Mestrado em Engenharia Informática
Dissertação
Relatório Final

Monitoria de Arquiteturas de Micro-serviços

Fábio Figueiredo Pina
fpina@student.dei.uc.pt

Orientador:
Prof. Dr. Filipe João Boavida Mendonça Machado de Araújo

Co-Orientadores:
Prof. Dr. Rui Pedro Pinto de Carvalho e Paiva
Prof. Dr. António Jorge Silva Cardoso

Data: 3 de Setembro de 2018

• U C •
FCTUC FACULDADE DE CIÊNCIAS
E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

Agradecimentos

Quero começar por agradecer ao meu orientador, o Professor Filipe Araújo por todo o apoio prestado, mesmo em férias. Era impossível ter concluido este trabalho sem a sua ajuda.

Também queria agradecer aos meus co-orientadores, o Professor Rui Paiva e o Professor Jorge Cardoso, pelos valiosos conselhos que me deram.

Tenho de dizer um grande obrigado ao Jaime Correia, pela orientação em todo o trabalho em termos tecnológicos. Sem dúvida que saio deste projeto com muitos mais conhecimentos.

Queria deixar também um agradecimento ao Ricardo Filipe por todos os conselhos durante a fase de implementação.

Queria agradecer ao resto dos membros do grupo de investigação, Fábio Ribeiro, Bruno Lopes e Artur Pedroso, pelos conselhos e pela ajuda prestada sempre que foi preciso.

Tenho também de agradecer à minha família, pelo apoio incondicional que me deram.

Por fim, queria agradecer a todos os meus amigos, que me deram sempre incentivo para continuar a trabalhar.

Abstract

Breaking large software systems into smaller functionally interconnected components is a trend on the rise. This architectural style, known as "microservices", simplifies development, deployment and management at the expense of complexity and observability. In fact, in large scale systems, it is particularly difficult to determine the set of microservices responsible for delaying a client's request, when one module impacts several other microservices in a cascading effect. Components cannot be analyzed in isolation, and without instrumenting their source code extensively, it is difficult to find the bottlenecks and trace their root causes. To mitigate this problem, we propose a much simpler approach: log gateway activity, to register all calls to and between microservices, as well as their responses, thus enabling the extraction of topology and performance metrics, without changing source code. For validation, we implemented the proposed platform, with a microservices-based application that we observe under load. Our results show that we can extract relevant performance information with a negligible effort.

Keywords

microservices, API gateway, black-box monitoring

Resumo

Uma das tendências mais recentes nos sistemas distribuídos é a de subdividir grandes componentes de software em pedaços mais pequenos. Este paradigma é conhecido por "micro-serviços" e, embora simplifique o desenvolvimento, instalação e gestão do software, torna o sistema mais complexo e bastante mais difícil de observar, dado o grande número de interações envolvidas. Por esta razão, num sistema de grandes dimensões, é particularmente difícil saber quais os componentes que mais contribuem para o tempo de espera medido pelos utilizadores. Por um lado, estes componentes não podem ser analisados separadamente; por outro, sem "instrumentar" extensivamente o código fonