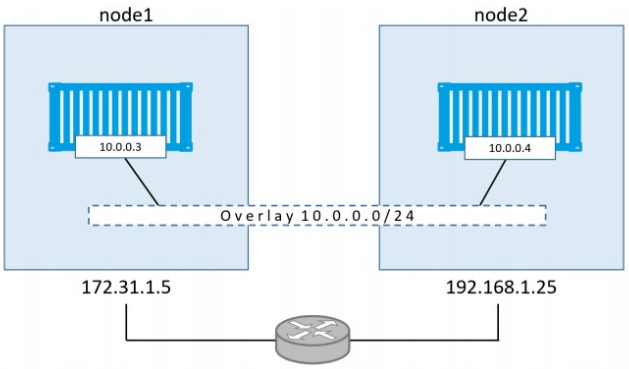

Figura 2.14: Rede *overlay* entre *containers* de diferentes nós [44]

Capítulo 3

Solução proposta

O ambiente de execução do sistema de monitorização engloba as camadas de *cloud*, *edge* e *devices*. O posicionamento dos diferentes componentes definiu-se com base nas capacidades computacionais que cada nó disponibiliza. Enquanto que a computação *cloud* permite guardar e processar grandes quantidades de dados, pode-se revelar ineficiente num ambiente que usa nós da *edge*, dispersos e heterogéneos, onde a escalabilidade e o tempo de reação são fatores importantes a considerar.

Assim, com os objetivos de não sobrecarregar monitores na *cloud* (neste caso o Prometheus), reduzir o tráfego na rede e gerar alertas em tempo quase real, foi concebido, num trabalho desenvolvido no Departamento de Informática FCT/UNL [32], um monitor mais leve para ser *deployed* em nós da *edge*.

O desempenho das aplicações que executam na *edge* é altamente variável e dependente de várias condições tais como qualidade de conexão da rede ou número de pedidos a serem processados. Esta variabilidade é aumentada na presença de um sistema elástico, inerente ao uso de micro-serviços *deployed* em *containers* que são migrados/replicados de acordo com as necessidades locais inesperadas ou previsíveis. Houve assim a necessidade de estender o sistema de monitorização base para suportar o contexto da computação *edge*, garantindo o uso eficiente da infraestrutura de forma escalável.

3.1 Trabalho anterior

O trabalho anterior teve como principal objetivo a criação de um sistema de monitorização com uma arquitetura que permitisse tirar partido do *continuum* entre a *cloud* e a *edge* para realizar a atividade de monitorização em contextos de micro-serviços.

A origem do problema deveu-se ao facto das soluções existentes na altura serem direcionadas para ambientes *cloud* e, por conseguinte, não haver limites/preocupações na

utilização dos recursos consumidos pela atividade de monitorização na infraestrutura. Adicionalmente, a solução objetivava resolver os problemas que uma solução centralizada na *cloud* apresenta, principalmente, o congestionamento da rede para conseguir providenciar alertas em tempo real.

Para tal, foi criada a ferramenta de monitorização EdgeMon. Consiste numa solução leve do sistema existente Prometheus. A arquitetura está representada na figura 3.1, que demonstra a possibilidade de haver uma federação de monitores onde o Prometheus, alojado na *cloud*, ocupa o nível mais alto. O componente desenvolvido reúne as seguintes funcionalidades:

- **Armazenamento de métricas**: Pushgateway do Prometheus incorporada, utilizada como *cache*, que guarda e disponibiliza as últimas métricas recolhidas. Possibilita também a injeção de métricas por parte dos alvos de monitorização;
- **Recolha de métricas**: Reúne métricas a nível do *host*/nó com recurso ao Node Exporter incorporado;
- **Avaliação de regras**: Avaliação de regras estáticas com base em métricas. A avaliação ocorre ou quando chegam novas métricas ou periodicamente, sendo que neste último caso o intervalo de tempo entre avaliações é configurável pelo utilizador;
- **Envio de alertas**: Caso as regras sejam ativadas, ou seja, se o valor da métrica avaliada não se encontrar dentro do intervalo esperado, são gerados alertas e enviados para o Alerts Pushgateway presente na *cloud*.

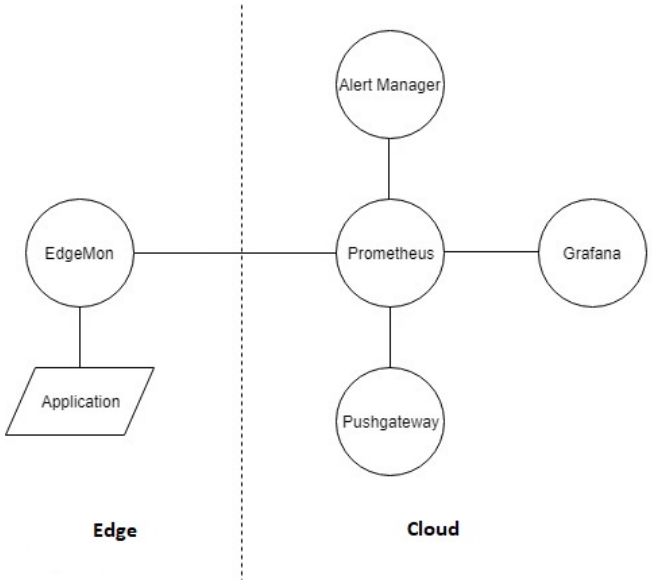

Figura 3.1: Arquitetura e comunicações do sistema de monitorização

3.1.1 Limitações

As limitações do sistema, face aos objetivos que esta dissertação propõe resolver, foram principalmente identificadas nas seguintes áreas:

- **Suporte ao dinamismo**: Não suporta o dinamismo que existe tanto a nível da infraestrutura como a nível dos micro-serviços que monitoriza. Falta a existência de um ou vários componentes responsáveis por mecanismos que suportem a resolução automática de problemas quando, por exemplo, um monitor ou a infraestrutura se encontram sobrecarregados (e.g. replicação e particionamento). Relativamente ao suporte à arquitetura de micro-serviços, o monitor visa apenas monitorizar o nó onde é executado. Logo, sempre que se quer monitorizar um determinado nó, tem de ser criado um novo monitor neste. Apesar do monitor ser leve, esta obrigatoriedade de acoplamento monitor-alvo de monitorização pode, por vezes, não constituir a solução mais eficiente no uso dos recursos da infraestrutura, já que não se justifica a alocação de recursos para criar um *container* que aloja um monitor, dado que a carga/trabalho à qual este último fica sujeito ser muito reduzida. Adicionalmente, não oferece opções de adaptabilidade que se revelam essenciais nestes ambientes, como por exemplo, a migração de monitores em resposta ao movimento dos micro-serviços;

- **Descoberta de serviços**: Não existe noção de autonomia a nível da deteção do aparecimento/desaparecimento dos serviços alvos de monitorização. A gestão de monitores é sempre realizada de forma manual;

- **Oferta de informação**: As métricas recolhidas e consequentemente disponibilizadas são apenas a nível do nó, que o Node Exporter oferece;

- **Parametrização da atividade de monitorização**: Parâmetros que influenciam o comportamento da atividade de monitorização não oferecem a configurabilidade desejada. Por exemplo, avaliar regras em momentos diferentes consoante a regra. Da mesma forma, as regras são estáticas pelo que não é possível inserir novas, alterar as configurações das existentes ou remover;

- **Monitorização dos monitores**: A carga dos monitores não é monitorizada. Esta limitação impossibilita observar o estado do monitor e consequentemente otimizar o sistema de monitorização. Eventos indesejados como falhas ou degradação da qualidade de serviço são impossíveis de prever.

3.2 Requisitos propostos para o novo sistema de monitorização

Para atingir os objetivos de um sistema de monitorização que suporte ambientes com propriedades dinâmicas, o sistema deve atender ao seguinte conjunto de requisitos:

- **Descoberta de serviços**: O sistema deve ser capaz de descobrir novos serviços e de notar o desaparecimento de outros. Num ambiente de micro-serviços seja na *cloud* ou *edge*, manter ficheiros de configuração de forma manual torna-se inexequível uma vez que o tempo de vida dos micro-serviços pode, por vezes, ser medido em segundos ou minutos. Na arquitetura de micro-serviços, as localizações dos micro-serviços estão constantemente a mudar, seja por motivos de reiniciação, falha, escalabilidade, migração, entre outros e portanto, é essencial que a reconfiguração aconteça "*on the fly*", para que quando um novo serviço seja criado ou reposicionado, o resto do sistema seja capaz de comunicar com ele;

- **Auto-gestão**: O sistema deve ser autónomo e auto-adaptativo. Deve providenciar o correto funcionamento da atividade de monitorização e adaptar-se às condições de cada instante. O sistema deve optar por tomar as decisões que maximizem o seu desempenho, adotando técnicas por exemplo de geo-localização para reduzir a latência e de otimização na utilização de recursos para evitar sobrecarga de nós ou componentes. Consequentemente, a QoS aumenta.

- **Aquisição de métricas e armazenamento**: Deve ser possível obter estatísticas dos serviços alvos de monitorização tanto na camada da *edge* como da *cloud*. O sistema deve ser capaz de reunir a maior quantidade de informação que faculte uma imagem acerca do estado do serviço e do ambiente que o aloja (métricas de baixo e alto nível).

  Adicionalmente ao *polling*, onde o processo de obtenção de métricas é despoletado pelo sistema de monitorização, deve também ser possível aceitar métricas transmitidas por iniciativa dos micro-serviços.

  É importante que as métricas possam ser armazenadas nas duas camadas. O armazenamento na *edge* deve ser temporário, recorrendo-se ao uso de *caches*, enquanto que na *cloud* pode ser persistente, dados os recursos disponíveis. Os clientes devem poder obter as métricas em ambos os casos.

- **Processamento de métricas**: O sistema deve possibilitar a existência de funções de agregação que permitem extrair novas métricas a partir das métricas armazenadas, tendo em vista a diminuição do volume de dados a ser transferido. A média, soma, máximo ou mínimo são alguns exemplos de funções.

  O sistema deve também suportar diferentes formatos de métricas. Devem existir mecanismos de compatibilidade para permitir a comunicação entre componentes.

- **Regras e alertas**: Os monitores são responsáveis por avaliar as regras e despoletar alertas para as entidades clientes. Para tal, é necessária a integração deste componente com um motor de regras. Este último pode ser mais ou menos complexo dependendo dos recursos disponíveis.

  A periodicidade e definição das regras deve ser configurável e a ativação das regras deve despoletar o envio de alertas para o componente especificado.

- **Interface**: Disponibilizar *endpoints* para que possam ser definidas configurações por parte do cliente do sistema de monitorização: intervalo entre recolhas, intervalo entre avaliações de métricas, entre outros. Adicionalmente, deve disponibilizar *endpoints* para o cliente poder manipular o sistema diretamente: criar/remover monitores, migrar monitores, atribuir serviços alvos de monitorização a um determinado monitor, entre outros.

- **Visualização**: Módulo para visualizar as métricas armazenadas de forma persistente na *cloud*. Disponibilizar as métricas em tempo real através de gráficos constitui uma das prioridades para o cliente. A análise de métricas de diversas fontes a partir de uma *interface* intuitiva permite, às empresas, compreender como otimizar processos internos, identificar a necessidade de intervenções e tornar a tomada de decisões mais rápida e eficiente.

3.3 Arquitetura geral e seus componentes

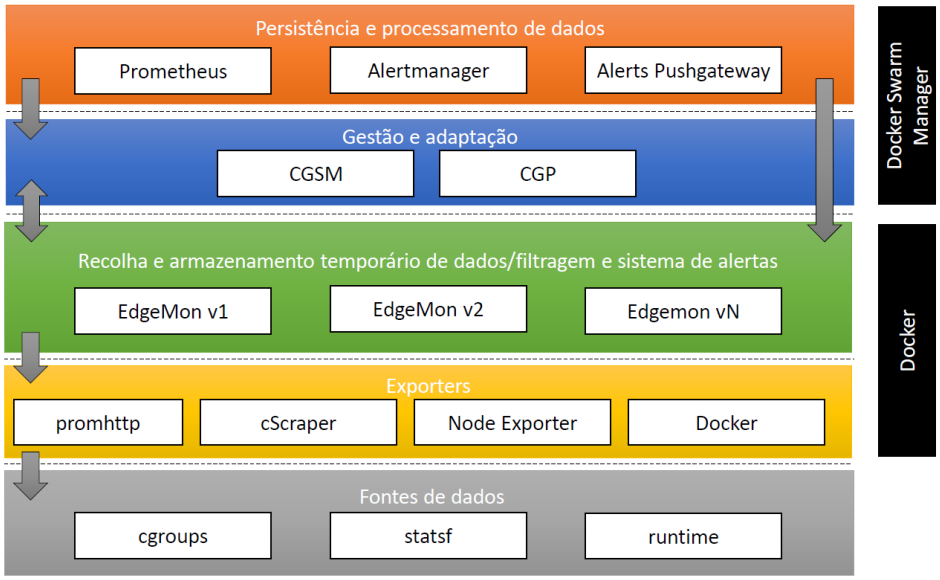

Figura 3.2: Arquitetura do sistema em camadas

Os componentes constituintes da arquitetura do sistema de monitorização são: EdgeMon, Gestor do sistema de monitorização (CGSM), Gestor do Prometheus (CGP), Prometheus, Pushgateway, Alertmanager, cScraper e Node Exporter. Note-se que o trabalho desenvolvido nesta dissertação recai sobre o aprimoramento e **extensão do EdgeMon e na criação do CGSM, CGP e cScraper**. Todos os outros componentes são externos a esta dissertação.

As responsabilidades e funcionalidades de cada um dos componentes descrevem-se a seguir. Os componentes EdgeMon e CGSM são também apresentados com mais detalhe

nas secções 3.5 e 3.6, respetivamente, e a comunicação/interação entre os componentes é explicada na secção 3.7 e ilustrada na figura 3.8.

CGSM Componente responsável pela gestão do sistema de monitorização e descoberta de serviços no contexto de um *cluster*. As suas operações são:

1. Descobrir serviços relevantes para a atividade de monitorização dentro do *cluster* onde está posicionado (e.g. EdgeMons, alvos de monitorização, *exporters* de métricas, entre outros);
2. Preparar o *cluster* para a atividade de monitorização por parte dos EdgeMons (*deployment* de *exporters*);
3. Atribuição otimizada de alvos de monitorização a EdgeMons;
4. Monitorizar os EdgeMons existentes no *cluster* (medição das suas cargas);
5. Gerir o ciclo de vida dos EdgeMons no *cluster* (criar, remover ou migrar instâncias);
6. Registar EdgeMons externos ao *cluster*.

CGP Componente responsável por gerir os alvos de monitorização do Prometheus no contexto do *cluster* onde o mesmo se encontra. Encarregue de:

1. Notar o aparecimento/desaparecimento de serviços e atualizar a lista de *targets* do Prometheus;
2. Adicionar serviços externos ao *cluster* à lista de *targets* do Prometheus.

EdgeMon Componente com responsabilidade principal de coletar e armazenar métricas de micro-serviços (alvos de monitorização ou *scrape targets*) e de si próprio. Responsável por avaliar regras parametrizáveis com base nas mesmas e despoletar alertas para as camadas superiores (CGSM e Alerts Pushgateway). É gerido por um único CGSM e pode constituir um alvo de monitorização na perspetiva do Prometheus. As suas operações são:

1. Coletar métricas dos alvos de monitorização;
2. Expor métricas dos alvos de monitorização;
3. Registar regras;
4. Avaliar regras com base em métricas;
5. Auto-monitorização;
6. Envio de alertas para os componentes superiores.

Prometheus Monitor *deployed* na *cloud* com recursos computacionais vistos como ilimitados. Responsável pela coleta, processamento e armazenamento de métricas. Oferece:

1. Monitorização de alvos de monitorização;
2. Agregação de métricas de forma a obter novas métricas através da configuração de regras;
3. Armazenamento de métricas de forma persistente;
4. Avaliação de regras;
5. Ponto de entrada para injeção de métricas (Pushgateway).

Alerts Pushgateway Pushgateway *deployed* na *cloud* para receber alertas provenientes da *edge*. Os alertas gerados e enviados pelos EdgeMons têm o formato de métricas aceite pelo Prometheus. Logo, caso se pretenda, é possível tornar este componente num alvo de monitorização do Prometheus, de maneira a que este volte a avaliar e que os alertas agora gerados sejam encaminhados para o Alertmanager.

Alertmanager Constitui o componente responsável por lidar com os alertas despoletados pelo Prometheus, detalhado na subsecção 2.5.1.

cScraper Componente responsável por gerar e disponibilizar métricas acerca do uso de recursos e de características de performance dos *containers* executados no mesmo *host*.

Node Exporter Componente responsável por gerar e disponibilizar métricas a nível do *host*.

3.4 Arquitetura de um sistema de monitorização com propriedades adaptativas

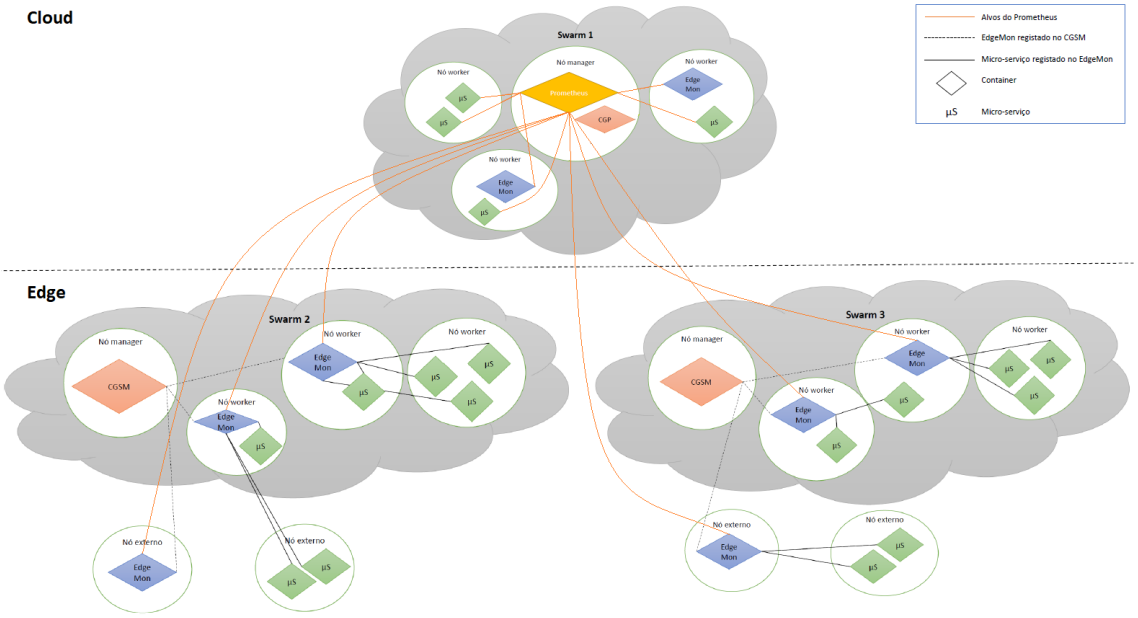

Figura 3.3: Exemplo de instanciação da arquitetura

Dado o grande número e dispersão dos nós na *edge*, que alojam micro-serviços, optou-se por organizar o sistema em *clusters* (*swarms* na perspetiva do Docker), cada um com os componentes necessários para realizar a atividade de monitorização de forma independente. Tal como se pode ver pela figura 3.3, o sistema é constituído por um número variável de *clusters*, cada um contendo um CGSM ou CGP e um número indefinido de micro-serviços e EdgeMons. O *deployment* dos micro-serviços é controlado pelo componente de gestão de micro-serviços (externo a esta dissertação). Isto implica que o *deployment* do sistema de monitorização deve ser feito tendo em conta o posicionamento dos micro-serviços, recorrendo a mecanismos de geo-localização para tal. Isto é, os alvos de monitorização devem ser monitorizados pelos monitores que se encontram mais próximos a eles. Desta forma, reduz-se a latência entre comunicações e aumenta-se a eficiência do sistema. Caso estes monitores locais fiquem sobrecarregados, cabe à solução adaptativa criar novos monitores nos locais mais adequados (e.g. tendo em conta a carga dos nós nas proximidades). A arquitetura é hierárquica possibilitando propriedades de escalabilidade, suportando assim o possível crescimento de nós no ambiente de monitorização. Como se pode ver pela figura 3.4, é possível ter uma federação de servidores Prometheus, em conjunto com EdgeMons, onde uns monitores constituem alvos de monitorização na perspetiva de outros. Na direção das folhas, estão os nós com menor capacidade computacional mas que estão geograficamente mais perto dos alvos de interesse. Na direção da raiz, estão os monitores complexos posicionados em nós mais ricos e distantes dos alvos. Desta maneira, apesar dos *clusters* serem independentes entre si, é possível agregar a informação obtida por cada um deles a partir das camadas superiores. Esta arquitetura, para além de resolver problemas de sobrecarga e diminuir a latência, possibilita coletar dados com diferentes níveis de detalhe (*instance-level drill-down* ou dados agregados) e permite que uma instância Prometheus recolha apenas dados específicos de outra, proporcionando assim o mecanismo de alertas e de consulta de dois conjuntos de dados num único servidor.

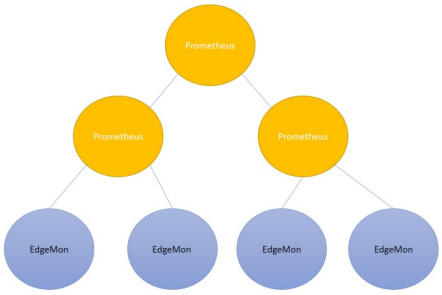

Figura 3.4: Exemplo de hierarquia possível

Qualquer tipo de serviços pode ser *deployed* em qualquer tipo de nó (*manager* ou *worker*), excepto o CGSM e o CGP que têm de estar obrigatoriamente acoplados ao *manager*. Esta obrigatoriedade deve-se ao facto do *swarm manager* ser o único que pode executar comandos que englobam o *deployment* e gestão dos serviços no *swarm*. Os componentes

cScraper e Node Exporter, que têm como únicos objetivos a recolha e disponibilização de métricas, são *deployed* nos nós onde não existam EdgeMons, já que conseguem obter o mesmo tipo de métricas.

Assim que um *swarm*/*cluster* é criado, é criada uma rede do tipo *overlay* (explicada na subsecção 2.6.2.3) usada pelos serviços, onde os *containers* expõem as suas portas para os outros *containers* dentro do *swarm*, permitindo a comunicação de forma segura através do seu endereço IP. A arquitetura tira assim partido das funcionalidades do gestor de *containers* Docker, embora possa ser adaptável a outro tipo de gestor de *containers*. De igual modo, tira partido do *service discovery* do Docker Swarm através do uso de DNS para os serviços executados no contexto de um *swarm*, providenciando a resolução de nomes para os *containers* de uma rede *overlay*. A comunicação entre componentes é feita através de APIs REST, uma solução típica nos dias de hoje.

3.5 Componente EdgeMon

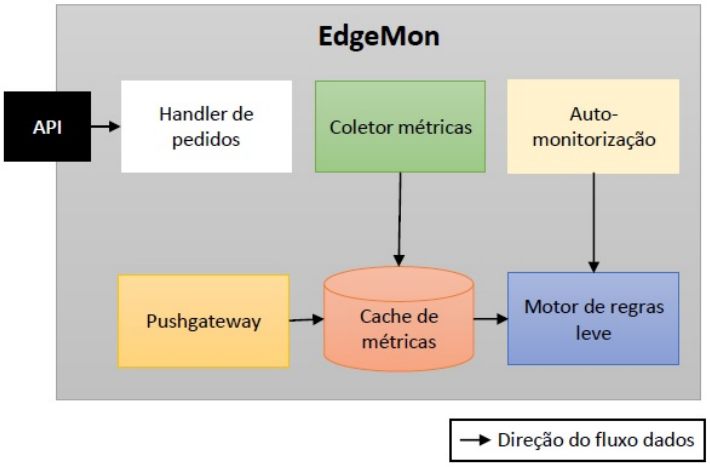

Figura 3.5: Visão detalhada do componente EdgeMon.

O monitor leve EdgeMon pode ser *deployed* em nós com poucos recursos computacionais comparativamente aos necessários para o Prometheus (verificado no teste 5.2.4). O seu principal objetivo é coletar métricas de si próprio e de alvos de monitorização que lhe são atribuídos. Para tal, os micro-serviços vão sendo registados no monitor e, periodicamente, são coletadas métricas sobre eles.

3.5.1 Tipos de métricas

As métricas podem ser de três tipos/níveis: *container*, *host* e aplicação. A recolha das métricas a nível da aplicação permite a monitorização e gestão da performance e disponibilidade das aplicações. Através destas, é possível medir a performance da aplicação, identificar formas de a otimizar e detetar a origem de problemas que surjam. Por exemplo, a métrica que corresponde ao tempo médio de resposta da aplicação aos pedidos é

um bom indicador da performance em geral da aplicação. Por outro lado, a métrica que dá informação acerca dos maiores tempos de resposta aos pedidos recebidos, é um bom revelador das partes da aplicação que podem estar a causar os picos. A métrica que indica o número de pedidos por segundo ajuda a perceber a carga à qual a aplicação está sujeita no momento. Como exemplo, a tabela A.4 (apêndice A) apresenta as métricas de nível aplicacional já disponibilizadas pela biblioteca da linguagem Go.

As métricas sobre os *containers* são igualmente importantes porque providenciam informação acerca das instâncias que partilham recursos da máquina que as aloja, de forma a poderem executar micro-serviços. A obtenção deste tipo de métricas permite otimizar a utilização dos recursos. Por exemplo, observar a quantidade de tempo que o uso de CPU é *throttled*, através da métrica "Throttled CPU Time", permite definir corretamente a partilha de CPU entre os vários *containers* no Docker. Picos deste indicador subentendem a necessidade de mais CPU, que o *host* não consegue providenciar. Já as métricas que dizem respeito ao uso de memória permitem otimizar os recursos, limitando o uso de memória por *container*, garantindo que a aplicação seja corretamente executada sem usar mais memória que aquela que necessita, não perturbando assim os outros *containers* do mesmo *host*. As métricas de rede, por sua vez, indicam o tráfego gerado, informação da qual se podem extrair diversos indicadores que ajudam a prever determinados eventos, como por exemplo, ataques de DoS. A listagem A.1, presente no apêndice A, apresenta todas as métricas que podem ser recolhidas a nível do *container*.

Por fim, as métricas a nível do *host* dão informação acerca da infraestrutura cuja performance dita consequentemente a performance das aplicações que são executadas nos *containers* nela alojados. Por exemplo, métricas acerca do consumo de CPU e memória permitem otimizar o uso de recursos nos *hosts* do gestor de *containers* (Docker). Como outro exemplo, tem-se o número total de *containers* a serem executados, que ajuda a verificar a consistência do ambiente depois de atualizações. As tabelas A.1, A.2 e A.3 (apêndice A) apresentam todos os coletores de métricas que o Node Exporter dispõe para coletar diferentes tipos de métricas a nível da máquina.

3.5.2 Recolha de métricas

O intervalo de *scraping* entre duas recolhas para um alvo de monitorização é parametrizável. Desta maneira, é possível adaptar o intervalo de maneira a cumprir os requisitos e necessidades do cliente e também perante a carga da rede nos diferentes instantes. Por exemplo, em situações onde a variabilidade do valor das métricas é insignificante para o cliente, é adequado aumentar o tempo de intervalo entre recolhas consecutivas, de forma a não sobrecarregar a rede com o tráfego induzido. Em contraste, perante situações de *stress* onde a variabilidade é grande ou de forma a diminuir o tempo de entrega de alertas, é oportuno diminuir este parâmetro. Adicionalmente, é possível escolher por micro-serviço qual o tipo de métricas a recolher, variando o parâmetro `MetricType`. Existem duas hipóteses de configuração: ou são recolhidas métricas apenas de um tipo ou de todos os

tipos.

A forma como são obtidas as métricas do tipo *container* e nó decide-se consoante a presença de EdgeMons no nó onde o micro-serviço está alojado. Caso não exista pelo menos um EdgeMon, a obtenção das métricas é feita com recurso ao cScraper e Node Exporter que recolhem e entregam os dados. Caso contrário, o pedido é redirecionado para o EdgeMon que consegue obter o mesmo tipo de métricas. Desta forma, evita-se o *deployment* dos dois *exporters* e, por conseguinte, poupa-se o uso de recursos computacionais necessários para alojar os mesmos.

Finalmente, para além das métricas obtidas através do mecanismo *pull* onde o EdgeMon faz o pedido e ao qual o serviço lhe responde com as métricas, a incorporação do Pushgateway no componente EdgeMon torna possível a injeção direta de métricas - serviço faz *push* das métricas deliberadamente. Adequa-se para serviços que não podem ser monitorizados, seja porque o seu tempo de vida é demasiado curto ou por razões de segurança (permissões). Apesar da obtenção dos dados ser diferente, todos os outros mecanismos funcionam de igual maneira para ambos.

3.5.3 Armazenamento das métricas e entrega ao nível superior

Os dados no EdgeMon são guardados numa *cache* que vai substituindo os valores de métricas com o mesmo nome. Consiste no *metrics scraper* para a camada superior. Quando o EdgeMon é *scraped*, os dados devolvidos podem ser diferentes, dependendo do modo de *scraping* ativado no momento. Na ausência de *cache* e consequente falta de persistência das métricas, seria necessário que o EdgeMon recolhesse, em resposta aos pedidos de camadas superiores, todas as métricas de todos os micro-serviços. Para além de tal aumentar significativamente o tráfego de dados na rede, introduzir-se-ia também carga sobre os EdgeMons que podia não ser suportável. Nestes casos, o cliente do sistema pode impedir que isso aconteça alterando o parâmetro `ScrapeMode` do EdgeMon. Por exemplo, caso o EdgeMon ou a rede estejam sobrecarregados, seria apropriado alterar o `ScrapeMode` para "store". Desta forma, as métricas retornadas seriam as da *cache*. Em cenários onde a precisão dos dados é prioritária, o `ScrapeMode` apropriado seria o "node" para obter as métricas mais recentes possíveis. Para conseguir obter uma visão evolutiva do sistema, o `ScrapeMode` "full" retorna as últimas métricas armazenadas na *cache* e as mais recentes.

3.5.4 Avaliação de regras e suas consequências

De forma a acelerar o processo de alarmes, os valores das métricas a observar são caracterizadas em regras que por sua vez são avaliadas num motor de regras dinâmico leve. A existência de motor de regras nas duas camadas *edge* e *cloud* é justificável com o custo. A sua existência na *edge* reduz substancialmente o tráfego na rede causado pela transmissão de todas as métricas dos serviços para a *cloud* e que seria necessária para entregar alertas em tempo quase real. A avaliação do valor das métricas na *edge* faz com que o tráfego para a *cloud* se reduza a alertas e/ou métricas que necessitem de outro tipo de avaliação (e.g.

as regras a codificar são mais complexas, podendo envolver diferentes tipos de métricas e intervalos temporais, pelo que devem ser avaliadas na *cloud*).

É possível adicionar regras de forma dinâmica, indicando o nome da regra, para que tipo de métricas se adequa, os valores das métricas que delimitam o intervalo razoável a observar, isto é, que não pode ser excedido e/ou minorado e o tipo de alerta que deve ser despoletado (exemplo na tabela 3.1). As métricas dos alvos de monitorização são avaliadas no momento em que os dados são inseridos na *cache* e quando são criadas novas regras. Adicionalmente, o EdgeMon aplica a atividade de monitorização a si próprio (auto-monitorização). Os monitores recolhem, periodicamente, métricas acerca do *container* e infraestrutura onde são executados e avaliam-nas igualmente com base em regras. Quando estas são ativadas, o monitor alerta a camada superior (CGSM) com uma notificação, o que pode levar à adaptação dinâmica do sistema de monitorização (e.g. criação de um novo monitor EdgeMon). A auto-monitorização torna assim possível a tomada de decisões por parte das camadas superiores e dentro do intervalo de tempo admissível para prevenir anomalias no sistema de monitorização (e.g. *crash* de monitores devido à sobrecarga).

Como consequência da ativação (*triggering*) das regras, é sempre enviado um alerta para as camadas superiores. As situações que é possível representar são:

- Deteção se o valor da métrica passado como argumento à regra ultrapassa os valores definidos;
- Deteção se o valor da métrica não atinge um valor mínimo definido.

Os valores associados às regras são sempre os valores efetivos, ou seja, o valor exato da métrica que foi recolhido.

Tabela 3.1: Exemplo de regra

Nome da métrica	Limite superior	Limite inferior	Tipo de métrica	Alerta
container_memory_usage_bytes	20600000	0	Container	Memory use exceeded 20MB

Para o caso de alvos de monitorização, os alertas são enviados para o Alerts Pushgateway (ver figura 3.2). No caso da métrica avaliada ser referente ao EdgeMon, o alerta é enviado para o componente da camada superior CGSM.

3.5.5 Parametrização do EdgeMon

O EdgeMon é suportado por um *container*, tal como os serviços monitorizados, pelo que faz parte de um *cluster*/grupo de *containers* gerido por um orquestrador de *containers*. No caso do trabalho proposto, descrito na seção 4, o gestor escolhido foi o Docker. O EdgeMon pode assim, por um lado, obter informação directamente do Docker sobre a execução dos *containers* no grupo (denominado *swarm* ou *cluster*, nesta tese) do qual faz parte, inclusivamente sobre ele próprio. Por outro lado, sendo um *container*, existe uma pré-configuração necessária que é necessário concretizar:

- A imagem do serviço;
- As portas para publicar o serviço, de maneira a tornar a sua API acessível a clientes que não estão ligados à rede do container (externos ao *swarm*);
- A que *swarm* se deve juntar, especificando, para isso, a rede *overlay* correspondente a este;
- Os paramêtros de entrada: valor que avisa se o EdgeMon pertence ou não a um *swarm* e o endereço para onde os alertas são enviados.

Durante a execução do EdgeMon, existe um conjunto de parâmetros que alteram o seu comportamento em várias áreas:

- Modo de *scraping*: "node", "store" ou "full";
- Tipo de métricas a recolher: "app", "container", "host" ou "all";
- Intervalo de *scraping*: período de tempo entre recolhas consecutivas de dados dos alvos de monitorização e de si próprio.

3.5.6 Limitações da infraestrutura e versões possíveis

Dado o ambiente de execução dos EdgeMons ser a *edge*, é importante notar que a capacidade computacional pode ser, por vezes, muito reduzida. De modo a adaptar a atividade de monitorização aos recursos disponíveis, vários tipos de configurações de EdgeMon podem ser feitas, resultando assim em EdgeMons mais ou menos leves.

A configuração mais simples do EdgeMon presta apenas funções de armazenamento e avaliação de métricas. Uma vez que o sistema visa diminuir a carga na rede e providenciar alertas em tempo quase real, faz sentido que o motor de regras seja uma funcionalidade base. Na figura 3.6, apresentam-se as capacidades opcionais que originam diferentes versões possíveis.

Caso não seja dito o contrário, assume-se, quando se faz referência a EdgeMons ao longo desta dissertação, que se está perante a versão mais completa.

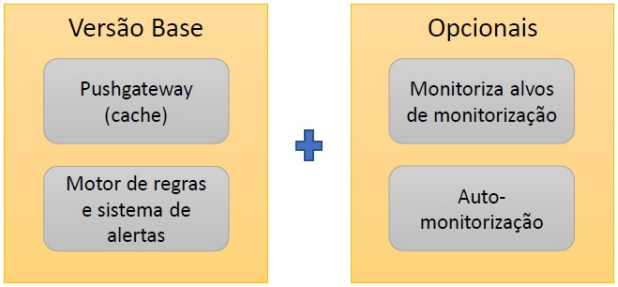

Figura 3.6: Diferentes configurações possíveis.

3.5.7 *Exporters* externos ao EdgeMon

Caso não exista pelo menos um EdgeMon no nó alvo que se deseja monitorizar, é preciso recorrer a dois serviços que ajudam a exportar as métricas quer a nível do *container* quer do *host*. Os dois *exporters* usados são:

- **cScraper**: Serviço que recolhe dados acerca dos recursos e performance dos *containers* do mesmo nó. As métricas dos *containers* (e.g. CPU, memória, entre outros) são obtidas a partir do Docker Engine que está a ser executado na máquina.
- **Node Exporter**: *Exporter* do Prometheus que recolhe e oferece métricas do *hardware* e do sistema operativo expostas pelos kernels NIX.

3.6 Componente CGSM

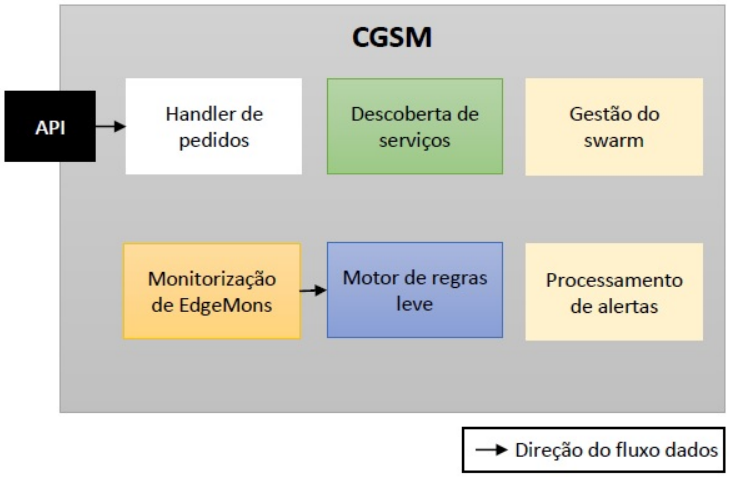

Figura 3.7: Visão detalhada do componente CGSM.

O objetivo deste componente consiste em gerir o sistema de monitorização de forma a adaptá-lo automaticamente perante diferentes tipos de cenários, procurando sempre optar pelas decisões que tornam a solução mais eficiente, proporcionando no fim, melhor QoS ao utilizador.

Note-se que a criação e desenvolvimento do *swarm*, isto é, o processo de juntar/remover nós *managers/workers* ao longo do tempo é da responsabilidade do cliente do sistema de monitorização. Este também é encarregue pelo posicionamento inicial do CGSM no nó *manager* e do(s) EdgeMon(s), caso pretenda. Caso contrário, os monitores serão criados pelo CGSM assim que existam micro-serviços para monitorizar.

Este gestor é *deployed* no nó *manager* de um *swarm* e a sua arquitetura interna está representada na figura 3.7. Os seus módulos têm como responsabilidades principais:

- Detetar novos serviços no sistema relacionados com o contexto da monitorização;

- Obter informação acerca dos serviços e nós pertencentes ao *cluster* onde está incluído;
- Fazer o *deployment* dos *exporters* nos nós sem EdgeMons;
- Gerir/monitorizar os EdgeMons com o intuito de maximizar a eficiência do sistema;
- Gerir e atribuir os micro-serviços aos EdgeMons;
- Receber os alertas provenientes da auto-monitorização dos EdgeMons;
- Reestruturar o *cluster* onde está inserido, como resultado da monitorização que realiza sobre os EdgeMons registados nele ou em resposta aos alertas que recebe destes;
- Disponibilizar API/*endpoints* para a gestão dos EdgeMons a cargo de um CGSM.

O CGP é uma simplificação deste componente, com o único objetivo de gerir a lista de alvos do Prometheus. Assim, dispõe apenas do mecanismo de descoberta de serviços. De outra forma, teria de se reescrever, manualmente, o ficheiro YAML que o Prometheus consulta para obter informação acerca da localização de cada serviço que monitoriza.

3.6.1 Descoberta de serviços e recolha de informação

Relativamente à descoberta de serviços, o componente realiza periodicamente uma pesquisa com recurso ao orquestrador de *containers* Docker Swarm da qual extrai a lista dos serviços que estão a ser executados e que, naquele momento, fazem parte do mesmo *cluster*/*swarm* que o CGSM. Desta forma, tem noção dos serviços que apareceram e desapareceram desde a última consulta e consegue assim manter sempre uma visão geral sobre o estado do *cluster* ao longo do tempo. Os serviços podem ser de vários tipos:

- **EdgeMon**: Os dados (localização, etiquetas, identificadores, entre outros) dos EdgeMons executados no *cluster* ficam guardados no CGSM responsável pelos mesmos;
- **Micro-serviço (alvo de monitorização ou *scrape target*)**: Os dados dos micro-serviços executados no *cluster* e que tenham como objetivo serem monitorizados (expresso através da utilização de etiquetas) ficam guardados no CGSM responsável pelo mesmo. Alguns destes dados são passados ao EdgeMon que fica responsável pelo micro-serviço, já que se revelam essenciais para que o mesmo possa realizar a atividade de monitorização (e.g. localização do micro-serviço e identificador do seu *container*);
- ***Exporter***: Toda a informação acerca dos Node Exporters e cScrapers executados no *cluster* fica igualmente guardada no CGSM;
- **Outro**: Qualquer outro tipo de serviço será ignorado.

Os EdgeMons executados no mesmo *cluster*/*swarm* onde está um CGSM são automaticamente registados por ele, tal como descrito. É no entanto também possível registar EdgeMons externos a um *swarm* particular, i.e. um EdgeMon que não esteja integrado em nenhum *cluster*/*swarm*. Para tal, é necessário que o *endpoint* `/register` da API do CGSM seja conhecido. Os EdgeMons devem indicar, no corpo do pedido, o seu IP e a porta onde a API está exposta.

3.6.2 *Deployment* dos *exporters*

O CGSM quando é executado, começa por preparar o ambiente necessário para a atividade de monitorização no *cluster*: cria os serviços cScraper e Node Exporter em todos os nós, à exceção dos tenham pelo menos um EdgeMon. Caso surja um EdgeMon num destes nós, os *exporters* deixam de executar automaticamente.

3.6.3 Gestão e monitorização dos EdgeMons

O CGSM tem um motor de regras tal como o EdgeMon, permitindo também a inserção de regras para gerir as instâncias EdgeMons. Desta forma, o CGSM monitoriza, recolhendo as métricas dos *containers* e *hosts* onde são executados EdgeMons, e avaliando as regras com base nestas. Existem dois momentos de avaliação: no momento de atribuição de micro-serviços, onde são utilizadas regras que indicam sobrecarga e periodicamente, como forma de controlo. Este último é importante para informar as camadas superiores acerca do estado dos monitores. Com a receção dos alertas, consegue-se prever padrões e, consequentemente, antecipar ações que suportem o funcionamento normal do sistema (e.g. criar ou migrar EdgeMons, particionar a carga), prevenindo possíveis *crashes* originados pela sobrecarga de EdgeMons ou tempos de resposta altos.

3.6.4 Gestão e atribuição de micro-serviços

O CGSM, a cada vez que obtém os serviços através do mecanismo de descoberta de serviços, verifica os micro-serviços que ainda não estão a ser monitorizados e atribui os mesmos aos EdgeMons registados, de acordo com o seguinte algoritmo. Primeiro, verifica-se se existe algum EdgeMon no nó onde o micro-serviço se encontra. Se existir e não apresentar sobrecarga, é feita a atribuição. Caso não existam EdgeMons no mesmo nó ou nenhum se revelar adequado, é atribuído ao EdgeMon menos sobrecarregado do *cluster*. No pior caso, se todos os EdgeMons do *cluster* apresentarem sobrecarga, é criada uma nova instância EdgeMon que fica responsável pelo micro-serviço. Para analisar a sobrecarga dos monitores, são avaliadas regras de sobrecarga com base nas métricas recolhidas destes.

3.6.5 Receção/processamento de alertas

Este componente tem também a responsabilidade de receber os alertas provenientes da auto-monitorização dos EdgeMons. Perante o alerta de sobrecarga e do valor da métrica

que não se encontra no intervalo apropriado, o CGSM decide o comportamento a ter:

- **Particionamento de carga**: São libertados micro-serviços do EdgeMon sobrecarregado e atribuídos a outros;
- **Replicação de EdgeMons**: É criada uma nova réplica e a carga é dividida por ambas.

3.6.6 API para a gestão não automática de EdgeMons

Complementarmente, são disponibilizados *endpoints* para a gestão de EdgeMons. Uma vez que, no futuro, o sistema de monitorização será integrado com um sistema de gestão de micro-serviços, e dado que este último gere o ciclo de vida dos mesmos, é adequado que possa ter algum controle sobre os EdgeMons. Por exemplo, se o sistema de gestão de micro-serviços migrar serviços como consequência do crescimento do número de acessos a partir de um determinado ponto, é oportuno migrar conjuntamente o EdgeMon responsável por estes ou criar um novo EdgeMon, para evitar que a latência aumente e manter ou melhorar a eficiência do sistema.

3.6.7 Parametrização do CGSM

Para fazer o *deployment* do CGSM, a pré-configuração obrigatória consiste em definir:

- A imagem do serviço;
- As portas para publicar o serviço, de maneira a tornar a sua API acessível a clientes que não estão ligados à rede do *container* (externos ao *swarm*);
- A que *swarm* se deve juntar, especificando, para isso, a rede *overlay* correspondente a este.

Durante a execução do CGSM, existe um conjunto de parâmetros que alteram o seu comportamento em várias áreas:

- Intervalo entre consultas: período de tempo entre execuções do mecanismo de descoberta de serviços no *swarm*;
- Intervalo entre avaliações: período de tempo entre avaliações consecutivas das métricas no motor de regras para uma dada regra;

3.7 Interação entre os componentes do sistema

Todos os componentes do sistema expõem uma API. O protocolo usado para a comunicação entre componentes foi HTTP. Assim, cada componente recebe pedidos HTTP que indicam a ação a ser executada para um dado recurso. De seguida, detalham-se as comunicações possíveis entre componentes no sistema e o tipo de dados que envolvem.

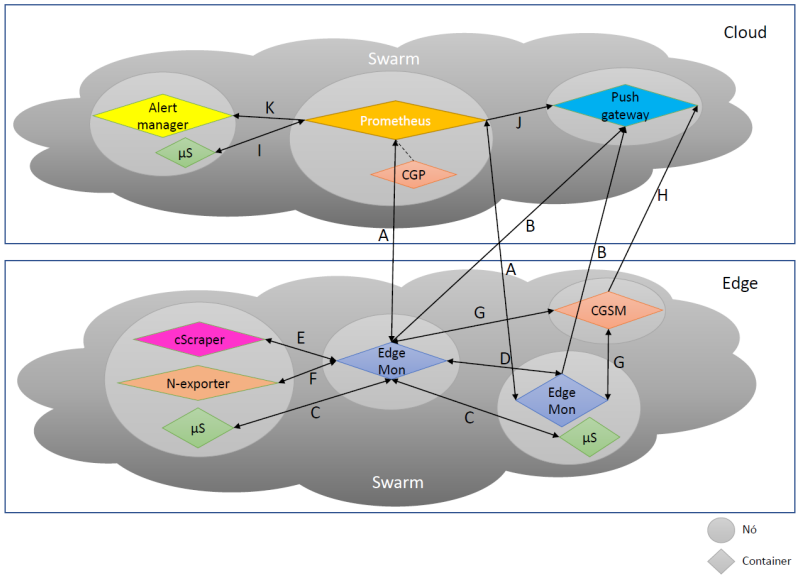

Figura 3.8: Instanciação exemplo do sistema com comunicações

- **Prometheus e EdgeMon (A)**: Relação cliente-servidor, onde o Prometheus é o cliente. O EdgeMon pode constituir um dos alvos de monitorização do Prometheus. Com isto, o Prometheus envia, periodicamente, um pedido de obtenção de métricas para o *endpoint* do EdgeMon `/metrics`. Este responde devolvendo-lhe as métricas. Estas podem ser métricas recolhidas no momento, a última versão armazenada na *cache*, ou a união das duas, dependendo do modo `ScrapeMode` configurado no EdgeMon. A periodicidade de interação é parametrizável.

- **EdgeMon e Alerts Pushgateway (B)**: Relação unidirecional onde o sentido do fluxo de dados é sempre EdgeMon-Alerts Pushgateway. Todas as regras ativadas com valores das métricas referentes a micro-serviços fazem com que o motor de regras despolete o envio do respetivo alerta para o Alerts Pushgateway.

- **EdgeMon e alvo de monitorização (C)**: Relação que pode ser unidirecional ou cliente-servidor. No primeiro caso, o alvo de monitorização injeta os seus dados diretamente no monitor (*push*). No segundo caso, o monitor interroga o alvo das suas métricas através do *endpoint* `/metrics` (*pull*). As métricas retornadas são estatísticas referentes à aplicação.

- **EdgeMon e EdgeMon (D)**: Relação cliente-servidor. Os EdgeMons que monitorizam micro-serviços alojados noutro nó que não o seu mas que contém pelo menos um EdgeMon, obtém as métricas dos *containers* e do *host* que alojam o alvo de monitorização a partir deste último através dos *endpoints* `/containermetrics` e `/nodemetrics`.

- **EdgeMon e cScraper (E)**: Relação cliente-servidor onde o monitor é cliente. O EdgeMon interroga o cScraper pelas métricas a partir do *endpoint* `/docker/containerID`.

O cScraper obtém os dados relacionados com o estado dos *containers* que alojam os alvos indicados pelo monitor e retorna-os.

- **EdgeMon e Node Exporter (F)**: Relação cliente-servidor onde o monitor é cliente. Este interroga o Node Exporter pelas métricas a partir do *endpoint* `/metrics`. O Node Exporter coleta as estatísticas acerca do *host* que aloja o alvo de monitorização e retorna-as.

- **EdgeMon e CGSM (G)**: As interações entre estes componentes são variadas: CGSM informa o EdgeMon dos alvos de monitorização que é responsável, enviando-lhe uma lista com objetos que representam micro-serviços em formato JavaScript *Object Notation* (JSON); CGSM interroga o EdgeMon acerca do seu estado atual para recuperar de falhas, retornando-lhe uma lista dos seus alvos de monitorização também em JSON; CGSM envia os objetos que representam alvos de monitorização e regras em formato JSON para o EdgeMon no momento imediatamente a seguir à migração ou replicação do mesmo; EdgeMon envia alertas para o CGSM quando as regras que indicam sobrecarga são ativadas, resultado da avaliação de métricas durante o processo de auto-monitorização.

- **CGSM e Alerts Pushgateway (H)**: Quando o CGSM não reconhece o alerta recebido de um EdgeMon, envia o mesmo para o Alerts Pushgateway.

- **Prometheus e micro-serviço (I)**: Relação que pode ser unidirecional ou cliente-servidor. No primeiro caso, o alvo de monitorização injeta os seus dados diretamente no Prometheus (*push*). No segundo caso, o Prometheus interroga periodicamente o alvo das suas métricas através do *endpoint* `/metrics` (*pull*). As métricas retornadas são estatísticas referentes à aplicação.

- **Prometheus e Alerts Pushgateway (J)**: Relação cliente-servidor onde o Prometheus é o cliente. O Prometheus recolhe, periodicamente, os alertas em forma de métricas armazenados no Alerts Pushgateway.

- **Prometheus e Alertmanager (K)**: Relação unidirecional em que o Prometheus envia os alertas gerados, resultado da avaliação de regras, para o Alertmanager.

Capítulo 4

Implementação

4.1 Tecnologias usadas

Os dois componentes estão escritos em Go. Go ou Golang foi desenvolvido pela Google em 2007. É uma linguagem estaticamente tipada e compilada, com *garbage collection* e *runtime reflection*. Apresenta-se como uma linguagem eficiente e rápida com mecanismos de concorrência que maximizam o uso de máquinas *multicore* ou interligadas pela rede. Permite também construir código flexível e modular. A escolha do Golang deveu-se essencialmente pela facilidade de integração com o Prometheus e restantes componentes existentes. Os objetivos que levaram à criação desta linguagem moldam-se às características que o sistema visa apresentar, principalmente facilidade de *deployment* e compatibilidade com ambientes distribuídos. O Golang para além de diminuir os problemas comuns relacionados com a gestão de dependências e concorrência, facilita também o *deployment* já que, tipicamente, um programa compilado é facilmente executado em servidores remotos que tenham o Go instalado, sem necessidade de bibliotecas adicionais. As tecnologias de virtualização simplificam ainda mais este processo. Foi escolhido o Docker para criar e gerir *containers* principalmente pelas características e vantagens apresentadas na secção 2.6.2.2 e pela facilidade de manipulação a partir do Go. Os componentes/serviços executam em *containers* Docker e, de maneira a criar e gerir os mesmos, e beneficiar de informações que o Docker oferece, o CGSM, CGP, EdgeMon e cScraper interagem com o Docker Engine instalado na máquina que os aloja. Para tal, foi usado o *Software Development Kit* (SDK) para Go do cliente[1] para a API do Docker Engine. A partir deste cliente, é possível executar todas as operações do Docker que equivale a executar os comandos na linha de comandos. Para tudo isto ser possível, é necessário, no lançamento de cada serviço, usar o *bind mount* do ficheiro `docker.sock` (demonstrado na listagem 4.3). Como

[1] https://godoc.org/github.com/docker/docker/client

explicado na secção 2.6.2, este constitui o *socket* Unix que o Docker Engine escuta por padrão e por onde os *endpoints* HTTP podem ser consumidos. Uma vez estando o Docker instalado na máquina que aloja o *container* onde são executadas as operações, o *bind mount* permite que o ficheiro `docker.sock` seja *mounted* dentro do *container*, possibilitando a comunicação com o Docker Engine a partir do interior do *container*.

Os Dockerfiles usados na construção das imagens dos componentes estão apresentados nas listagens 4.1 e 4.2. Ambos utilizam uma imagem base Ubuntu, declarado no comando `FROM`. Depois, através do comando `COPY`, é copiado o ficheiro executável do *host* para o *container*. De seguida, o `RUN` permite mudar as permissões e tornar o ficheiro anteriormente copiado efetivamente executável. Finalmente o comando `CMD` define o comando que é executado e respetivos argumentos quando é iniciado um *container*.

Listagem 4.1: Dockerfile do componente EdgeMon

```
1 FROM ubuntu:latest
2
3 COPY . /edgethesis
4 RUN chmod a+x /edgethesis/edgemonclient
5
6 EXPOSE 9999
7 CMD ["/edgethesis/edgemonclient", "--rules.reactive", "--swarm.environment",
8 "--alert.address=192.168.10.1:9091"]
```

Listagem 4.2: Dockerfile do componente CGSM

```
1 FROM ubuntu:latest
2
3 COPY . /compDiscovery
4 RUN chmod a+x /compDiscovery/cgsm
5
6 EXPOSE 8000
7 CMD ["/compDiscovery/cgsm", "discover"]
```

Para criar a imagem a partir dos Dockerfiles, é executado o seguinte comando:

`docker build -t <nome-imagem> <caminho para o Dockerfile>`

Para definir o estado inicial do sistema, é usado um ficheiro YAML. Na listagem 4.3, é possível ver um exemplo de instanciação do sistema. Neste caso, são definidos quatro serviços: CGSM, EdgeMon, Prometheus e Alertmanager. Para cada um deles, primeiro são definidas as imagens (cgsm:1 e edgemon:1, prom/prometheus e prom/alertmanager), depois os volumes (*socket* do Docker, ficheiro YAML com os *targets* do Prometheus, entre outros), seguido do mapeamento de portas entre o *host* e *container*, respetivamente. Por fim, declaram-se as restrições de posicionamento dos *containers*. O ficheiro contém também a referência da rede usada para a comunicação entre *containers*. Neste caso, está a ser usada uma rede *overlay* previamente criada (daí ser *external*) denominada "overlay-network-fase2".

Listagem 4.3: Ficheiro de configuração YAML

```
1
2  version: '3'
3
4  networks:
5      default:
6        external:
7          name: overlay-network-fase2
8
9  services:
10
11     CGSM:
12         image: cgsm:1
13         volumes:
14           - /var/run/docker.sock:/var/run/docker.sock
15         ports:
16           - 8000:8000
17
18     EdgeMon:
19         image: edgemon:1
20         volumes:
21           - /var/run/docker.sock:/var/run/docker.sock
22         ports:
23           - 9999:9999
24
25     Prometheus:
26         image: prom/prometheus
27         volumes:
28           - ./prometheus.yml:/etc/prometheus/prometheus.yml
29           - ./swarm-endpoints.json:/etc/swarm-endpoints/endpoints.json
30           - ./rules/alert.rules_nodes.yml:/etc/prometheus/alert.rules_nodes.yml
31           - ./rules/alert.rules_tasks.yml:/etc/prometheus/alert.rules_tasks.yml
32         command:
33           - '--config.file=/etc/prometheus/prometheus.yml'
34         ports:
35           - 9090:9090
36         deploy:
37           placement:
38             constraints:
39               - node.hostname==cloudmachine
40
41     Alertmanager:
42         image: prom/alertmanager
43         ports:
44           - 9093:9093
45         volumes:
46           - ./alertmanager.yml:/etc/alertmanager/config.yml
47         command:
48           - '--config.file=/etc/alertmanager/config.yml'
```

Tabela 4.1: Estruturas de dados em memória de cada componente

Componente	Nome	Tipo	Descrição
EdgeMon	targets	sync.Map	Armazena alvos de monitorização
EdgeMon	rules	sync.Map	Armazena regras
EdgeMon	ms	github.com/prometheus/pushgateway/storage	Armazena métricas
CGSM	emds	sync.Map	Armazena EdgeMons
CGSM	notscrapedds	sync.Map	Armazena alvos de monitorização não monitorizados
CGSM	scrapedds	sync.Map	Armazena alvos de monitorização monitorizados
CGSM	cscrapers	Map	Armazena as localizações dos cScrapers
CGSM	nexporters	Map	Armazena as localizações dos Node Exporters

Para fazer o *deployment* dos serviços com recurso a este, é executado o seguinte comando:

```
docker stack deploy -c <nome-ficheiro> <nome-stack>
```

4.2 Estado dos componentes de monitorização

Tanto o componente de gestão de sistema de monitorização (CGSM), como o componente do monitor (EdgeMon) utilizam a mesma técnica de manipulação de dados. Os componentes dependem essencialmente do armazenamento de dados em memória. Recorrem ao armazenamento em disco, por outro lado, para recuperar de falhas. A tabela 4.1 especifica quais os dados e respetivas estruturas de dados em memória presentes em cada componente.

Dadas as vantagens que existem em usar a memória e visto que os requisitos dos serviços são compatíveis com esta metodologia (principalmente acessos rápidos e frequentes aos dados), priorizou-se as leituras e escritas à RAM com recurso a estruturas de dados. A alternativa podia ser usar *in-memory databases* (e.g. H2, SQLite, db4o, entre outros) que oferecerem funcionalidades que facilitam a manipulação de grandes quantidades de dados, mas que têm sempre o custo associado no consumo de CPU e memória. O armazenamento em disco foi feito através de ficheiros JSON.

O número de escritas no disco é consideravelmente menor, acontecendo, no caso do EdgeMon, quando lhe é atribuído um novo alvo de monitorização ou quando é adicionada uma nova regra, e, no caso do CGSM, apenas quando são adicionadas novas regras. O mesmo se passa no número de leituras do disco, onde em ambos os casos, somente acontece no início de execução do processo, causado pela criação do serviço ou pela reiniciação do processo, numa situação pós-falha. Nestas situações, os EdgeMons leem os alvos de monitorização e regras que lhe foram anteriormente atribuídos e o CGSM lê as regras. O CGSM não necessita de guardar os alvos de monitorização de forma persistente dado a

periodicidade na descoberta de serviços. Eventualmente, irá recuperar o estado atual do sistema.

Desta forma, conseguiu-se melhorar a performance e guardar os dados necessários de forma persistente (combatendo assim o facto dos dados em memória serem temporários).

Num caso de *deployment* real, poderia fazer sentido usar uma *on-disk database*, devido à quantidade limitada de memória atribuída ao *container* que aloja o EdgeMon.

De maneira a iniciar os servidores HTTP, foi utilizada a função `ListenAndServe` que recebe um determinado endereço e um *handler*, da biblioteca `net/http`[2]. Esta função invoca, por sua vez, a função `Serve` para lidar com os pedidos das conexões. Depois, esta cria um novo serviço *goroutine* (*thread lightweight*) para cada conexão, o que possibilita o tratamento de múltiplos pedidos em paralelo.

Para controlar o acesso à memória partilhada por estas *goroutines*, de forma a providenciar consistência e manter a integridade dos dados, recorreu-se ao uso do tipo `Map` da biblioteca `sync`[3], que incorpora mecanismos de sincronização, tornando possível e seguro o acesso simultâneo à estrutura de dados por várias *goroutines*. Optou-se pelo uso do mapa concorrente dado já ser oferecido pela biblioteca com ganhos de performance comparativamente ao uso de `Mutex` e `RWMutex` separados. Estes trancam todos os *buckets* do mapa, enquanto que, com o mapa concorrente, cada *bucket* tem o seu próprio `Mutex`.

4.3 Componente CGSM

O componente assim que é executado começa por iniciar vários módulos através do uso de *goroutines*. Cria os serviços *exporters* nos nós do *cluster* que não dispõem de EdgeMon(s), ativa o mecanismo periódico da descoberta de serviços, e inicia o servidor HTTP onde expõe a sua API (tabela 4.2) através da função `ListenAndServe` da biblioteca `net/http`. Os outros módulos são invocados ao longo da execução do sistema.

Apresenta-se de seguida os detalhes de implementação considerados mais relevantes de cada módulo.

4.3.1 Descoberta de serviços

A grande diferença dos sistemas adaptativos em relação aos estáticos é o facto de podermos ter uma gestão automática sem interferência humana. Isto torna o sistema consideravelmente mais consistente perante as ocorrências que existem ao longo do tempo de vida do mesmo: aparecimento, *crash* ou migração de serviços. Este módulo é responsável por eliminar a necessidade de registar a informação de serviços num dado local (e.g. *service registry*, base de dados, entre outros).

Para a gestão de serviços dentro do contexto do *swarm*, utilizou-se um mecanismo de consulta periódica ao Docker Swarm *manager*, integrado no Docker Engine. Deste modo,

[2] https://golang.org/pkg/net/http/
[3] https://golang.org/pkg/sync/

Tabela 4.2: API do componente CGSM

Método	Tipo	Endpoint	Descrição
RegisterEdgemon	POST	/register	Registar um EdgeMon externo ao *swarm*. O CGSM deve receber o URL no *body* do pedido.
CreateEdgemon	POST	/edgemons/{hostname}	Criar um novo EdgeMon. Recebe, como *path variable*, o nome da máquina onde deve ser *deployed*.
RemoveEdgemon	DELETE	/edgemons/{id}	Remover um EdgeMon do sistema. Recebe, como *path variable*, o identificador do EdgeMon em questão. Todos os micro-serviços monitorizados por estes serão eventualmente atribuídos a outro(s).
GetEdgemons	GET	/edgemons	Devolver a lista de EdgeMons registados no CGSM.
HandleMigrate	GET	/migrate/{id}/{hostname}/{keep}	Replicar ou migrar um EdgeMon. As *path variables* necessárias são o identificador do *container* (IP), o nome da máquina destino e um *boolean* que decide se a ação é de migrar ou replicar.
AddRule	POST	/rules/{name}/{low}/{high}/{description}/{type}/{overload}	Adicionar uma nova regra. Recebe como *path variables* o nome da métrica, os limites inferior e superior do intervalo considerado normal, uma descrição da métrica, o tipo de métricas alvo (*container* ou *host*) e se é ou não uma regra a ser utilizada durante a avaliação de sobrecarga dos EdgeMons.
UpdateRule	PUT	/rules/{id}/{interval}	Atualizar uma regra. Recebe, como *path variables*, o identificador da regra e o intervalo de tempo entre duas avaliações consecutivas.
RemoveRule	DELETE	/rules/{id}	Remover uma regra. Recebe, como *path variable*, o identificador da regra.
GetScrapeTargets	GET	/scrapetargets	Devolve a lista de serviços para monitorizar que descobriu.
SetTaskToEdgemon	POST	/edgemons/{id}/scrapetargets/{id}	Atribuir um micro-serviço a um dado EdgeMon. Recebe, como *path variables*, os identificadores do EdgeMon e do micro-serviço, respetivamente.
GetNodeMetrics	GET	/metrics/{swarmnodeid}	Retorna a lista de métricas a nível do *host*. Recebe, como *path variable*, o identificador do nó pertencente ao *swarm*.

assim que o CGSM inicia, realiza periodicamente uma pesquisa para atualizar o estado do sistema.

A partir das funções do pacote cliente do Docker `ServiceList` e `TaskList`, é possível obter todos os serviços e os seus respetivos *tasks*. Destes, obtém-se o necessário para a atividade de monitorização: a que serviços pertencem os *tasks*, informações de identificação do serviço (e.g. nome, etiquetas), as localizações dos *tasks* (URL) e em que nó do *swarm* se encontram. O CGSM extrai vários resultados essenciais ao reunir esta informação:

1. Diferencia o tipo de serviço encontrado, isto é, se é um novo alvo de monitorização, um novo EdgeMon, cScraper, Node Exporter ou outro. Isto consegue-se a partir das etiquetas associadas ao serviço (e.g. um serviço que deva ser alvo de monitorização, quando é criado, deve ter a *label* `Monitoring` com o valor de verdade). Caso encontre um novo EdgeMon, é adicionado uma etiqueta ao nó do *swarm* onde este está a ser executado. As etiquetas dos nós constituem metadados que descrevem os mesmos. A partir destes, é possível criar restrições de *deployment* de outros serviços. Neste caso específico, a etiqueta adicionada é um par *key-value*, onde a chave é "type" e

o valor é "EdgeMon". É através deste mecanismo que é assegurado que os serviços cScraper e Node Exporter só são *deployed* em nós com ausência de EdgeMons. Foi utilizada a função `NodeUpdate` para atualizar as etiquetas do nó;

2. Descobre novos micro-serviços que ainda não estão a ser monitorizados: reúne todos os novos serviços desde a última consulta e adiciona-os ao mapa de micro-serviços não monitorizados;

3. Caso o CGSM tenha recuperado de falhas, recebe informação acerca do estado atual do sistema (quais os micro-serviços já monitorizados e por quem) através de pedidos HTTP GET para o *endpoint* `/scrapedtargets` de cada EdgeMon. Desta forma, evita situações em que um micro-serviço é monitorizado por mais do que um EdgeMon;

4. Guarda o URL e identificador do nó de todos os Node Exporters e cScrapers do sistema.

Para além do aparecimento de novos serviços, é possível, através desta pesquisa, concluir os serviços do *swarm* que desapareceram, ou devido a falhas, ou simplesmente porque foram removidos. Perante a desconexão de um EdgeMon no *swarm*, procede-se à redistribuição dos micro-serviços que lhe competiam pelos restantes monitores. Caso o nó deixe de ter monitores, a etiqueta `EdgeMonNode` é removida do mesmo.

Para endereçar o mesmo problema quando os monitores registados no CGSM são externos ao *swarm*, foi usada uma implementação existente do mecanismo de *heartbeat*[4] que permite verificar a disponibilidade dos mesmos. O protocolo usado para a troca de mensagens é HTTP com um *secret*.

O mecanismo implica o monitor enviar ao servidor (executado no CGSM), de forma periódica, uma mensagem com o seu identificador, o *timestamp* que primeiramente obtém do CGSM e de um *mac hash*. Caso o CGSM não receba esta mensagem num intervalo de tempo pré-definido, assume que o monitor não está vivo e remove-o.

Dada a instabilidade da rede, o CGSM quando tenta comunicar com algum EdgeMon dispõe de um mecanismo de *retries*. Caso a comunicação não seja estabelecida dentro do valor de *timeout* de 5 segundos, volta a reenviar o pedido passado um tempo de espera. Caso exceda o número máximo de tentativas, assume que o serviço falhou e remove-o do sistema utilizando a função `ServiceRemove` do pacote cliente.

Por fim, este módulo invoca o método de atribuição dos novos micro-serviços aos EdgeMons existentes.

4.3.2 Atribuição de novos micro-serviços a EdgeMons

A atribuição de novos micro-serviços a EdgeMons é feita de forma a maximizar a eficiência do sistema de monitorização. Com isto, os fatores tomados em conta na atribuição são:

[4] https://github.com/codeskyblue/heartbeat/

1. **Posicionamento do EdgeMon e micro-serviço**: O facto de ambos estarem no mesmo nó constitui o fator preferencial na atribuição. Caso ambos partilhem o mesmo *host* e o EdgeMon não esteja sobrecarregado, é feita a atribuição com sucesso. Desta forma, é possível obter melhores valores de latência entre comunicações e diminuir o tráfego na rede. Caso não exista nenhum EdgeMon no mesmo nó, é escolhido o EdgeMon com menos trabalho disponível.

2. **Carga do EdgeMon**: De maneira a não sobrecarregar os EdgeMons afetando assim a atividade de monitorização e degradando possivelmente a qualidade de serviço, a carga do EdgeMon é sempre avaliada (em ambos os casos mencionados no ponto 1 contra as regras existentes no motor de regras. Primeiro, obtêm-se as métricas do tipo *container* do EdgeMon, a partir do *endpoint* da sua API `/containermetrics` que as disponibiliza. Depois avaliam-se no motor de regras as regras que indicam sobrecarga dos EdgeMons. Caso alguma das métricas não esteja dentro do intervalo normal definido na regra, é criado um novo EdgeMon. O posicionamento deste é decidido pelo *manager* do Docker Swarm que usa bibliotecas do Swarmkit[5], ferramenta para orquestração de *clusters*. O mecanismo de *deployment* de *tasks* no *cluster*, denominado *scheduling*, escolhe o nó mais apropriado para alojar o *container* do novo monitor com base em filtros (filtragem de recursos, restrições, entre outros).

Quando o EdgeMon é escolhido, é-lhe enviado os alvos de monitorização pelos quais está responsável, através do *endpoint* `/app`. A partir deste momento, começam a ser monitorizados com as configurações padrão. É possível observar todo o processo descrito na listagem 4.4.

Listagem 4.4: Função para atribuir alvos de monitorização aos EdgeMons

```
1  func setEdgeMons(newScrapeTasks []*scrapeTask) {
2      //Docker engine API client
3    cli, _ := client.NewClientWithOpts(client.WithVersion("1.39"))
4      //For each scrape target
5    for _ , newScrapeTask := range newScrapeTasks{
6          //Set new scrape target's cAdvisor and node-exporter locations
7      cadvisorIP:= cadvisors[newScrapeTask.SwarmNodeID]
8          nodeExporterIP:= nExporters[newScrapeTask.SwarmNodeID]
9      newScrapeTask.setCAdvisorIP(cadvisorIP)
10     newScrapeTask.setNodeExporterIP(nodeExporterIP)
11
12     //Set new scrape target's container ID so EdgeMon might use it
13     _, stask := getTask(newScrapeTask.ServiceID)
14     newScrapeTask.setContainerID(stask.Status.ContainerStatus.ContainerID)
15
16         //JSON encoding of the new scrape target
17     body, _:=json.Marshal(newScrapeTask)
18
```

[5]https://github.com/docker/swarmkit/

```
19      decided:=false
20      var edgemonURL string
21      overloaded := make(map[string]bool)
22        //Sort edgemons by number of microservices
23      sort.Slice(edgeMons, func(i, j int) bool {
24      return edgeMons[i].Ntargets < edgeMons[j].Ntargets})
25
26      for _, edgeMon := range edgeMons{
27          //Checks the priority case: is there any edgemon in the same node?
28        if sameNode(*newScrapeTask,edgeMon){
29            //If yes, checks if it's overloaded
30          if !isEdgeMonOverloaded(edgeMon.Target,contID){
31            decided= true
32            //Chosen monitor!
33            edgemonURL=edgeMon.Target
34            break
35          }else{
36            overloaded[edgeMon.Target] = true
37          }
38        }
39      }
40      var serviceID string
41      var task swarm.Task
42      if(!decided){
43              //There was no EdgeMon on the same node or it was overloaded
44          wasEvaluated, _  := overloaded[edgeMons[0].Target]
45          //Checks the least overloaded EdgeMon from the remaining ones
46          if !wasEvaluated && !isEdgeMonOverloaded(edgeMons[0].Target, contID){
47            //Chosen monitor!
48            edgemonURL=edgeMons[0].Target
49          }else{
50              //Every EdgeMon is overloaded
51              //Create a new EdgeMon
52            serviceID = CreateService(TesteServiceSpec,map[string]interface{}{
53            "name": getEmName(), "image": "edgemon:1",
54             "network":"overlay-network-fase2","ports":getEmPorts(),
55            "mounts": edgeMonMounts})
56            var edgemonIP string
57            edgemonIP, task = getEdgeMonIP(serviceID)
58            edgemonURL= edgemonIP+":9999"
59          }
60      }
61      //Assign new scrape target to the new EdgeMon
62      NewHttpRequest("POST", "http://"+edgemonURL+"/app", body)
63      if serviceID!=""{
64          //Update node where EdgeMon is placed
65        updateNode(task.NodeID, map[string]string{"type":"EdgeMonNode"})
66        //Update monitors that scrape targets from this node so they will request
67        //metrics from the monitor instead of cAdvisor and node-exporter
68        updateEdgeMons(task.NodeID, edgemonURL,true)
```

```
69      }
70      // Update state
71    }
72  }
```

4.3.3 Criação de serviços

De modo a criar qualquer tipo de serviço no sistema, foi usada a função `ServiceCreate` do pacote cliente do Docker, que recebe como parâmetro uma função que especifica como o serviço deve ser *deployed* (listagem 4.5), à semelhança do que acontece quando o *deployment* é manual com recurso ao ficheiro de configuração (como exemplificado na listagem 4.3).

Listagem 4.5: Função para definir configurações do serviço

```
1  func ServiceSpec(serviceParams map[string]interface{}) swarm.ServiceSpec {
2    var spec swarm.ServiceSpec
3    ServiceName(serviceParams["name"].(string))(&spec)
4    ServiceImage(serviceParams["image"].(string))(&spec)
5    ServiceNetwork(serviceParams["network"].(string))(&spec)
6    ServiceEndpoint(serviceParams["ports"].([]swarm.PortConfig))(&spec)
7    //if mounts exist
8    if _, ok := serviceParams["mounts"]; ok {
9      ServiceMounts(serviceParams["mounts"].([]mount.Mount))(&spec)
10   }
11   //if placement constraints exist
12   if mode, ok := serviceParams["placement"]; ok {
13     if mode == "global"{
14       ServiceMode()(&spec)
15     }else{
16       ServicePlacement(serviceParams["placement"].(string))(&spec)
17     }
18   }
19   // if constraints exist
20   if (serviceParams["name"].(string) == "cAdvisor" ||
21   serviceParams["name"].(string) == "node-exporter"){
22     ServiceConstraints("EdgeMonNode")(&spec)
23   }
24   return spec
25 }
```

Atribui-se, como demonstrado na listagem 4.5, os vários campos da estrutura `ServiceSpec`, através de funções auxiliares, nomeadamente:

- **Nome do serviço**: Quando são criados EdgeMons, o nome do serviço tem de ter como prefixo "EdgeMon" para poder ser reconhecido durante a descoberta de serviços;
- **Imagem**: Tem de existir na máquina local ou num *registry* público/privado;

- ***Mounts***: Informação estática que o CGSM guarda;
- **Rede**: Rede à qual se deve juntar. Neste caso é sempre a mesma onde o CGSM está a ser executado.
- **Modo do serviço**: Pode ser global ou replicado. No caso dos *exporters*, o modo é global. São executados em todos os nós ativos do *swarm*. No caso dos EdgeMons, é replicado, especificando-se o número de réplicas a serem criadas.
- **Portas**: Mapeamento de portas *container-host* para tornar o serviço disponível a partir do exterior do *swarm*.
- **Posicionamento**: Restrições de posicionamento dos nós. Neste caso, para os serviços replicados, a restrição verifica se o nome do nó do *swarm* é igual ao *input* (e.g. node.hostname==node13). No *deployment* dos *exporters*, a restrição de posicionamento verifica se o nó tem uma etiqueta "type" com o valor "EdgeMonNode" (e.g. node.labels.type==EdgeMonNode). Se sim, implica que existe pelo menos um EdgeMon no nó e, portanto, não são criados.

4.3.4 Migração/Replicação

A migração é o processo de mover um monitor de um nó para outro no *swarm*, enquanto que a replicação cria uma nova réplica com base no estado e configuração de um dado monitor. Evita-se assim a necessidade de reinserção das mesmas regras e reorganização dos micro-serviços na nova instância. A lógica de implementação para ambas as funcionalidades é semelhante na perspetiva do CGSM.

Na migração, o novo monitor passa a existir na nova localização com a mesma configuração e responsabilidade que o anterior, sendo o primeiro depois removido. No caso da replicação, a primeira instância não é eliminada e os micro-serviços são repartidos pelos dois monitores. Assim, para ambos os casos, obtém-se inicialmente os dados que constituem o estado do monitor (regras e micro-serviços), a partir dos *endpoints* disponibilizados por este `/rules` e `/scrapetargets`, respetivamente. Depois, cria-se um novo serviço EdgeMon num dado nó do *swarm* (escolhido pelo cliente no caso da migração e pelo Docker no caso da replicação). Finalmente, o CGSM passa-lhe o estado que obteve, igualmente através da API.

4.4 Componente EdgeMon

Componente encarregue pela monitorização dos micro-serviços. É responsável pela coleta e armazenamento temporário das métricas de diferentes tipos. É também encarregue por extrair e servir métricas dos *containers* com quem partilha a máquina que os aloja e métricas respeitantes ao nó em si, através do Docker, maximizando assim a eficiência do sistema em determinadas condições.

Tabela 4.3: API do componente EdgeMon

Método	Tipo	Endpoint	Descrição
HandleScrapeMode	PUT	/scrape/{mode}	Alterar o `ScrapeMode`. Recebe, como *path variable*, o modo pretendido (*node*, *store* ou *all*).
RegisterScrapeTarget	POST	/register	Registar micro-serviço externo ao *swarm*. Recebe no *body*, a sua localização (URL).
AddScrapeTarget	POST	/scrapetarget	Registar um micro-serviço interno ao *swarm*.
AddScrapeTargetList	POST	/scrapetargets	Registar uma lista de micro-serviços internos ao *swarm*. Recebe no *body*, em formato JSON, os micro-serviços.
GetScrapedTargets	GET	/scrapetargets	Devolver todos os micro-serviços monitorizados por este EdgeMon.
AddRule	POST	/rule/{name}/{min}/{max}/{type}/{alert}	Criar uma nova regra. Recebe, como *path variables*, o nome da família de métricas, os valores mínimo e máximo, o tipo de métricas e o tipo de alerta que deve ser despoletado.
AddRuleList	POST	/rules	Registar uma lista de regras. Recebe no *body*, em formato JSON, as regras.
GetRulesList	GET	/rules	Devolver as regras registadas no EdgeMon
RemoveScrapeTargets	DELETE	/scrapetargets	Remover uma lista de micro-serviços.
GetMetrics	GET	/metrics	Devolver as métricas armazenadas em *cache*.
GetContainerMetrics	GET	/containermetrics/ {containerID}	Devolver as métricas do tipo *container* para o *container* identificado pelo IP recebido como *path variable*
GetNodeMetrics	GET	/nodemetrics	Devolver as métricas do tipo *host* referentes ao nó local.
StopMonitoring	DELETE	/activetargets/{containerID}	Pausa a atividade de monitorização para o *container* identificado pelo IP recebido como *path variable*.
RestartMonitoring	POST	/activetargets/{containerID}	Reinicia a atividade de monitorização para o container identificado pelo IP recebido como *path variable*.
ChangeMonitoringParams	PUT	/activetargets/ {containerIP}/ {metricType}/ {scrapeInterval}	Redefinir parâmetros configuráveis de monitorização. Recebe, como *path variables*, o identificador do *container* em questão, o tipo de métricas e o intervalo entre dois *scrapes* consecutivos desejados.

Disponibiliza uma API (tabela 4.3) para que o CGSM possa comunicar com este e também para o cliente configurar e gerir a atividade de monitorização.

De seguida, apresentam-se os detalhes de implementação considerados relevantes.

4.4.1 Monitorização

A atividade de monitorização realizada pelo EdgeMon é uma atividade composta por duas subactividades: monitorização de alvos de monitorização e auto-monitorização. Isto é, para além do EdgeMon obter métricas referentes às aplicações que lhe são atribuídas, obtém também métricas de si próprio. Desta maneira, o EdgeMon vai enviando alertas, quando as regras são ativadas, para a camada superior CGSM que reconfigura o sistema face as necessidades. Por exemplo, eventos de que a utilização de um recurso excedeu um certo valor podem ser indicadores para prever uma situação de sobrecarga do nó ou *container* onde o EdgeMon está alojado. Caso o CGSM seja notificado destes, pode tentar combatê-los com mecanismos que previnem a sobrecarga como a replicação ou migração.

4.4.1.1 Monitorização de *scrape targets*

As métricas recolhidas dos micro-serviços podem ser de três tipos: *container*, *host* ou aplicação. Relativamente às métricas de aplicação, só são possíveis de obter caso o micro-serviço as exponha ou as injete. Para tal, existem bibliotecas e servidores que ajudam a exportar métricas num formato que o Prometheus consiga consumir. No caso de aplicações Go, é necessário ter instalado as bibliotecas `prometheus`[6], `promauto`[7] e `promhttp`[8]. Depois, as aplicações devem disponibilizar um *endpoint* HTTP `/metrics` que expõe as métricas a partir da função `Handler` da biblioteca `promhttp` (como se pode ver na listagem 4.6).

Listagem 4.6: Expor métricas numa aplicação Go

```
1  package main
2
3  import (
4          "net/http"
5
6          "github.com/prometheus/client_golang/prometheus/promhttp"
7  )
8
9  func main() {
10         http.Handle("/metrics", promhttp.Handler())
11         http.ListenAndServe(":2112", nil)
12 }
```

Outro *exporter* do Prometheus é o Node Exporter, que está incorporado no componente EdgeMon e é usado para obter as métricas do tipo *host*. O Node Exporter obtém métricas do *hardware* e do sistema operativo expostas pelos kernels do Linux. Como se pode ver na listagem 4.7, são criados novos *node collectors* a partir da função `NewNodeCollector` e posteriormente registados no *registry* de forma a serem incluídos na coleção de métricas. Este `NodeCollector` inclui os vários colectores responsáveis por cada tipo de estatísticas (e.g. arp, cpu, ipvs, diskstats, entre outros). A lista de coletores disponíveis para cada sistema operativo é apresentada nas tabelas A.1, A.2 e A.3 no apêndice A. É possível ativar ou desativar qualquer coletor de forma a recolher ou não as métricas que estes obtêm. Depois, o `gatherers` é usado para juntar os resultados dos diferentes `node collectors` registados (só um neste caso). A função `Gather`, finalmente, invoca por sua vez a função `Collect` de cada coletor registado, reunindo as métricas que cada um destes obtém. O Node Exporter, para além de estar incorporado no EdgeMon, é *deployed* como um serviço independente nos nós do *swarm* que não possuem EdgeMons. Assim, dependendo da existência destes, as métricas a nível do *host* são obtidas ou de um EdgeMon ou do serviço Node Exporter que disponibiliza as métricas no *endpoint* `/metrics` da sua API. Esta

[6] https://godoc.org/github.com/prometheus/client_golang/prometheus/
[7] https://godoc.org/github.com/prometheus/client_golang/prometheus/promauto/
[8] https://godoc.org/github.com/prometheus/client_golang/prometheus/promhttp/

otimização evita o gasto desnecessário de recursos computacionais para alojar o Node Exporter nos nós que têm pelo menos um EdgeMon.

Listagem 4.7: Obter métricas do *host*/nó a partir do Node Exporter

```
func nodeMetrics() map[string]*dto.MetricFamily{
  registry := prometheus.NewRegistry()
  var gatherers prometheus.Gatherers

  arr := []string{}

  nc, err := collector.NewNodeCollector(arr...)
  if err != nil {
    log.Warnln("Couldn't create", err)
  }

  err = registry.Register(nc)
  if err != nil {
    log.Errorln("Couldn't register collector:", err)
  }

  gatherers = prometheus.Gatherers{
    registry,
    prometheus.DefaultGatherer,
  }

  mfs, err := gatherers.Gather()
  if err != nil {
    log.Errorln("error gathering metrics:", err)
  }

  metricFamilies := make(map[string]*dto.MetricFamily)
  for _, mf := range mfs {
    metricFamilies[*mf.Name] = mf
  }

  return metricFamilies
}
```

Relativamente às métricas do tipo *container*, à semelhança do que acontece nas de tipo *host*, e pelo mesmo motivo de otimização, podem ser obtidas, conforme o caso, ou de um EdgeMon, ou do *exporter* cScraper.

cScraper é um *daemon* que disponibiliza métricas, para um dado *container*, no *endpoint* `/docker/{containerID}`. Quando o micro-serviço a ser monitorizado está num nó que tem um EdgeMon, as métricas do *container* são obtidas através deste, diretamente do Docker Engine. É possível conseguir as estatísticas sobre um *container* em específico a partir do método `ContainerStats` que recebe, como argumento, o identificador do mesmo. O cScraper implementa o mesmo mecanismo para obter as métricas. Tal como o EdgeMon, este componente dispõe de um *parser* responsável por transformar as métricas

para o formato desejável, explicado na próxima subsecção 4.4.1.2.

A monitorização dos alvos de monitorização é periódica, sendo que o intervalo de tempo entre dois *pollings* e o tipo de métricas a obter é configurável pelo cliente. Similarmente à avaliação das regras (explicado na secção 4.5), é utilizada a biblioteca `cron`[9] para agendar a função de recolha para cada micro-serviço.

Complementarmente, os alvos de monitorização podem injetar as suas métricas diretamente na *cache* através do *endpoint* que o EdgeMon disponibiliza `/metrics/jobs`, indicando o nome do micro-serviços/*job* e etiquetas adicionais em *path variables*. O EdgeMon aplica também a atividade de monitorização a si próprio usando o mesmo conjunto de regras criadas para métricas com o tipo *container* e *host*. Recolhe, diretamente do Docker instalado e executado na máquina local, as métricas do seu próprio *container*, através do método `ContainerStats`.

4.4.1.2 Armazenamento de métricas

Conforme dito anteriormente, o EdgeMon incorpora o Pushgateway do Prometheus, de modo a possibilitar a injeção de métricas por parte dos micro-serviços e também para armazenar as mesmas (servindo de *cache*) para serem, futuramente, disponibilizadas para camadas superiores. Cada conjunto de métricas tem um grupo de etiquetas associado, denominado de *grouping key*, que neste caso é constituído pelo *job*, que representa o nome do micro-serviço monitorizado, instância, que indica o IP do micro-serviço e, por fim, o tipo que descreve o tipo de métricas coletadas.

Intencionalmente, caso existam métricas de um determinado micro-serviço na *cache*, a próxima coleta de métricas vai fazer com que estas sejam substituídas pelas mais recentes sempre que o *grouping key* for o mesmo, mantendo a última versão em memória.

Para as métricas serem armazenadas na *cache*, necessitam de estar sob a forma de mapa onde os valores são do tipo `MetricFamily` da biblioteca go[10]. Para tal, é usado a biblioteca expfmt[11] que contém ferramentas para ler e escrever métricas Prometheus. A função `TextToMetricFamilies` transforma texto no formato consumido pelo Prometheus para um mapa de objetos `MetricFamily`.

Enquanto que o Node Exporter expõe os dados no formato correto de texto, foi necessário implementar um *parser* que lê as métricas do tipo *container* obtidas a partir da função `ContainerStats` e converte-as para o formato desejado. Este formato é orientado por linhas separadas pelos caracteres "\n". São reconhecidos os dois tipos seguintes de registos:

- Começam por # e logo de seguida apresentem o *token* `HELP` ou `TYPE`. No caso de ser `HELP`, espera-se outro *token* com o nome da métrica. Todos os restantes *tokens* são *docstrings* que ajudam a especificar o que é medido. No caso de ser `TYPE`, são

[9] https://godoc.org/github.com/robfig/cron/
[10] https://godoc.org/github.com/prometheus/client_model/go/
[11] https://godoc.org/github.com/prometheus/common/expfmt/

esperados exatamente mais dois *tokens*: o nome da métrica e o seu tipo (*counter*, *gauge*, *histogram*, *summary*, or *untyped*).

- Descrevem métricas, usando a seguinte sintaxe EBNF:

Listagem 4.8: Sintaxe de uma métrica

```
metric_name [
     "{" label_name "=" `"` label_value `"` { "," label_name "=" `"`
     label_value `"` } [ "," ] "}"
    ] value [ timestamp ]
```

4.5 Regras e alertas

Ambos os componentes CGSM e EdgeMon têm o módulo de avaliação de regras sobre as métricas. Tal como o Prometheus permite definir condições de alerta com expressões na linguagem do Prometheus PromQL e enviar notificações para o Alertmanager, fez sentido incorporar um motor de regras simples no EdgeMon. Este já inclui um motor de regras leve, o EasyRulesGo[12], que torna possível o envio de alertas em tempo quase real. De outra forma, e porque os mecanismos de coleta e avaliação de regras do Prometheus são periódicos, faria com que o envio de alertas fosse muitas vezes inútil dado o atraso introduzido pelos *delays* destes mecanismos (e.g. o atraso no envio de um alerta que indica o aumento exponencial de acessos a um micro-serviço pode não evitar com que este falhe pela demora consequente da ação de replicação).

O EdgeMon disponibiliza o *endpoint* `/rules` da API que permite a inserção de três tipos de regras que correspondem aos tipos de métricas que coleciona.

As regras no EdgeMon são avaliadas em dois momentos diferentes: no momento de criação de regras e antes das métricas coletadas serem guardadas na *cache*. Para o primeiro caso, como se pode ver na listagem 4.10, verifica-se inicialmente se existem métricas sobre as quais a regra pode ser avaliada, isto é, se existem métricas da família para qual a regra é destinada. Se sim, para cada métrica, adiciona-se uma nova regra ao motor de regras e este é executado no fim. O funcionamento é semelhante para o segundo caso. Contudo, aqui, o motor de regras adiciona todas as regras registadas destinadas para as famílias das métricas que chegaram, caso existam. Desta forma garante-se que todas as regras são avaliadas sobre as métricas o mais cedo possível. O motor de regras, para cada regra, avalia a sua condição com o valor da métrica.

$$LimiteInferior < x < LimiteSuperior \tag{4.1}$$

Caso seja verdade, nada acontece. Caso contrário, é construído um alerta e posteriormente enviado para as camadas superiores: para o Alerts Pushgateway, caso o alerta seja referente a um micro-serviço monitorizado, ou para o CGSM, quando a regra é ativada

[12] https://github.com/CrowdStrike/easyrulesgo/

(condição não é satisfeita) com métricas respeitantes ao EdgeMon (auto-monitorização). O CGSM é encarregue de receber este tipo de alertas e providenciar a melhor solução que encontra. Neste momento, existem duas soluções para alertas de sobrecarga: particionamento de carga e replicação.

- **Particionamento**: O número de micro-serviços é dividido. O EdgeMon fica responsável apenas por monitorizar uma parte dos micro-serviços que tinha. A outra parte é distribuída pelos restantes EdgeMons do *swarm* caso reúnam condições de recursos.

- **Replicação**: Atribuição de metade dos micro-serviços do EdgeMon sobrecarregado a uma réplica criada (instância que contém as mesmas regras).

O mecanismo de avaliação de regras no CGSM é diferente, apesar das regras e o motor de regras serem idênticos. Neste caso, a avaliação de regras incide sobre as métricas recolhidas dos EdgeMons pelos quais o CGSM é responsável. Com a monitorização periódica de EdgeMons, consegue-se prever padrões e, consequentemente, antecipar ações que suportem o funcionamento normal do sistema (e.g. criar novos EdgeMons, migrar EdgeMons, particionar a carga), prevenindo *crashes* originados pela sobrecarga de EdgeMons ou tempos de resposta elevados dado o posicionamento dos monitores. Assim, é especificado, para cada regra, o intervalo de tempo entre avaliações sucessivas da mesma.

Para a avaliação periódica, é usado a biblioteca `cron`. Esta permite o registo de funções para que sejam agendadas. Como se pode ver na listagem 4.9, é possível agendar uma função com a função `AddFunc`, onde o primeiro parâmetro consiste no modo de agendamento da mesma (e.g. executar a função de ano a ano ou a cada 30 segundos, indefinidamente) e o segundo, a função que deve ser periodicamente executada. Neste caso específico, o primeiro parâmetro é definido pelo cliente. No momento de execução, o `cron` lança uma *goroutine*.

Adicionalmente à avaliação periódica, à semelhança do EdgeMon, as métricas são avaliadas quando é inserida uma nova regra. Este procedimento evita o atraso na avaliação da mesma caso o intervalo entre avaliações seja grande.

Listagem 4.9: Exemplo do modo de agendamento de funções

```
1 id, _ := c.AddFunc("@every "+interval, func() {
2       evaluateEdgeMons(*r.(*rule))
3 })
```

Listagem 4.10: Função que prepara e executa o motor de regras

```
1 func metricsAgainstRule(families []*storage.GobbableMetricFamily, r rule,
2 instance string) {
3     //Creating rule engine instance
4   re := core.NewDefaultRulesEngine()
5   //Finding matches between rule and extracted metrics
```

```
6     for _, family := range families {
7       if *family.Name == r.FamilyName {
8             //Match found
9           for _, metric := range family.Metric{
10            cr := newMetricRule(metric, r.Origin, r.LThreshold,
11            r.HThreshold, r.MetricName, r.MetricHelp, instance)
12            //Add rule to rule engine
13            re.AddRule(cr)
14          }
15      }
16    }
17      //Evaluate and execute
18    err := re.FireRules()
19    if err != nil {
20      fmt.Println("Rules failed!")
21    }
22  }
```

Capítulo 5

Validação e avaliação experimental

De modo a validar e avaliar os componentes, procedeu-se à realização de testes funcionais, testes de performance e ainda outros dois testes, avaliando a migração e comparando esta solução com outra que utiliza apenas o Prometheus. Os testes funcionais têm como objetivo verificar que cada função da aplicação opera em conformidade com os requisitos da sua especificação. Desta forma, testa-se cada funcionalidade, executando-a com um determinado *input*, observando os seus resultados e comparando os mesmos com os resultados esperados. Os testes de performance, por outro lado, permitem avaliar aspetos não funcionais da aplicação (escalabilidade, uso de recursos), quando esta é sujeita a diferentes condições.

As avaliações do sistema foram realizadas utilizando um micro-serviço previamente desenvolvido pela Weaveworks[1], que faz parte da aplicação Sock Shop[2]. Esta aplicação tem como única finalidade ser utilizada para testes de sistemas que incluíam uma arquitetura de micro-serviços. Os micro-serviços estão escritos em várias linguagens e disponibilizam métricas a nível da aplicação no formato compatível com o Prometheus. Neste caso, o micro-serviço usado em todos os testes foi o *Catalogue*, escrito em Go. O tipo de informação recolhida é equivalente para qualquer um e, portanto, o objetivo concreto do micro-serviço não é relevante para esta avaliação experimental.

5.1 Testes funcionais

Para cada teste, primeiro são indicados os objetivos, depois o cenário e procedimento de realização e, por fim, os resultados observados. O sistema de monitorização, durante a

[1] https://www.weave.works/
[2] https://microservices-demo.github.io/

realização destes testes, foi *deployed* nos nós do *cluster* da FCT/UNL com as seguintes especificações:

- **Processador**: 2 Intel Xeon E5-2620 v2 (com *Hyper-Threading*)
- **Memória RAM**: 64 GB
- **Rede**: 2 NIC Intel Corporation I210 Gigabit Network Connection

Os nós estão ligados por rede *Ethernet* de 1GB.

5.1.1 CGSM - Teste 1

5.1.1.1 Objetivos

O objetivo deste teste foi confirmar o funcionamento dos mecanismos da criação, descoberta e atribuição de serviços, explicados nas subsecções 4.3.3, 4.3.1 e 4.3.2, respetivamente. Aproveitou-se também para avaliar os tempos necessários para as diversas operações relacionadas. Estes estão presentes na tabela 5.1.

5.1.1.2 Cenário

Foi criado um *swarm* com três nós do *cluster* (nó 1, nó 2 e nó 3). O nó 1 é manager do *swarm*.

5.1.1.3 Procedimento

1. *Deployment* do EdgeMon nó 2;
2. *Deployment* de dois micro-serviços no nó 1;
3. *Deployment* de um micro-serviço no nó 2;
4. *Deployment* de um micro-serviço no nó 3;
5. *Deployment* do CGSM no nó 1.

5.1.1.4 Resultados

No momento em que o CGSM inicia, este começa por despoletar o mecanismo de pesquisa periódico (neste caso configurado de 30 em 30 segundos), para descobrir os serviços existentes no *swarm*. Demorou em média 131,68 milissegundos para descobrir e armazenar os primeiros serviços Docker existentes (4 micro-serviços e 1 EdgeMon). Paralelamente, cria os serviços *exporters* nos nós 1 e 3. O CGSM alterou como esperado a etiqueta "type" do nó 2 para "EdgeMonNode" após a sua descoberta e portanto os *exporters* não foram criados neste. É também possível concluir, atráves da tabela 5.1, que o valor médio para a inicialização de um serviço de qualquer tipo (desde o momento quando é criado até começar a executar) é de aproximadamente 2,5 segundos. Já o processo que envolve encontrar

Tabela 5.1: Medidas de criação, descoberta e atribuição de serviços

Parâmetro	1ª medida	2ª medida	3ªmedida	Média
Tempo de inicialização do CGSM (s)	2,19	2,19	2,16	2,18
Tempo de inicialização do EdgeMon (s)	1,84	2,30	2,25	2,13
Tempo de inicialização do cScraper (s)	2,79	2,48	2,76	2,67
Tempo de inicialização do Node Exporter (s)	2,50	2,91	2,64	2,68
Tempo de descoberta dos 4 micro-serviços (ms)	130,62	132,34	132,09	131,68
Tempo atribuição de 1 micro-serviço (s)	1,20	1,21	1,24	1,21

um EdgeMon adequado para o micro-serviço, definir e construir um objeto JSON e por fim, enviar o pedido HTTP ao EdgeMon escolhido levou cerca de 1,21 segundos para um micro-serviço.

5.1.2 CGSM - Teste 2

5.1.2.1 Objetivos

Com este teste tenciona-se validar o mecanismo de avaliação de carga dos EdgeMons a partir do motor de regras existente no CGSM (explicado na secção 4.5) e, consequentemente, a ação de replicação que é despoletada quando um EdgeMon está sobrecarregado (explicado na subsecção 4.3.4). Para tal, adicionou-se uma regra ao CGSM que indica sobrecarga. De seguida, foram criados o EdgeMon e o micro-serviço. Mediram-se também os tempos necessários, apresentados na tabela 5.2, das operações associadas a estas funcionalidades.

5.1.2.2 Cenário

Foi criado um *swarm* com dois nós do *cluster* (nó 1, nó 2). O nó 1 é *manager* do *swarm*.

5.1.2.3 Procedimento

1. *Deployment* do serviço CGSM no nó 1;
2. Adicionar regra que indica sobrecarga dos EdgeMons no CGSM;
3. *Deployment* do serviço EdgeMon no nó 2;
4. *Deployment* de um micro-serviço no nó 2.

5.1.2.4 Resultados

Quando o CGSM descobre os serviços e tenta atribuir o micro-serviço ao único EdgeMon existente, verifica primeiro se este está sobrecarregado. Para tal, envia-lhe um pedido HTTP a solicitar as métricas do *container* onde está a ser executado, avaliando depois a

Tabela 5.2: Medidas da recolha e avaliação de métricas

Parâmetro	**1ª medida**	**2ª medida**	**3ªmedida**	**Média**
Tempo avaliação do EdgeMon com uma regra (s)	2,02	1,82	1,72	1,85
Tempo de criação de novo EdgeMon e atribuição do micro-serviço (s)	9,57	9,14	9,12	9,27
Tempo de adicionar uma nova regra (ms)	1,23	1,04	1,21	1,16
Tempo que demora a recolha de métricas (s)	1,06	1,14	1,34	1,18
Dados transmitidos (*bytes*)	860	860	860	860

regra sobre as métricas. Como se pode ver na tabela 5.2, o tempo médio do processo de coleta e avaliação é de 1,85 segundos, sendo que a maior parte deste tempo é consumido pela recolha, que demora em média 1,18 segundos. Este valor alto deve-se ao mecanismo de *parsing* que o EdgeMon usa para disponibilizar as métricas. Os dados recebidos do EdgeMon correspondem a cerca de 860 *bytes*.

A regra inserida neste caso foi ativada, o que despoletou a ação de criar uma nova instância EdgeMon por parte do CGSM, para este ser responsável pelo alvo de monitorização pendente. O processo desde o momento da criação até ao fim de execução da atribuição demora em média 9,27 segundos. Adicionalmente ao tempo exigido para a criação do serviço, existe um *delay* introduzido de, no máximo, 3 segundos para que o Docker Engine disponibilize os dados do novo *container* (e.g. identificador do *container*).

5.1.3 CGSM - Teste 3

5.1.3.1 Objetivos

O teste tem como objetivo validar a migração e replicação de serviços, mecanismos explicados na subsecção 4.3.4. Para tal, inicialmente migrou-se o EdgeMon para outro nó do *swarm*. Espera-se que este continue a monitorizar os mesmos micro-serviços que lhe foram anteriormente atribuídos e a avaliar as mesmas regras previamente registadas. Depois, foi testado o mecanismo de replicação criando uma nova instância com base na configuração e estado da primeira. A partir deste momento, os micro-serviços devem ter sido repartidos pelos EdgeMons e ambas as instâncias devem incluir a regra inserida. A tabela 5.3 apresenta medidas sobre os tempos e quantidade de dados transferidos das operações.

5.1.3.2 Cenário

Foi criado um *swarm* com dois nós do *cluster* (nó 1, nó 2). O nó 1 é *manager* do *swarm*.

5.1.3.3 Procedimento

1. *Deployment* do serviço CGSM no nó 1;
2. *Deployment* do serviço EdgeMon no nó 2;

3. Adicionar 1 regra ao EdgeMon;
4. *Deployment* de 4 micro-serviços no nó 2;
5. Migrar o EdgeMon para o nó 1;
6. Replicar EdgeMon para o nó 2;

5.1.3.4 Resultados

O processo que envolve criar o EdgeMon no nó 1, atualizar o estado do CGSM e o envio de pedidos para a nova instância para lhe atribuir o estado (informação de todos os micro-serviços e uma regra), demora cerca de 9 segundos. Observou-se que este continuou responsável pela monitorização de todos os micro-serviços.

O tempo médio de replicação é semelhante ao da migração, já que envolve as mesmas instruções. A única diferença consiste no tempo adicional em particionar e atribuir os micro-serviços pelas duas instâncias. Quando se replicou, foi atribuído metade dos micro-serviços à nova instância. Consequentemente, o primeiro EdgeMon ficou responsável pela outra parte de micro-serviços. Adicionalmente, o mecanismo de replicação difere da migração já que a instância original não é removida. Este procedimento ocorre durante um período de 10 segundos, em média.

Ambos os mecanismos introduzem o *delay* explicado no teste anterior.

Os dados que equivalem a um micro-serviço na rede correspondem a aproximadamente 426 *bytes*. De uma regra, dizem respeito a 193 *bytes*.

Tabela 5.3: Medidas da migração/replicação de EdgeMons

Parâmetro	**1ª medida**	**2ª medida**	**3ªmedida**	**Média**
Tempo migração de 1 EdgeMon com 1 regra e 1 micro-serviço (s)	9,29	10,21	8,26	9,25
Tempo replicação de 1 EdgeMon com 1 regras e 1 micro-serviços (s)	11,18	11,18	9,14	10,5
Dados de um micro-serviço transmitidos (*bytes*)	426	426	426	426
Dados de uma regra transmitidos (*bytes*)	193	193	193	193

5.1.4 EdgeMon - Teste 1

5.1.4.1 Objetivos

Este teste visa validar os mecanismos de recolha de métricas por parte do EdgeMon, explicados na subsecção 4.4.1.1. Para tal, o EdgeMon responsável pelo micro-serviço existente deve conseguir obter as métricas de todos os tipos a partir das fontes de dados disponíveis. Inicialmente, o micro-serviço é colocado num nó sem EdgeMons. Aqui, o EdgeMon responsável pelo mesmo deve ser capaz de obter as métricas a partir do cScraper e Node Exporter. Quando é migrado um outro EdgeMon para o nó do micro-serviço, os *exporters* devem ser removidos e espera-se que o EdgeMon responsável pelo *scrape*

target seja notificado pelo CGSM, redirecionando os pedidos de obtenção de dados para o EdgeMon migrado.

Durante este teste, mediram-se as métricas de tempo de recolha e transferência de *bytes*, cujos resultados estão presentes na tabela 5.4.

5.1.4.2 Cenário

Foi criado um *swarm* com três nós do *cluster* (nó 1, nó 2 e nó 3). O nó 1 é *manager* do *swarm*.

5.1.4.3 Procedimento

1. *Deployment* do serviço CGSM no nó 1;
2. *Deployment* de 2 EdgeMons nos nós 1 e 3;
3. *Deployment* de um micro-serviço no nó 2;
4. Migração do EdgeMon do nó 3 para o nó 2.

5.1.4.4 Resultados

Inicialmente o CGSM criou os *exporters* no nó 2 e atribuiu o micro-serviço ao EdgeMon posicionado no nó 1. Este iniciou a atividade de monitorização recolhendo, periodicamente, as métricas do micro-serviço e dos *exporters*. Observou-se que, em média, demora 1,18 segundos e 47 milissegundos a obter as métricas do cScraper e Node Exporter, respetivamente. O valor mais alto do cScraper justifica-se com o processamento necessário para converter as métricas para o formato desejado. Depois de se migrar o EdgeMon do nó 3 para o nó 2, o CGSM remove os *exporters* e notifica o EdgeMon do nó 1 da existência de um monitor no nó 2. A partir daqui, o EdgeMon começou por obter as métricas a partir do EdgeMon agora no nó 2. Mediu-se, neste momento, o tempo consumido a obter as métricas da nova fonte de dados. A nível do *container*, o tempo é igualmente justificável com o mecanismo de *parsing* incorporado (média de 1,21 segundos). Já as métricas a nível do *host* eram obtidas depois de cerca de 492 milissegundos.

Para ambos os casos, o pedido das métricas de aplicação disponibilizadas pelo micro-serviço era satisfeito em apenas 5,9 milissegundos, aproximadamente.

Relativamente à quantidade de dados recebidos, o Node Exporter gera cerca de 85750 *bytes*, a aplicação próximo de 6400 *bytes* e, por último, o mecanismo incorporado no cScraper e EdgeMon fornece informação correspondente a 853 *bytes*. A quantidade de métricas a nível do *host* e *container* podem, contudo, variar caso sejam ativados/desativados coletores no Node Exporter ou se o *parser* for implementado para obter outro tipo de métricas a partir dos dados oferecidos pelo Docker.

Tabela 5.4: Medidas EdgeMon

Parâmetro	**1ª medida**	**2ª medida**	**3ªmedida**
Recolher métricas do tipo *container* do EdgeMon (s)	1,23	1,19	1,21
Recolher métricas do tipo *host* do EdgeMon (ms)	490,05	497,20	491,60
Recolher métricas do tipo *container* do cScraper (s)	1,07	1,21	1,28
Recolher métricas do tipo *host* do Node Exporter (ms)	43,15	50,85	49,51
Recolher métricas do tipo aplicação do micro-serviço (ms)	5,90	5,88	6,03
Dados obtidos do tipo *host* (*bytes*)	85757	85750	85745
Dados obtidos do tipo aplicação (*bytes*)	5976	6664	6686
Dados obtidos do tipo *container* (*bytes*)	853	853	853

5.2 Testes de performance

Os testes apresentados nesta secção visam verificar o comportamento e uso de recursos do EdgeMon face a diferentes cenários de configuração. Através destes, pretende-se validar que o EdgeMon cumpre efetivamente os requisitos de uma solução que é criada para ser inserida no contexto da *edge*, onde os nós podem dispor de diferentes recursos e a rede sofrer de maiores latências. Tenciona-se observar uma solução leve que exija poucos recursos computacionais (principalmente CPU e memória RAM) e adaptável às condições de cada momento.

Para tal, realizou-se uma bateria de testes de acordo com o seguinte procedimento. Primeiro é executado o CGSM, um EdgeMon e os *exporters* (Node Exporter e cScraper) em *containers* separados. Depois, é executado um *script*, apresentado na listagem 5.1, que vai criando réplicas do micro-serviço *Catalogue* periodicamente (de 13 em 13 segundos) até atingir um limite configurável. Todos estes serviços fazem parte do mesmo *swarm*. O ambiente está ilustrado na figura 5.1.

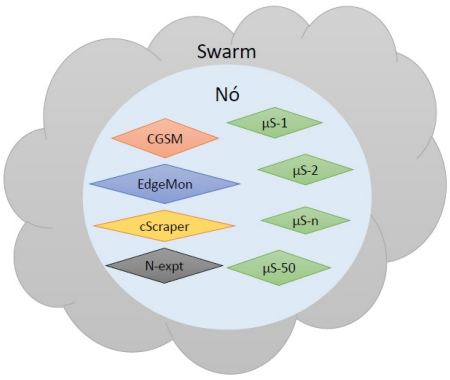

Figura 5.1: Ilustração do ambiente de execução

Durante a execução de cada teste, recolhe-se a partir do comando `docker stats`, estatísticas acerca do uso de memória RAM e CPU consumido pelo *container* que aloja o EdgeMon e sobre quantidade de dados que saem e entram do mesmo. O CPU indicado é resultado da soma dos CPUs usados (quatro). Já as métricas dos dados recebidos e transmitidos correspondem ao total de dados que recebeu/enviou até ao momento.

Os testes foram realizadas num ambiente não distribuído já que os resultados são independentes da latência adicional provocada pela rede. Uma vez não sendo distribuído, esperar-se-ia que as métricas ao nível do nó e *container* fossem recolhidas pelo EdgeMon. Contudo, foram sempre utilizados os componentes *exporters* visto ser o cenário mais provável de acontecer num *deployment* real. Os testes foram feitos numa máquina virtual com o sistema operativo Ubuntu, versão 18.04.1 LTS (Bionic Beaver), que usa 4 CPUs e 4096MB de memória do *host* com as seguintes especificações:

- **Processador**: AMD Ryzen 5 1600 Hexa-Core 3.2GHz com Turbo 3.6GHz 16MB SktAM4
- **Memória RAM**: 8GB (2x4GB) DDR4-2400MHz

Os valores parametrizáveis considerados em cada teste foram: número de regras e intervalo entre recolhas de métricas. O EdgeMon esteve sempre configurado para recolher os três tipos de métricas.

O processo de descoberta de serviços do CGSM ocorre de 30 em 30 segundos. Este processo despoleta a atribuição dos novos micro-serviços, que vão surgindo, ao único EdgeMon existente.

Para cada teste, é apresentado:

- O propósito do teste;
- Os valores dos parâmetros (configuração do EdgeMon);
- Os resultados em forma de gráficos que demonstram o uso de CPU, memória e tráfego de dados ao longo do tempo; tabelas com valores médios e máximos do uso de CPU e memória; e histogramas de ocorrências para intervalos de CPU;
- A análise de resultados.

Listagem 5.1: *Script* que cria micro-serviços periodicamente

```
1 #!/bin/bash
2 FORMAT="'{{.Name}}\t{{.CPUPerc}}\t{{.MemUsage}}\t{{.NetIO}}'"
3 Name=catalogue1
4 COUNT=0
5
6 function function_B {
7   docker stats --format $FORMAT
8   stack1_EdgeMon.1.9re8ceg0q4b32iedg3p1zz54e >> result
```

```
}
function function_A {
current_date_time="`date "+%Y-%m-%d %H:%M:%S"`";
echo $current_date_time;
while [ $COUNT -ge 0 ]; do
  echo "Creating new service: $COUNT"
  docker service create --name ${Name}_${COUNT}
  --container-label "monitoring=true"
  --network overlay-network-fase2 weaveworksdemos/catalogue:0.3.5
  date "+%Y-%m-%d %H:%M:%S" >> services
  COUNT=$((COUNT+1))
  sleep 3s
  if [ $COUNT -ge 31 ] ; then
    break
  fi
done
current_date_time2="`date "+%Y-%m-%d %H:%M:%S"`";
echo $current_date_time2;
}

function_A & function_B
```

5.2.1 Teste performance 1

Tabela 5.5: Parâmetros e duração do teste

Nº micro-serviços	Nº regras	Intervalo entre recolhas	Duração do teste
50	0	1m	11:12m

Este teste tem como objetivo perceber o comportamento do EdgeMon à medida que é responsável por cada vez mais micro-serviços ao longo do tempo. Como seria expectável, a memória tem um crescimento quase linear, como se pode ver pela figura 5.2. À medida que vão sendo adicionados mais micro-serviços, o EdgeMon armazena, consequentemente, maior quantidade de dados à Pushgateway incorporada e em memória. Para além deste aspeto, o número de objetos criados em memória será também maior, apesar do impacto que estes têm no crescimento ser substancialmente menor. Observa-se então que o EdgeMon necessita, de base, de cerca de 5MB de memória e que consome aproximadamente 28MB quando monitoriza 50 micro-serviços.

Relativamente ao CPU, pode-se observar que existem picos ao longo da execução e que estes podem atingir um valor maior à medida que existem mais micro-serviços. Como se pode ver no histograma 5.4, durante grande parte da execução, o processador não é usado pelo EdgeMon (0% de CPU). Por outro lado, os picos devem-se essencialmente às instruções de processamento criadas nos momentos de recolha e armazenamento dos dados. Apesar de haver picos de quase 30% no uso de CPU, o maior número de ocorrências

posiciona-se no intervalo entre 0% e 10%, pelo que a média no final do teste é baixa (2.97%).

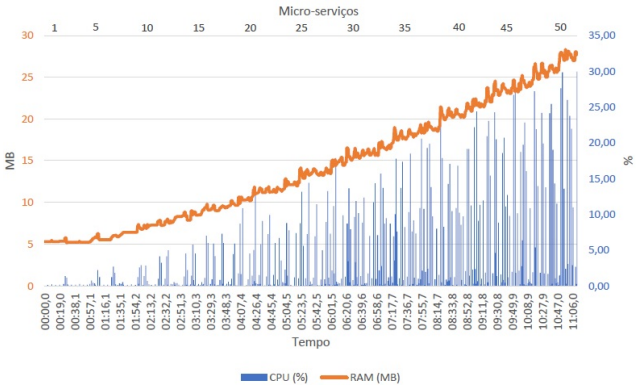

Figura 5.2: Métricas de CPU e memória do EdgeMon

As métricas do tráfego de dados (representadas na figura 5.3) evoluem como seria esperado. É uma linha crescente e não linear já que os valores correspondem à totalidade de dados enviados/recebidos até um dado momento. Esperava-se também que a quantidade de dados recebidos seria substancialmente maior que a dos enviados, já que os dados que saem do *container* correspondem apenas aos pedidos HTTP do mecanismo de *heartbeat* para o CGSM e dos pedidos de recolha de métricas para os alvos de monitorização (*pull*). Pelo contrário, os recebidos envolvem uma quantidade de dados com tamanho significante, que dizem respeito às métricas.

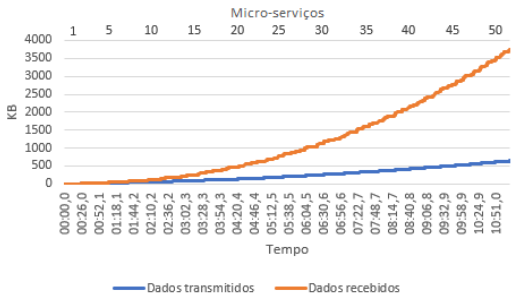

Figura 5.3: Métricas do tráfego de dados do *container* do EdgeMon

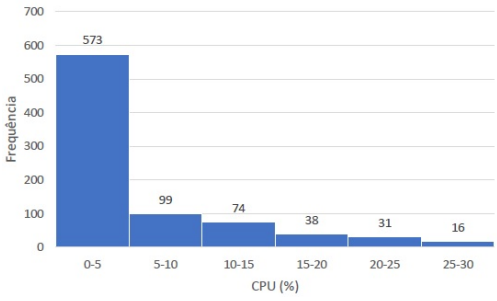

Figura 5.4: Histograma de ocorrências

Tabela 5.6: Valores máximos e médio de CPU e memória

Valor médio CPU	Valor máximo CPU	Valor máximo memória
2,97%	29,92%	28,29MB

5.2.2 Teste performance 2

Tabela 5.7: Parâmetros e duração do teste

Nº micro-serviços	Nº regras	Intervalo entre recolhas	Duração do teste
50	5	1m	10:54m

Com este teste, pretendeu-se analisar o impacto da avaliação de regras no EdgeMon. Para tal, foram extraídas estatísticas do EdgeMon na presença de 5 e 10 regras. O motor de regras avalia as métricas cada vez que estas são recolhidas e todas as regras são avaliadas, neste caso de 1 em 1 minuto para cada alvo de monitorização.

Como se pode ver pelo gráfico 5.5 o comportamento do EdgeMon quanto ao uso de CPU e RAM é semelhante ao que acontece no teste 5.2.1. Esperava-se igualmente que ambas as linhas crescessem com o aumento de micro-serviços e que os valores relativos ao uso de RAM seriam praticamente iguais à do teste 5.2.1, já que a quantidade de dados manipulados em memória é semelhante.

Esperar-se-ia, pelo contrário, que se pudesse observar algumas diferenças a nível do uso de CPU, uma vez o EdgeMon despoleta as instruções de processamento adicionais resultado da avaliação de métricas. Contudo, ao analisar os histogramas 5.7 e 5.8, percebe-se que o impacto destas não é significativo. Os picos voltam a concentrar-se no intervalo de 0% a 10% numa frequência muito parecida (541 e 573) e os restantes valores não correlacionam o uso de CPU com a avaliação de métricas. Isto é, enquanto que no intervalo de 15% a 20%, por exemplo, o número de picos no teste com 5 regras é maior comparativamente ao teste 5.2.1, no intervalo de 25% a 30% acontece o oposto. O mesmo acontece com o teste 5.2.1 e o teste com 10 regras, para os intervalos de 10% a 15% e 25% a 30%, por exemplo. Os valores médios (tabelas 5.8 e 5.9) de CPU para ambos os testes (3,09% e 2,96%) resumem estas conclusões, já que está muito próximo do valor obtido no teste 5.2.1 (2,96%).

Este parâmetro não interfere com o tráfego de dados, e, portanto, os gráficos são semelhantes ao do teste 5.2.1.

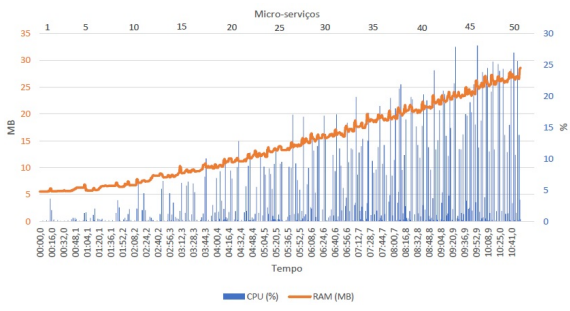

Figura 5.5: Métricas de CPU e memória do EdgeMon com 5 regras

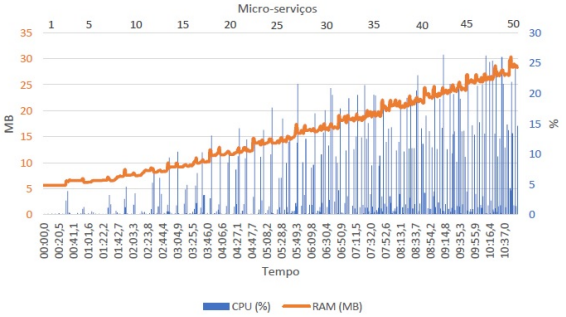

Figura 5.6: Métricas de CPU e memória do EdgeMon com 10 regras

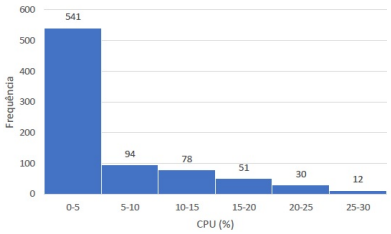

Figura 5.7: Histograma de ocorrências do EdgeMon com 5 regras

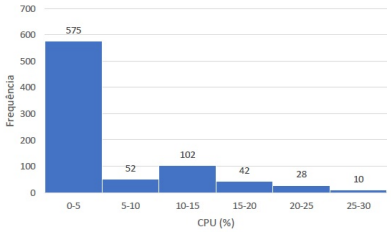

Figura 5.8: Histograma de ocorrências do EdgeMon com 10 regras

Tabela 5.8: Valores máximos e médio de CPU e memória para 5 regras

Valor médio CPU	Valor máximo CPU	Valor máximo memória
3,09%	28,05%	28,60MB

Tabela 5.9: Valores máximos e médio de CPU e memória para 10 regras

Valor médio CPU	Valor máximo CPU	Valor máximo memória
2,96%	26,41%	30,40MB

5.2.3 Teste performance 3

Tabela 5.10: Parâmetros e duração do teste

Nº micro-serviços	Nº regras	Intervalo entre recolhas	Duração do teste
50	0	15s	10:57m

Este teste, pretende mostrar o efeito do intervalo de recolha de métricas na carga do EdgeMon. Neste caso, a cada 15 segundos desde o momento em que o alvo de monitorização é atribuído ao EdgeMon, este último procede à recolha das suas métricas.

Como se pode ver pela figura 5.9, este é um parâmetro que tem um impacto evidente nas métricas aqui representadas.

Ambas as linhas que representam o CPU e RAM crescem, como seria de esperar. Contudo, com a diminuição do intervalo entre recolhas, pode-se observar um aumento significativo no uso dos recursos ao longo do tempo, principalmente de RAM. Apesar da quantidade de dados armazenados no EdgeMon ser igual independentemente do intervalo, o uso de RAM aumenta, apresentando valores próximos de 54MB no fim do teste.

Relativamente ao CPU, previa-se um aumento no valor final da média. Isto porque a quantidade de instruções originadas pela recolha e armazenamento quadruplicava por minuto. Tal aumento comprovou-se como se pode ver na tabela 5.11. O facto de haver mais instruções por minutos fez com que o número de vezes medido em que o CPU não tivesse a ser utilizado baixasse para 240 (ocorrências exibidas no histograma 5.11). Para além da média de CPU que aumentou cerca de 5% comparativamente aos testes

anteriores, verificou-se um aumento incidente principalmente nos intervalos entre 5% e 10% de CPU e entre 20% a 25% em comparação com os testes onde o intervalo era de 1 minuto. Atingiu, também, valores no intervalo entre 30% e 50% pela primeira vez.

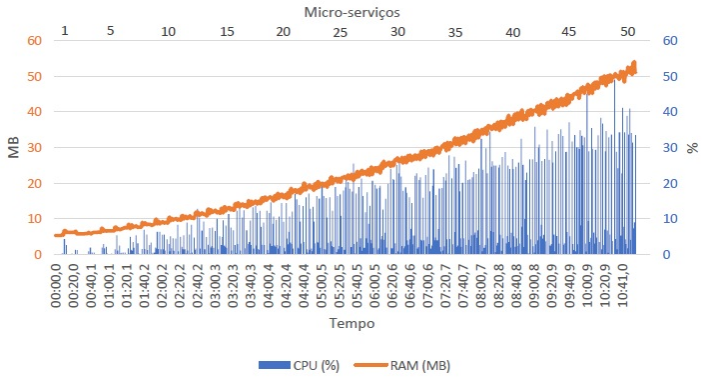

Figura 5.9: Métricas de CPU e memória do EdgeMon

A quantidade de dados introduzidos na rede, visível no gráfico 5.10, é proporcional ao intervalo entre recolhas. Neste caso, a quantidade já é significativa (cerca de 17MB). Este teste permite demonstrar que este parâmetro deve ser otimizado para cada micro-serviço. Por exemplo, caso o cliente não pretenda ser notificado por alertas acerca de uma métrica com valores abaixo de um determinado *threshold* e os valores no momento estejam distantes do mesmo, se estes variarem lentamente ao longo do tempo, é essencial que o intervalo seja reconfigurado para um intervalo maior. Isto porque apesar dos dados introduzidos, resultado de uma única recolha, serem poucos, a soma destes ao longo do tempo é substancial. Por fim, conclui-se que facilmente se introduz carga na rede.

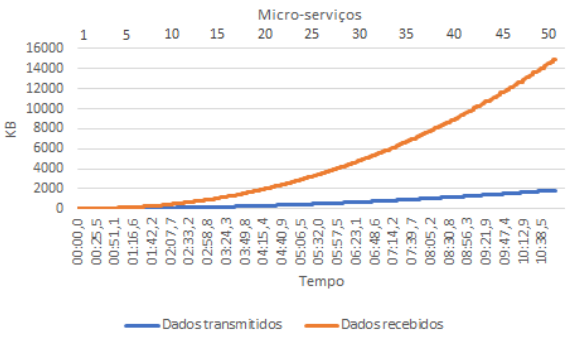

Figura 5.10: Métricas do tráfego de dados do *container* do EdgeMon

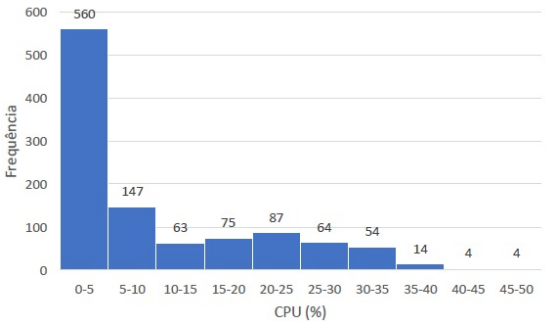

Figura 5.11: Histograma de ocorrências

Tabela 5.11: Valores máximos e médio de CPU e memória

Valor médio CPU	Valor máximo CPU	Valor máximo memória
8,09%	49,09%	54MB

5.2.4 Teste performance 4

Tabela 5.12: Parâmetros e duração do teste

Nº micro-serviços	Nº regras	Intervalo entre recolhas	Duração do teste
31	0	15s	6:57m

Este teste pretende demonstrar uma das principais vantagens do sistema apresentado: a adaptabilidade do sistema à infraestrutura.

Para tal, o teste realizado utiliza o *script* apresentado na listagem 5.1, com a única diferença que o número máximo de réplicas é 31. Depois, foi inserida uma regra que indica sobrecarga dos EdgeMons no CGSM. Assim, no momento de atribuição de alvos de monitorização ao EdgeMon, o CGSM começa por obter a carga naquele momento do *container* onde está a ser executado o EdgeMon. Depois, as métricas recolhidas são avaliadas com base na regra. Neste caso, verifica-se que o valor da métrica "*container memory usage bytes*" é menor que 10MB. Caso seja verdade, o alvo de monitorização é atribuído com sucesso. Caso exceda, é criado um novo EdgeMon e atribuído a este (utilizada a solução de replicação para o problema de sobrecarga).

Com isto, pretende-se apresentar o meio utilizado para ultrapassar as limitações impostas pela possível escassez de recursos nos nós da *edge*. É possível posicionar um EdgeMon em qualquer nó controlando os recursos consumidos por este. Contudo, não se trata de limitar os recursos no sentido em que só são disponibilizados 10MB para o *container* do EdgeMon, mas sim fazer com que este apenas necessite de consumir uma determinada quantidade de memória, atribuindo-lhe assim tantos alvos de monitorização quanto possível. Desta forma, não se interfere com a qualidade da atividade de monitorização.

Pode-se ver a evolução do sistema no gráfico 5.12. Para 31 micro-serviços, foi necessário criar 4 EdgeMons. No fim do teste, cada EdgeMon ficou responsável por 9 micro-serviços, excepto o último que ficou apenas com 4. Isto significa que, em média, os EdgeMon atingiram os 10MB no uso de memória quando ficaram responsáveis pelo 9º micro-serviço.

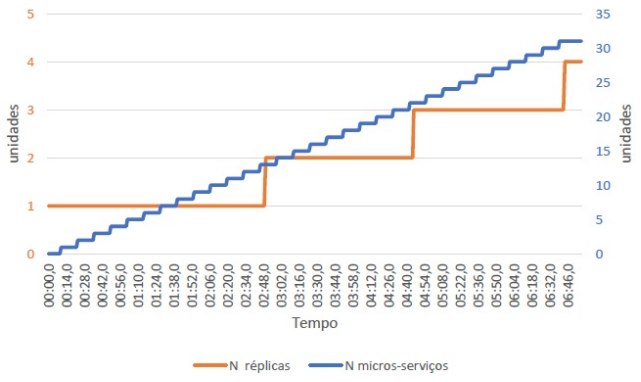

Figura 5.12: Evolução do sistema

O Prometheus, como objeto de comparação, consome de base perto de 15MB de memória. Portanto qualquer nó que disponha menos de 15MB é incapaz de alojar um monitor. Já utilizando o EdgeMon, que ocupa de base 5MB, perante o mesmo cenário, torna possível o *deployment* de um monitor que fique responsável pela atividade de monitorização de cerca de 17 micro-serviços. Adicionalmente, como se pode ver na tabela 5.13 que indica os tamanhos dos diferentes componentes em disco, o EdgeMon tem um tamanho em disco cerca de três vezes menor que o Prometheus. Desta forma, tira-se proveito máximo da infraestrutura disponível.

O processo que origina a criação de novos EdgeMons é a ativação da regra inserida, logo, apesar deste exemplo ter sido criado para um cenário onde se considerava a memória, é possível utilizar qualquer outra métrica para o mesmo fim.

As figuras 5.13, 5.14 e 5.15 mostram o comportamento dos três primeiros EdgeMons desde o momento da sua criação até ao término do teste. É possível observar nos gráficos, que depois de terem ocorrido os eventos que representam a criação de uma nova instância, o uso de memória da última instância criada começa a estabilizar, sendo que a linha não se afasta dos 11/12MB. Surge então a necessidade de existir um valor *threshold* para lidar com a diferença face ao valor inserido na regra.

A análise da evolução das métricas de CPU e tráfego de dados é semelhante à realizada no teste 5.2.1. Conclui-se que é possível fazer com que o monitor se torne numa solução leve a nível de consumo de memória e CPU.

Esta funcionalidade foi igualmente testada em contexto distribuído, onde se utilizou dois nós da Azure com características iguais para simular um *swarm*/*cluster*. O objetivo desta abordagem consistiu em garantir que o *deployment* de EdgeMons era otimizado de acordo com o estado do *swarm*. Isto é, perante vários nós candidatos para alojar as réplicas do EdgeMon, o Docker Swarm deve escolher com base nos seus critérios internos que incluem, por exemplo, o número de serviços executados em cada uma das máquinas, o nó mais adequado. Para tal, forçou-se a execução de um maior número de micro-serviços num nó e observou-se o posicionamento do novo EdgeMon quando o primeiro atinge sobrecarga. Verificou-se que a réplica era sempre posicionada no nó que alojava um menor número de serviços.

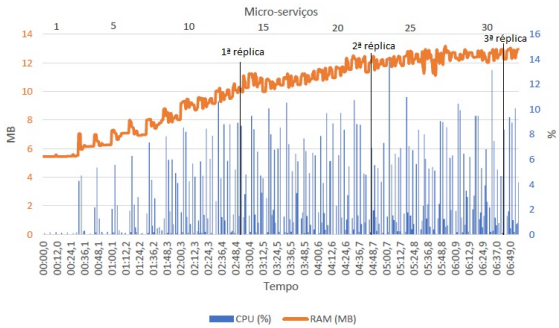

Figura 5.13: Métricas de CPU e memória do EdgeMon criado inicialmente

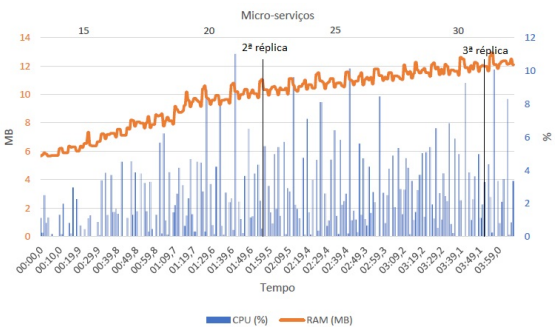

Figura 5.14: Métricas de CPU e memória da 1ª réplica

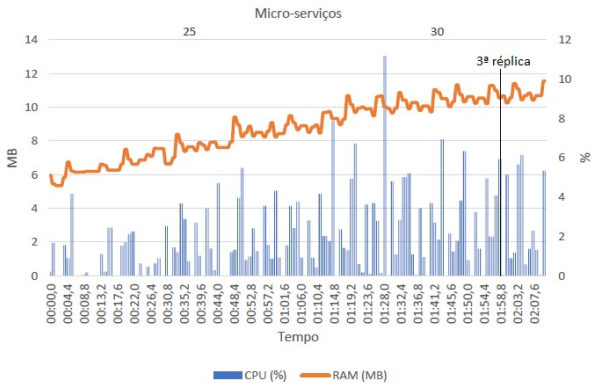

Figura 5.15: Métricas de CPU e memória da 2ª réplica

Tabela 5.13: Espaço em disco em MB usado pelos *containers* que alojam os diferentes componentes

Prometheus	CGSM	EdgeMon	Node Exporter	cScraper	Alertmanager
138	42,7	46,4	22,9	33,3	53

5.3 Teste de migração

Com este teste, pretende-se comprovar a importância da existência de um mecanismo de migração de monitores que permite responder à adaptabilidade necessária em contextos de micro-serviços, onde, neste caso, o sistema de monitorização atua.

Para proceder ao teste, simulou-se um possível cenário que envolve duas localizações separadas por uma distância capaz de introduzir diferenças significativas na latência: Portugal e Estados Unidos da América.

Para tal, este teste foi executado incluindo máquinas na Azure e a máquina local. As máquinas virtuais da Azure, localizadas a este dos EUA, com o sistema operativo Ubuntu Server 18.04 (64-bit), possuem 2 CPUs e 8 GiB de memória RAM. A figura 5.16 demonstra a configuração inicial do sistema. Tem-se um *swarm* com 2 nós, onde um deles é responsável por alojar um monitor e o CGSM e o outro por alojar micro-serviços e os *exporters*. Eventualmente, o CGSM acaba por atribuir todos os micro-serviços ao único EdgeMon existente.

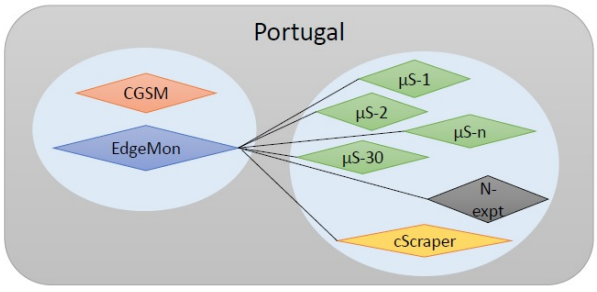

Figura 5.16: Estado inicial do sistema de monitorização

Depois, a certa altura, dois nós da Azure são adicionados ao *swarm* e os micro-serviços são migrados para um destes. Tal como demonstrado na figura 5.17, o EdgeMon continua a monitorizá-los a partir do mesmo nó em Portugal. Neste ponto, foi medido o tempo de recolha de métricas, isto é, o tempo que demora o EdgeMon a coletar os três tipos de métricas de um micro-serviço. Este processo envolve enviar um pedido ao micro-serviço que lhe retorna as métricas a nível da aplicação, enviar um pedido para o Node Exporter que lhe retorna as métricas do nó e por fim ao cScrape que lhe responde com as métricas a nível do *container*. As medições foram feitas variando a carga à qual o EdgeMon estava sujeito, ou seja, quando o EdgeMon era responsável por 1, 5, 10, ou 30 micro-serviços. Como se pode ver pela tabela 5.14, a variação do número de micro-serviços não teve qualquer impacto no tempo de recolha de métricas, sendo que esta ronda os 2 segundos. Neste caso, a falta de impacto deste parâmetro seria de esperar já que o número de *threads* criadas no momento de servir cada *polling* nunca ultrapassa o número de *threads* disponibilizadas pela máquina.

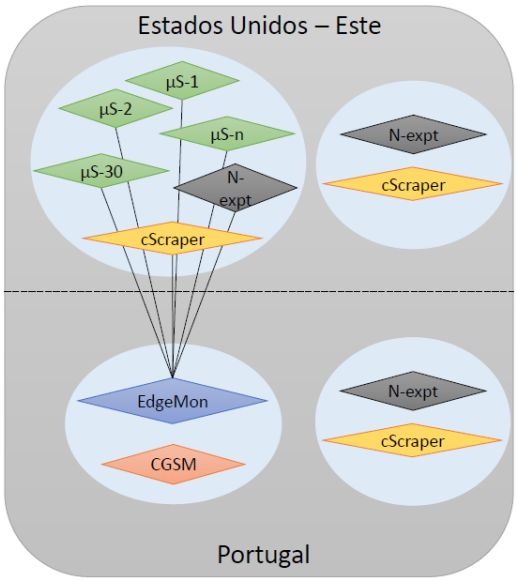

Figura 5.17: Aparecimento de novos nós e migração de micro-serviços

Finalmente, em resposta à migração dos micro-serviços, o utilizador do sistema de monitorização opta por migrar o EdgeMon para perto dos micro-serviços, para outro nó posicionado nos EUA, situação representada pela figura 5.18. Aqui, foi igualmente medido o tempo de recolha de métricas de modo a perceber a influência da distância entre as entidades de monitorização na latência. A tabela 5.15 apresenta os resultados observados. Como seria de esperar, o tempo consumido pelo mecanismo de coleta é significativamente menor (cerca de 1 segundo a menos).

Esta redução de tempo pode afetar consideravelmente, de forma positiva, o comportamento do sistema de monitorização em termos de performance que se reflete, posteriormente, na QoS entregue aos clientes. Primeiro, porque a avaliação das métricas sucede, neste caso, 1 segundo mais cedo, no que irá resultar na entrega de alertas num tempo mais

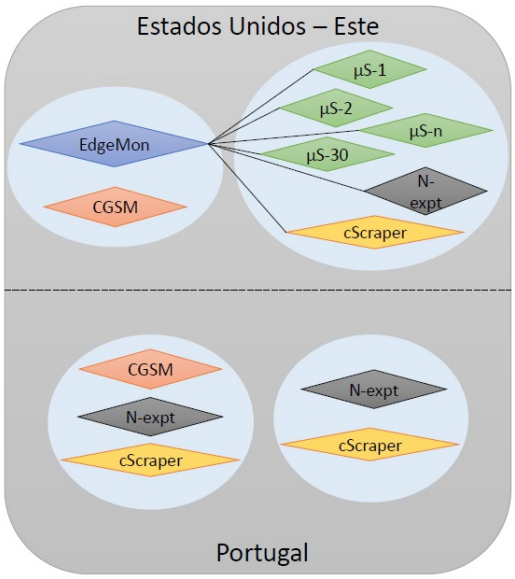

Figura 5.18: Migração do EdgeMon

Tabela 5.14: Tempos de recolha de métricas com o monitor em Portugal e os micro-serviços nos EUA

Nº micro-serviços	**1**	**5**	**10**	**30**
1ª medida (s)	2,29	2,16	2,22	2,31
2ª medida (s)	2,15	2,05	2,17	2,32
3ª medida (s)	2,16	2,19	2,20	2,24
Média (s)	2,2	2,1	2,2	2,3

Tabela 5.15: Tempos de recolha de métricas com o monitor e os micro-serviços nos EUA

Nº micro-serviços	**1ª medida (s)**	**2ªmedida (s)**	**3ªmedida (s)**	**Média**
1	1,16	1,20	1,23	1,2

aproximado do real. Depois, apesar da carga induzida neste teste não ter sido suficiente para que seja criada uma fila de espera para servir os vários pedidos, é possível que esta surja ao longo do tempo. Perante isto, quanto menor for o tempo para servir um pedido, que neste caso, pode ser reduzido diminuindo o tempo introduzido pela comunicação na rede, maior será o número de pedidos satisfeitos por unidade tempo.

5.4 Teste comparativo

Com a realização deste teste, pretende-se demonstrar o benefício ao utilizar uma solução que tira partido da infraestrutura no *continuum* entre a *edge* e *cloud* comparativamente às soluções conhecidas que adotam uma topologia centralizada na *cloud*. Mais especificamente, este teste destaca a resolução de dois problemas principais que este sistema de monitorização visa resolver: evitar a sobrecarga dos monitores e reduzir o tráfego de

dados para a *cloud*. Para tal, foi criada uma infraestrutura com nós heterogénea, geograficamente distribuída, onde certos nós possuem mais recursos computacionais que outros, nomeadamente:

- Nó localizado na austrália: 1 vcpus e 2 GiB de memória
- Nó localizado no Reino unido: 1 vcpus e 2 GiB de memória
- Nó localizado no Japão: 1 vcpus e 3.5 GiB de memória
- Nó localizado nos Estados Unidos: 2 vcpus e 8 GiB de memória

O nó localizado nos EUA representa a *cloud* e, por conseguinte, dispõe de maior capacidade computacional, enquanto que os restantes simulam nós da *edge*, localizados mais próximos dos alvos de monitorização.

Primeiro, foi criado o cenário de uma solução com topologia centralizada, sem usar EdgeMons, representada na figura 5.19. Foram *deployed* 30 micro-serviços em cada um dos três nós da *edge*. No nó da *cloud*, executou-se o monitor Prometheus. Este ficou responsável por todos os micro-serviços, recolhendo assim as métricas a nível da aplicação que estes disponibilizavam com uma periodicidade configurada de 1 segundo.

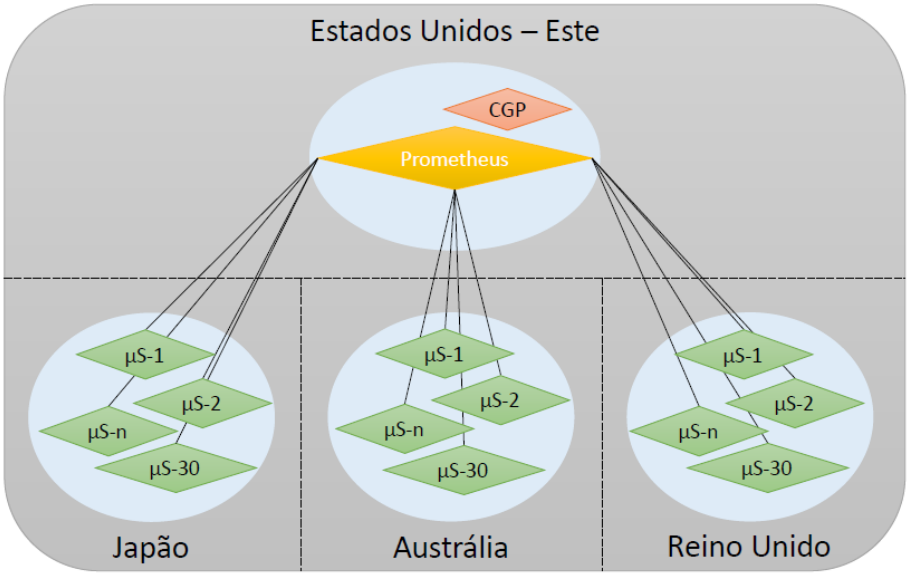

Figura 5.19: *Deployment* usando apenas o Prometheus como entidade de monitorização

Depois, criou-se um cenário onde se conciliou o Prometheus com EdgeMons, de forma a concentrar a atividade de monitorização num local próximo dos micro-serviços. Neste caso, os EdgeMons foram *deployed* nos três nós que alojavam os mesmos. Assim, o EdgeMon posicionado no nó do Japão monitorizava os micro-serviços do mesmo nó e assim sucessivamente. Depois, o Prometheus recolhia as métricas de cada um dos monitores locais e não diretamente dos micro-serviços como no primeiro cenário. Este cenário está representado na figura 5.20. O intervalo entre recolhas foi igualmente definido como

1 segundo para ambas as situações: periodicidade que o Prometheus recolhia métricas dos EdgeMons e periodicidade que estes últimos recolhiam métricas dos micro-serviços. Desta forma, é possível comparar o CPU consumido pelo Prometheus obtendo dados igualmente recentes.

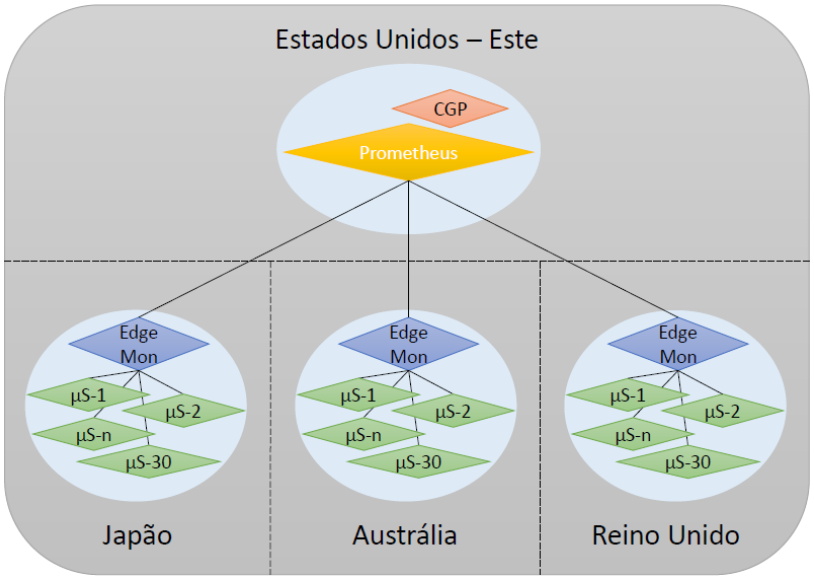

Figura 5.20: *Deployment* conciliando o Prometheus na *cloud* e EdgeMons na *edge*

Durante 5 minutos, mediram-se os valores de CPU e de transferência de *bytes* para o *container* que alojava o Prometheus. No primeiro cenário, o consumo de CPU variou, principalmente, entre os valores de 12% e 18%, visível no gráfico 5.21, que resulta num valor médio final de 14,96% (tabela 5.16). Relativamente ao tráfego gerado, como se pode ver pela figura 5.23, as métricas transferidas dos alvos de monitorização para o Prometheus refletem-se em 50MB e os pedidos, por parte do Prometheus, em 10MB.

No segundo cenário, observou-se que, diminuindo o número de alvos pelo qual o Prometheus é responsável, obteve-se ganhos significativos em termos de consumo de CPU. Como a figura 5.22 demonstra, o consumo de CPU variou entre 4% e 8%, obtendo-se um valor médio final de 5,70%, cerca de 10% inferior comparativamente ao primeiro cenário. O tráfico de dados é semelhante.

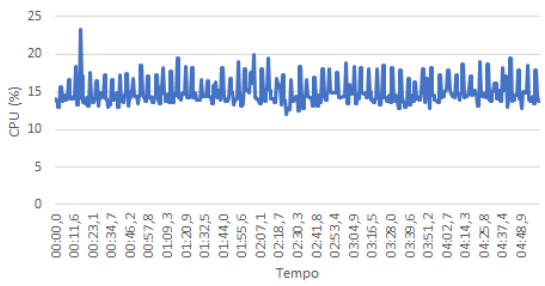

Figura 5.21: Uso de CPU pelo *container* que aloja o Prometheus - primeiro cenário

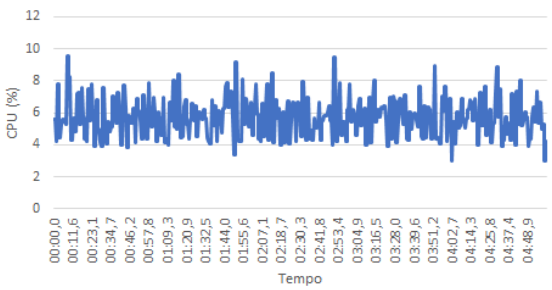

Figura 5.22: Uso de CPU pelo *container* que aloja o Prometheus - segundo cenário

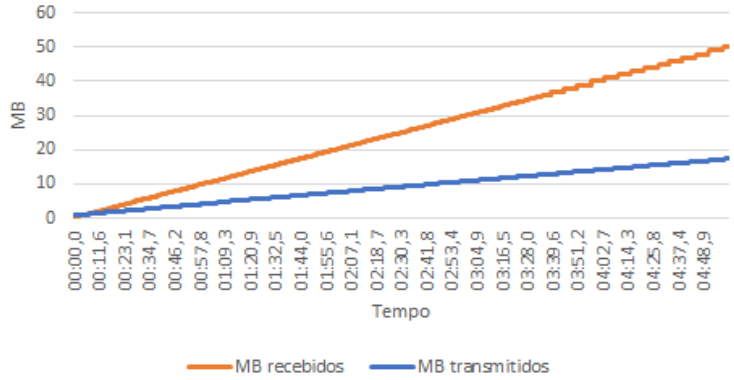

Figura 5.23: Quantidade de *bytes* que entram e saem pelo *container* que aloja o Prometheus

Já o consumo de CPU por parte dos EdgeMons, apresentado nas figuras 5.24, 5.25 e 5.26, resultam em valores próximos de 4% e 7%.

Perante estes resultados, conclui-se que é possível diminuir a carga do Prometheus apenas reduzindo o número de alvos pelo qual é responsável. Faz também sentido que o Prometheus obtenha as métricas dos EdgeMons com menor frequência em relação à frequência definida nos EdgeMons, já que estes dispõe de mecanismos de avaliação de métricas e alertas para, por exemplo, métricas cujo seu valor altere muito rapidamente em curtos espaços de tempo. Aumentando este parâmetro, é possível reduzir drasticamente o número de *bytes* introduzidos na rede e transferidos para a *cloud*.

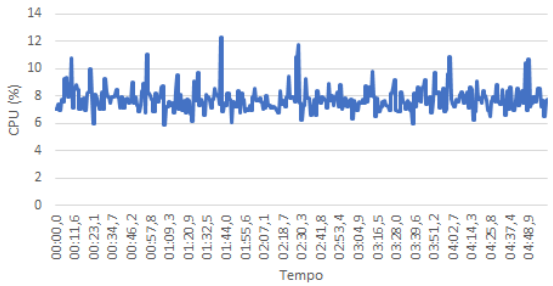

Figura 5.24: Uso de CPU pelo *container* que aloja o EdgeMon do Japão ao longo do tempo

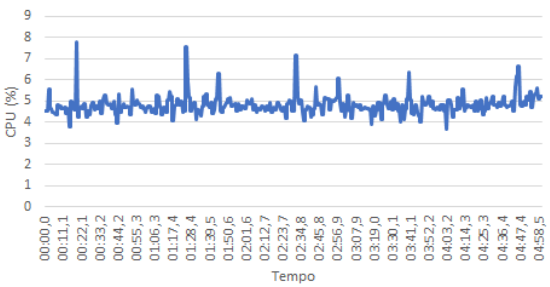

Figura 5.25: Uso de CPU pelo *container* que aloja o EdgeMon da Austrália ao longo do tempo

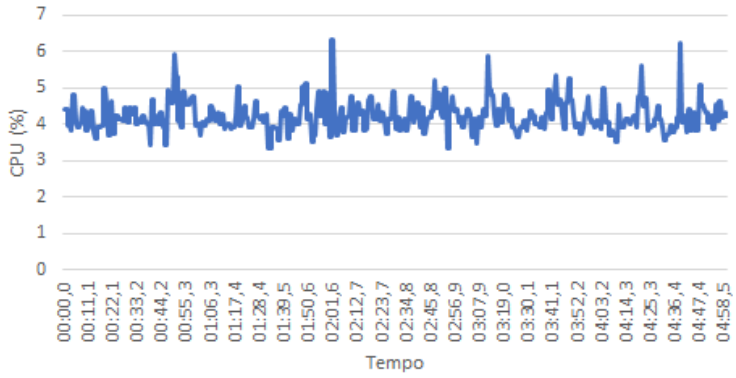

Figura 5.26: Uso de CPU pelo *container* que aloja o EdgeMon do Reino Unido ao longo do tempo

Tabela 5.16: Percentagem do uso médio de CPU

Prometheus cenário 1	**Prometheus cenário 2**	**EdgeMon Japão**	**EdgeMon Reino Unido**	**EdgeMon Austrália**
14,96	5,70	7,76	4,82	4,23

Capítulo 6

Conclusões e trabalho futuro

Nesta dissertação foi apresentado um protótipo no contexto da gestão e monitorização de aplicações em ambientes híbridos que combinam os recursos heterogéneos da *cloud* e da *edge*. Por um lado, os nós da infraestrutura podem falhar e são muito heterogéneos, apresentando muitos deles uma capacidade computacional muito limitada e incapaz de suportar o correto funcionamento da aplicação. Por outro, as aplicações são constituídas por um elevado número de micro-serviços passíveis de falhar, com requisitos computacionais muito distintos e volumes de acessos muito variáveis ao longo do tempo. A solução apresentada visa, principalmente, providenciar a atividade de monitorização correta, ágil e adaptável para aplicações de micro-serviços nesses ambientes, resultando num sistema de monitorização que pode ser mais leve mas que continua a notificar com baixa latência as situações consideradas urgentes.

Tirar partido dos recursos dos nós computacionais na *edge* para alojar um sistema de monitorização composto por componentes mais leves e mais próximos dos pontos de interesse, não só reduz o tráfego induzido e mitiga a congestão na rede, como contribui para a redução dos custos computacionais e energéticos dos centros de dados da *cloud*. Também, uma vez que os dados são recolhidos e podem ser processados/filtrados na *edge*, próximo do local onde são gerados, faz com que se diminua a latência na recolha e consequentemente, na entrega de alertas. Por vezes, é também possível diminuir o tempo de entrega das métricas aos clientes do monitor com uma posição geográfica mais próxima da periferia relativamente à *cloud*, melhorando assim a QoS providenciada aos mesmos. Este tipo de ambiente, aliado com as características da arquitetura de micro-serviços, onde estes podem ser criados, eliminados ou migrados num espaço muito curto de tempo, exigem a criação de componentes de monitorização. Estes devem disponibilizar funcionalidades que suportem ações de adaptação em termos dos alvos e regras de monitorização, bem como da sua própria arquitetura.

A maioria dos sistemas existentes apresenta uma solução orientada para o contexto da *cloud*, não abordando os benefícios da utilização de uma estrutura heterogénea. Dado isto, nenhuma destas soluções conhecidas, à altura da realização deste trabalho, reúne todos os requisitos enumerados na secção 3.2.

O trabalho desenvolvido envolveu a criação dos componente CGSM, CGP e cScraper e a extensão e aprimoramento do componente que existia EdgeMon. O EdgeMon foi criado para que seja possível alojar monitores na *edge*. Este constitui uma versão mais leve do que o *Prometheus*, capaz de recolher métricas a diferentes níveis (*container*, aplicação e *host*), armazenar as suas últimas versões, e avaliá-las com recurso a um motor de regras incorporado. Para conseguir obter as métricas do tipo *container*, foi criado o cScraper que é *deployed* em conjunto com o outro *exporter*, Node Exporter, em todos os nós que não disponham de pelo menos um EdgeMon. Já o CGSM, foi desenvolvido para gerir tanto os alvos de monitorização como os EdgeMons. Este gestor é responsável pela descoberta e adaptação da atividade de monitorização, isto é, executar/remover os componentes necessários para a monitorização no *cluster* pelo qual é responsável e obter informação acerca do estado dos nós e serviços pertencentes ao mesmo. Adicionalmente, monitoriza os EdgeMons com recurso a um motor de regras similar ao EdgeMon e, consoante o que observa, é capaz de alterar a arquitetura do sistema de monitorização, adicionando ou removendo instâncias de monitores. Ambos os componentes disponibilizam uma API para ser utilizada pelos clientes do sistema de monitorização. No caso do EdgeMon, são disponibilizados *endpoints* para a definir parâmetros configuráveis, controlar a atividade de monitorização para um dado micro-serviço, inserir regras e obter as métricas armazenadas na cache. Já o CGSM, permite manipular e gerir manualmente os monitores. É ainda possível controlar as regras de monitorização para estes, criar/remover/migrar instâncias e delinear os alvos de monitorização de acordo com os monitores, permitindo construir soluções que se adaptam às alterações na aplicação, na infraestrutura ou nos próprios monitores. O CGP foi também criado para gerir os alvos de monitorização, mas agora da responsabilidade do Prometheus.

A avaliação incluiu testes de funcionalidade em cenários de utilização onde foi possível demonstrar a flexibilidade e dinamismo do sistema, perante situações de descoberta de novos serviços da aplicação ou de adaptação dos monitores em resposta à sobrecarga. Adicionalmente às medições realizadas para as diferentes operações, avaliou-se também o uso de recursos durante o funcionamento dos componentes com testes de performance. A avaliação permitiu comprovar que o sistema consegue facilmente adaptar-se às condições da infraestrutura. A existência de regras no CGSM que despoletam mecanismos de replicação/migração/particionamento quando são ativadas, permitem que o sistema se adapte aos recursos disponíveis, continuando a providenciar a atividade de monitorização desejável. Foi também possível observar o impacto que cada parâmetro configurável provoca no uso dos recursos na execução de um EdgeMon. Complementarmente, foram realizados testes para comprovar os benefícios reais para os clientes do sistema, a nível do tempo de resposta de métricas e entrega de alertas. Para tal, compararam-se os tempos

variando a proximidade do sistema de monitorização com os micro-serviços e cliente. Por fim, demonstrou-se a partir do teste comparativo que a solução desenvolvida nesta dissertação, que combina o Prometheus com um número adaptável de EdgeMons, apresenta melhores resultados a nível de tráfego de dados, latências e carga induzida nos monitores, face a soluções centralizadas na *cloud* que não usam EdgeMons.

Conclui-se que todos os objetivos inicialmente propostos foram atingidos. No entanto, na secção seguinte, apresentam-se alguns aspetos considerados relevantes para aprimorar o sistema.

6.1 Trabalho futuro

O principal objetivo com o desenvolvimento deste sistema de monitorização foi implementar e testar a viabilidade de um conjunto de funcionalidades que suportavam o dinamismo existente no contexto da *edge*. Posto isto, é possível identificar um conjunto de tarefas que não foram abordadas por não estarem propriamente relacionadas com o âmbito do trabalho, mas que não deixam de ser essenciais:

- **Integração com componente gestão de micro-serviços**: A integração do sistema de monitorização com o componente de gestão de micro-serviços, pretende vir a proporcionar uma maior autonomia ao sistema. Neste momento, o *deployment* do sistema de monitorização requer intervenção humana para especificação dos nós onde o CGSM e EdgeMons iniciais são alojados. Similarmente, o procedimento atual para dar resposta aos eventos (e.g. migração de serviços, alertas de sobrecarga de EdgeMons, entre outros) que ocorrem ao longo do tempo, exige que haja, primeiramente, uma análise do cliente para entender como deve agir (e.g. entender a nova localização dos micro-serviços que migraram para depois migrar/criar/remover o(s) EdgeMon(s) responsáveis por estes). Com recurso ao *service registry*, base de dados onde são guardadas as localizações dos micro-serviços, que o componente de gestão de micro-serviços possa oferecer, será então possível automatizar estes mecanismos já que se tem uma visão global do sistema.

  Para além disso, a integração dos dois componentes permitirá a manipulação dos serviços externos ao *swarm* (micro-serviços e EdgeMons). Neste momento, caso um destes falhe, o sistema de monitorização esquece-o depois de um número de tentativas de comunicação falhadas por *timeout*. Quando o serviço recupera da falha, existe necessidade que ele se volte a registar no EdgeMon ou CGSM, conforme o caso. Isto implica que o serviço externo tenha de conhecer a localização destes componentes.

  Por último, a integração permitirá acelerar a atividade de monitorização e reduzir a carga do CGSM. Este necessita de interrogar o Docker, periodicamente, para obter o estado do *swarm* atual. Caso o intervalo entre consultas seja grande e dado que o tempo de vida dos micro-serviços pode ser bastante curto, corre-se o risco que este

não seja monitorizado. Com a existência do *service registry*, o CGSM podia ser notificado acerca do aparecimento/desaparecimento de serviços assim que ocorressem, resultando numa monitorização quase imediata.

- **Interface gráfica do utilizador**: A criação de uma *interface* gráfica pode facilitar o uso de sistema de monitorização, possibilitando que a gestão e controlo possa ser feita por pessoas não especializadas na área. Para além de simplificar a inserção ou alterações dos parâmetros configuráveis dos componentes CGSM e EdgeMon, seria útil poder analisar o estado do sistema em tempo real. Isto é, conseguir dispor a informação útil, de forma simples e intuitiva, acerca dos *clusters* existentes no momento: quais os EdgeMons e micro-serviços que os constituem; carga dos EdgeMons; carga dos CGSM; mapa que mostrasse os nós de cada *cluster* e onde os serviços estão posicionados.

- **Adicionar funções de agregação no componente EdgeMon**: O processamento de dados na periferia evita que a totalidade de métricas recolhidas sejam transmitidas para os monitores acima na hierarquia ou para as camadas superiores (Prometheus). Através destas funções, podem ser criadas novas métricas que sumarizem e filtrem as métricas recolhidas (por exemplo, calculando estatísticas em intervalos de tempo). Para além de diminuir o tráfego na rede, permitem diminuir a carga em termos de processamento e armazenamento dos monitores que possam vir a receber/manipular as métricas.

- **Mecanismo autenticação**: Neste momento não existe nenhum mecanismo para proteger as APIs. Logo, os recursos podem ser manipulados por qualquer pessoa com acesso à rede privada dos serviços. Com um mecanismo de autenticação a integrar, o acesso aos *endpoints* das APIs é restrito aos utilizadores que apresentem um *token* permitindo acesso a pontos específicos devidamente autorizados, sem necessidade de acesso à rede privada do monitor ou da aplicação.

Bibliografia

[1] G. Aceto, A. Botta, W. de Donato e A. Pescapè. "Cloud monitoring: A survey". Em: *Computer Networks* 57.9 (2013), pp. 2093 –2115. ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2013.04.001. URL: http://www.sciencedirect.com/science/article/pii/S1389128613001084.

[2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica e M. Zaharia. "A View of Cloud Computing". Em: *Commun. ACM* 53.4 (abr. de 2010), pp. 50–58. ISSN: 0001-0782. DOI: 10.1145/1721654.1721672. URL: http://doi.acm.org/10.1145/1721654.1721672.

[3] D. Berman. *Prometheus vs. Graphite: Which Should You Choose for Time Series or Monitoring?* Accessed: 2019-01-01. 2018. URL: https://logz.io/blog/prometheus-vs-graphite/.

[4] D. Bermbach, F. Pallas, D. G. Pérez, P. Plebani, M. Anderson, R. Kat e S. Tai. *A Research Perspective on Fog Computing*.

[5] K. Bilal, O. Khalid, A. Erbad e S. U. Khan. "Potentials, trends, and prospects in edge technologies: Fog, cloudlet, mobile edge, and micro data centers". Em: *Computer Networks* 130 (2018), pp. 94 –120. ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2017.10.002. URL: http://www.sciencedirect.com/science/article/pii/S1389128617303778.

[6] A. Brogi, S. Forti e M. Gaglianese. "Measuring the Fog, Gently". Em: *Service-Oriented Computing*. Ed. por S. Yangui, I. Bouassida Rodriguez, K. Drira e Z. Tari. Cham: Springer International Publishing, 2019, pp. 523–538. ISBN: 978-3-030-33702-5.

[7] B. Butzin, F. Golatowski e D. Timmermann. "Microservices approach for the internet of things". Em: *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*. 2016, pp. 1–6. DOI: 10.1109/ETFA.2016.7733707.

[8] S. Carlini. *The Drivers and Benefits of Edge Computing*. Schneider Electric.

[9] G. Cloud. *Containers at Google*. Accessed: 2019-08-22. 2019. URL: https://cloud.google.com/containers.

[10] T. C. L. Company. *virtual machine monitor*. Accessed: 2019-08-23. 2019. URL: https://encyclopedia2.thefreedictionary.com/VMM+types.

[11] A. V. Dastjerdi e R. Buyya. "Fog Computing: Helping the Internet of Things Realize Its Potential". Em: *Computer* 49.8 (2016), pp. 112–116. ISSN: 0018-9162. DOI: 10.1109/MC.2016.245.

[12] A. V. Dastjerdi, H. Gupta, R. N. Calheiros, S. K. Ghosh e R. Buyya. "Fog Computing: Principles, Architectures, and Applications". Em: *CoRR* abs/1601.02752 (2016).

[13] S. Daya, N. V. Duy, K. Eati, C. M. Ferreira, D. Glozic, V. Gucer, M. Gupta, S. Joshi, V. Lampkin, M. Martins, S. Narain e R. Vennam. *Microservices from theory to practice*. First. IBM Corporation, International Technical Support Organization, 2015. ISBN: 0738440817.

[14] Docker. *About images, containers, and storage drivers*. Accessed: 2019-08-21. 2019. URL: https://docs.docker.com/v17.09/engine/userguide/storagedriver/imagesandcontainers.

[15] Docker. *How services work*. Accessed: 2019-08-20. 2019. URL: https://docs.docker.com/engine/swarm/how-swarm-mode-works/services.

[16] Docker. *Services*. Accessed: 2019-08-20. 2019. URL: https://docs.docker.com/get-started/part3.

[17] Docker. *Swarm mode overview*. Accessed: 2019-08-20. 2019. URL: https://docs.docker.com/engine/swarm.

[18] Docker. *Use overlay networks*. Accessed: 2019-08-19. 2019. URL: https://docs.docker.com/network/overlay.

[19] R. Dua, A. R. Raja e D. Kakadia. "Virtualization vs Containerization to Support PaaS". Em: *2014 IEEE International Conference on Cloud Engineering*. 2014, pp. 610–614. DOI: 10.1109/IC2E.2014.41.

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

[21] V. Farcic. *The DevOps 2.1 Toolkit: Docker Swarm*. Packt Publishing, 2017. ISBN: 1787289702, 9781787289703.

[22] K. Fatema, V. C. Emeakaroha, P. D. Healy, J. P. Morrison e T. Lynn. "A survey of Cloud monitoring tools: Taxonomy, capabilities and objectives". Em: *Journal of Parallel and Distributed Computing* 74.10 (2014), pp. 2918 –2933. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2014.06.007. URL: http://www.sciencedirect.com/science/article/pii/S0743731514001099.

[23] I. Foster, Y. Zhao, I. Raicu e S. Lu. "Cloud Computing and Grid Computing 360-Degree Compared". Em: *2008 Grid Computing Environments Workshop*. 2008, pp. 1–10. DOI: 10.1109/GCE.2008.4738445.

[24] P. Garcia Lopez, A. Montresor, D. Epema, A. Datta, T. Higashino, A. Iamnitchi, M. Barcellos, P. Felber e E. Riviere. "Edge-centric Computing: Vision and Challenges". Em: *SIGCOMM Comput. Commun. Rev.* 45.5 (set. de 2015), pp. 37–42. ISSN: 0146-4833. DOI: 10.1145/2831347.2831354. URL: http://doi.acm.org/10.1145/2831347.2831354.

[25] M. Garriga. "Towards a Taxonomy of Microservices Architectures". Em: fev. de 2018, pp. 203–218. ISBN: 978-3-319-74780-4. DOI: 10.1007/978-3-319-74781-1_15.

[26] H. S. Gill A.Q. "Cloud Monitoring Data Challenges: A Systematic Review". Em: *IEEE Cloud Computing* (2016). ISSN: 978-3-319-46687-3.

[27] S. Haselböck e R. Weinreich. "Decision Guidance Models for Microservice Monitoring". Em: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. 2017, pp. 54–61. DOI: 10.1109/ICSAW.2017.31.

[28] R. Hat. *What is DOCKER?* Accessed: 2019-08-21. 2019. URL: https://www.redhat.com/en/topics/containers/what-is-docker.

[29] IBM. *SOA versus microservices: What's the difference? - Cloud computing news*. Accessed: 2019-01-01. 2018. URL: https://www.ibm.com/blogs/cloud-computing/2018/09/06/soa-versus-microservices/.

[30] A. Iqbal, C. Pattinson e A. Kor. "Performance monitoring of Virtual Machines (VMs) of type I and II hypervisors with SNMPv3". Em: *2015 World Congress on Sustainable Technologies (WCST)*. 2015, pp. 98–99. DOI: 10.1109/WCST.2015.7415127.

[31] Q. D. La, M. V. Ngo, T. Q. Dinh, T. Q. Quek e H. Shin. "Enabling intelligence in fog computing to achieve energy and latency reduction". Em: *Digital Communications and Networks* 5.1 (2019). Artificial Intelligence for Future Wireless Communications and Networking, pp. 3 –9. ISSN: 2352-8648. DOI: https://doi.org/10.1016/j.dcan.2018.10.008. URL: http://www.sciencedirect.com/science/article/pii/S2352864818301081.

[32] A. Lameirinhas. *Monitoring on the Edge*. Rel. téc. FCT-UNL, 2018.

[33] D. Lee, J. Dongarra e R. S. Ramakrishna. "visPerf: Monitoring Tool for Grid Computing". Em: jan. de 2003, pp. 233–243. DOI: 10.1007/3-540-44863-2_24.

[34] J. Lewis. *Microservices*. https://martinfowler.com/articles/microservices.html. Accessed: 2019-01-01. 2014.

[35] G. C. Li. *Networking your docker containers using docker0 bridge*. Accessed: 2019-08-19. 2016. URL: https://developer.ibm.com/recipes/tutorials/networking-your-docker-containers-using-docker0-bridge.

[36] Q. H. Mahmoud. *SOA and Web Services*. Accessed: 2019-01-01. 2005. URL: https://www.oracle.com/technetwork/articles/javase/soa-142870.html.

[37] R. Mahmud, R. Kotagiri e R. Buyya. "Fog Computing: A Taxonomy, Survey and Future Directions". Em: *Internet of Everything: Algorithms, Methodologies, Technologies and Perspectives*. Ed. por B. Di Martino, K.-C. Li, L. T. Yang e A. Esposito. Singapore: Springer Singapore, 2018, pp. 103–130. ISBN: 978-981-10-5861-5. DOI: 10.1007/978-981-10-5861-5_5. URL: https://doi.org/10.1007/978-981-10-5861-5_5.

[38] N. Manohar. "A Survey of Virtualization Techniques in Cloud Computing". Em: *Proceedings of International Conference on VLSI, Communication, Advanced Devices, Signals & Systems and Networking (VCASAN-2013)*. Ed. por V. S. Chakravarthi, Y. J. M. Shirur e R. Prasad. India: Springer India, 2013, pp. 461–470. ISBN: 978-81-322-1524-0.

[39] D. C. Marinescu. *Cloud Computing: Theory and Practice*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013. ISBN: 0124046274, 9780124046276.

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

[41] I. Miell e A. H. Sayers. *Docker in Practice*. 1st. Greenwich, CT, USA: Manning Publications Co., 2016. ISBN: 1617292729, 9781617292729.

[42] S. Newman. *Building Microservices*. First. O'Reilly Media, 2015. ISBN: 978-1-491-95035-7.

[43] M. P. Papazoglou. "Service-oriented computing: concepts, characteristics and directions". Em: *Proceedings of the Fourth International Conference on Web Information Systems Engineering, 2003. WISE 2003.* 2003, pp. 3–12. DOI: 10.1109/WISE.2003.1254461.

[44] N. Poulton. *Docker Deep Dive*. Leanpub, 2018. ISBN: 9781521822807.

[45] Prometheus. Accessed: 2019-01-01. URL: https://prometheus.io/.

[46] M. Pérez e A. Sanchez. "FMonE: A Flexible Monitoring Solution at the Edge". Em: *Wireless Communications and Mobile Computing* 2018 (nov. de 2018). DOI: 10.1155/2018/2068278.

[47] G. Rodrigues, R. N. Calheiros, V. Tavares Guimaraes, G. Lessa dos Santos, M. De Carvalho, L. Granville, L. Tarouco e R. Buyya. "Monitoring of cloud computing environments: concepts, solutions, trends, and future directions". Em: abr. de 2016, pp. 378–383. DOI: 10.1145/2851613.2851619.

[48] S. D. Santis, L. Florez, D. V. Nguyen e E. Rosa. *Evolve the Monolith to Microservices with Java and Node*. First. IBM Corporation, International Technical Support Organization, 2016. ISBN: 0783442119.

[49] A. W. Services. *Characteristics of Microservices - Microservices on AWS*. Accessed: 2019-01-01. URL: https://docs.aws.amazon.com/pt_br/aws-technical-content/latest/microservices-on-aws/characteristics-of-microservices.html.

[50] B. Sosinsky. *Cloud Computing Bible*. 1st. Wiley Publishing, 2011.

[51] A. Souza, N. Cacho, P. P. Jayaraman, R. Ranjan, A. Romanovsky e A. Noor. "Osmotic Monitoring of Microservices between the Edge and Cloud". Em: *2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)* (2018), pp. 758–765.

[52] H. J. Syed, A. Gani, R. W. Ahmad, M. K. Khan e A. I. A. Ahmed. "Cloud monitoring: A review, taxonomy, and open research issues". Em: *Journal of Network and Computer Applications* 98 (2017), pp. 11 –26. ISSN: 1084-8045. DOI: https://doi.org/10.1016/j.jnca.2017.08.021. URL: http://www.sciencedirect.com/science/article/pii/S1084804517302783.

[53] 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, pp. 20–26. DOI: 10.1109/SmartCloud.2016.18.

[54] M. Villari, M. Fazio, S. Dustdar, O. Rana e R. Ranjan. "Osmotic Computing: A New Paradigm for Edge/Cloud Integration". Em: *IEEE Cloud Computing* 3.6 (2016), pp. 76–83. ISSN: 2325-6095. DOI: 10.1109/MCC.2016.124.

[55] vmware. *Virtualization*. Accessed: 2019-08-21. 2019. URL: https://www.vmware.com/solutions/virtualization.html.

[56] X. Zhao, J. Yin, C. Zhi e Z. Chen. "SimMon: a toolkit for simulation of monitoring mechanisms in cloud computing environment". Em: *Concurrency and Computation: Practice and Experience* 29.1 (). e3832 cpe.3832, e3832. DOI: 10.1002/cpe.3832. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.3832. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.3832.

APÊNDICE

A

Métricas recolhidas

Tabela A.1: Coletores de métricas ativados por configuração padrão (Node Exporter)

Name	Description	OS
arp	Exposes ARP statistics from /proc/net/arp.	Linux
bcache	Exposes bcache statistics from /sys/fs/bcache/.	Linux
bonding	Exposes the number of configured and active slaves of Linux bonding interfaces.	Linux
boottime	Exposes system boot time derived from the kern.boottime sysctl.	Darwin, Dragonfly, FreeBSD, NetBSD, OpenBSD, Solaris
conntrack	Shows conntrack statistics (does nothing if no /proc/sys/net/netfilter/ present).	Linux
cpu	Exposes CPU statistics	Darwin, Dragonfly, FreeBSD, Linux, Solaris
cpufreq	Exposes CPU frequency statistics	Linux, Solaris
diskstats	Exposes disk I/O statistics.	Darwin, Linux, OpenBSD
edac	ExposesExposes error detection and correction statistics.	Linux
entropy	Exposes available entropy.	Linux
exec	Exposes execution statistics.	Dragonfly, FreeBSD
filefd	Exposes file descriptor statistics from /proc/sys/fs/file-nr.	Linux
filesystem	Exposes filesystem statistics, such as disk space used.	Darwin, Dragonfly, FreeBSD, Linux, OpenBSD
hwmon	Expose hardware monitoring and sensor data from /sys/class/hwmon/.	Linux
infiniband	Exposes network statistics specific to InfiniBand and Intel OmniPath configurations.	Linux
ipvs	Exposes IPVS status from /proc/net/ip_vs and stats from /proc/net/ip_vs_stats.	Linux
loadavg	Exposes load average.	Darwin, Dragonfly, FreeBSD, Linux, NetBSD, OpenBSD, Solaris
mdadm	Exposes statistics about devices in /proc/mdstat (does nothing if no /proc/mdstat present).	Linux
meminfo	Exposes memory statistics.	Darwin, Dragonfly, FreeBSD, Linux, OpenBSD
netclass	Exposes network interface info from /sys/class/net/	Linux
netdev	Exposes network interface statistics such as bytes transferred.	Darwin, Dragonfly, FreeBSD, Linux, OpenBSD
netstat	Exposes network statistics from /proc/net/netstat. This is the same information as netstat -s.	Linux
nfs	Exposes NFS client statistics from /proc/net/rpc/nfs. This is the same information as nfsstat -c.	Linux
nfsd	Exposes NFS kernel server statistics from /proc/net/rpc/nfsd. This is the same information as nfsstat -s.	Linux
pressure	Exposes pressure stall statistics from /proc/pressure/.	Linux (kernel 4.20+ and/or CONFIG_PSI)
schedstat	Exposes task scheduler statistics from /proc/schedstat.	Linux
sockstat	Exposes various statistics from /proc/net/sockstat.	Linux
stat	Exposes various statistics from /proc/stat. This includes boot time, forks and interrupts.	Linux

Tabela A.2: Continuação dos coletores de métricas ativados por configuração padrão (Node Exporter)

Name	Description	OS
textfile	Exposes statistics read from local disk. The –collector.textfile.directory flag must be set.	any
thermal_zone	Exposes thermal zone & cooling device statistics from /sys/class/thermal.	Linux
time	Exposes the current system time.	any
timex	Exposes selected adjtimex(2) system call stats.	Linux
uname	Exposes system information as provided by the uname system call.	Darwin, FreeBSD, Linux, OpenBSD
vmstat	Exposes statistics from /proc/vmstat.	Linux
xfs	Exposes XFS runtime statistics.	Linux (kernel 4.4+)
zfs	Exposes ZFS performance statistics.	Linux, Solaris

Tabela A.3: Coletores de métricas desativados por configuração padrão (Node Exporter)

Name	Description	OS
buddyinfo	Exposes statistics of memory fragments as reported by /proc/buddyinfo.	Linux
devstat	Exposes device statistics	Dragonfly, FreeBSD
drbd	Exposes Distributed Replicated Block Device statistics (to version 8.4)	Linux
interrupts	Exposes detailed interrupts statistics.	Linux, OpenBSD
ksmd	Exposes kernel and system statistics from /sys/kernel/mm/ksm.	Linux
logind	Exposes session counts from logind.	Linux
meminfo_numa	Exposes memory statistics from /proc/meminfo_numa.	Linux
mountstats	Exposes filesystem statistics from /proc/self/mountstats. Exposes detailed NFS client statistics.	Linux
ntp	Exposes local NTP daemon health to check time	any
processes	Exposes aggregate process statistics from /proc.	Linux
qdisc	Exposes queuing discipline statistics	Linux
runit	Exposes service status from runit.	any
supervisord	Exposes service status from supervisord.	any
systemd	Exposes service and system status from systemd.	Linux
tcpstat	Exposes TCP connection status information from /proc/net/tcp and /proc/net/tcp6. (Warning: the current version has potential performance issues in high load situations.)	Linux
wifi	Exposes WiFi device and station statistics.	Linux
perf	Exposes perf based metrics (Warning: Metrics are dependent on kernel configuration and settings).	Linux

Tabela A.4: Métricas do nível aplicacional expostas pelo Go

Name	Type	Description
go_memstats_heap_inuse_bytes	Gauge	Number of heap bytes that are in use
go_memstats_heap_sys_bytes	Gauge	Number of heap bytes obtained from system
go_memstats_stack_inuse_bytes	Gauge	Number of bytes in use by the stack allocator.
process_resident_memory_bytes	Gauge	Resident memory size in bytes.
go_goroutines	Gauge	Number of goroutines that currently exist.
go_memstats_gc_sys_bytes	Gauge	Number of bytes used for garbage collection system metadata.
go_memstats_heap_released_bytes	Gauge	Number of heap bytes released to OS.
go_memstats_last_gc_time_seconds	Gauge	Number of seconds since 1970 of last garbage collection.
go_memstats_mspan_inuse_bytes	Gauge	Number of bytes in use by mspan structures.
go_memstats_sys_bytes	Gauge	Number of bytes obtained from system.
go_memstats_lookups_total	Counter	Total number of pointer lookups.
go_memstats_mcache_sys_bytes	Gauge	Number of bytes used for mcache structures obtained from system.
go_memstats_mspan_sys_bytes	Gauge	Number of bytes used for mspan structures obtained from system.
process_start_time_seconds	Gauge	Start time of the process since unix epoch in seconds.
go_memstats_frees_total	Counter	Total number of frees.
go_memstats_heap_alloc_bytes	Gauge	Number of heap bytes allocated and still in use
go_memstats_mallocs_total	Counter	Total number of mallocs.
go_memstats_mcache_inuse_bytes	Gauge	Number of bytes in use by mcache structures.
go_memstats_stack_sys_bytes	Gauge	Number of bytes obtained from system for stack allocator.
go_memstats_heap_objects	Gauge	Number of allocated objects."
process_open_fds	Gauge	Number of open file descriptors.
go_memstats_heap_idle_bytes	Gauge	Number of heap bytes waiting to be used.
process_cpu_seconds_total	Counter	Total user and system CPU time spent in seconds.
process_max_fds	Gauge	Maximum number of open file descriptors.
go_gc_duration_seconds	Summary	A summary of the GC invocation durations.
go_memstats_alloc_bytes	Gauge	Number of bytes allocated and still in use.
go_memstats_alloc_bytes_total	Counter	Total number of bytes allocated, even if freed.
go_memstats_buck_hash_sys_bytes	Gauge	Number of bytes used by the profiling bucket hash table.
go_memstats_next_gc_bytes	Gauge	Number of heap bytes when next garbage collection will take place.
go_memstats_other_sys_bytes	Gauge	Number of bytes used for other system allocations.
process_virtual_memory_bytes	Gauge	Virtual memory size in bytes.

Listagem A.1: Exemplo de resposta com as métricas retornadas pelo Docker stats do tipo *container* e que podem ser disponibilizadas pelo cScraper ou EdgeMon

```
{
    "read": "2016-02-07T13:26:56.142981314Z",
    "precpu_stats": {
        "cpu_usage": {
            "total_usage": 0,
            "percpu_usage": null,
            "usage_in_kernelmode": 0,
            "usage_in_usermode": 0
        },
```

```
        "system_cpu_usage": 0,
        "throttling_data": {
            "periods": 0,
            "throttled_periods": 0,
            "throttled_time": 0
        }
    },
    "cpu_stats": {
        "cpu_usage": {
            "total_usage": 242581854769,
            "percpu_usage": [242581854769],
            "usage_in_kernelmode": 33910000000,
            "usage_in_usermode": 123040000000
        },
        "system_cpu_usage": 3367860000000,
        "throttling_data": {
            "periods": 0,
            "throttled_periods": 0,
            "throttled_time": 0
        }
    },
    "memory_stats": {
        "usage": 693821440,
        "max_usage": 818733056,
        "stats": {
            "active_anon": 282038272,
            "active_file": 28938240,
            "cache": 82534400,
            "hierarchical_memory_limit": 9223372036854771712,
            "hierarchical_memsw_limit": 9223372036854771712,
            "inactive_anon": 329543680,
            "inactive_file": 53284864,
            "mapped_file": 26558464,
            "pgfault": 809513,
            "pgmajfault": 2559,
            "pgpgin": 1015608,
            "pgpgout": 940757,
            "rss": 611270656,
            "rss_huge": 136314880,
            "swap": 249049088,
            "total_active_anon": 282038272,
            "total_active_file": 28938240,
            "total_cache": 82534400,
            "total_inactive_anon": 329543680,
            "total_inactive_file": 53284864,
            "total_mapped_file": 26558464,
            "total_pgfault": 809513,
            "total_pgmajfault": 2559,
            "total_pgpgin": 1015608,
            "total_pgpgout": 940757,
```

```
            "total_rss": 611270656,
            "total_rss_huge": 136314880,
            "total_swap": 249049088,
            "total_unevictable": 0,
            "total_writeback": 0,
            "unevictable": 0,
            "writeback": 0
        },
        "failcnt": 0,
        "limit": 1044574208
    },
    "blkio_stats": {
        "io_service_bytes_recursive": [{
                "major": 8,
                "minor": 0,
                "op": "Read",
                "value": 301649920
            }, {
                "major": 8,
                "minor": 0,
                "op": "Write",
                "value": 248315904
            }, {
                "major": 8,
                "minor": 0,
                "op": "Sync",
                "value": 201003008
            }, {
                "major": 8,
                "minor": 0,
                "op": "Async",
                "value": 348962816
            }, {
                "major": 8,
                "minor": 0,
                "op": "Total",
                "value": 549965824
            }],
        "io_serviced_recursive": [{
                "major": 8,
                "minor": 0,
                "op": "Read",
                "value": 41771
            }, {
                "major": 8,
                "minor": 0,
                "op": "Write",
                "value": 72796
            }, {
                "major": 8,
```

```
                "minor": 0,
                "op": "Sync",
                "value": 61246
            }, {
                "major": 8,
                "minor": 0,
                "op": "Async",
                "value": 53321
            }, {
                "major": 8,
                "minor": 0,
                "op": "Total",
                "value": 114567
            }],
        "io_queue_recursive": [],
        "io_service_time_recursive": [],
        "io_wait_time_recursive": [],
        "io_merged_recursive": [],
        "io_time_recursive": [],
        "sectors_recursive": []
    },
    "pids_stats": {},
    "networks": {
        "eth0": {
            "rx_bytes": 40192,
            "rx_packets": 285,
            "rx_errors": 0,
            "rx_dropped": 0,
            "tx_bytes": 222138,
            "tx_packets": 150,
            "tx_errors": 0,
            "tx_dropped": 0
        }
    }
}
```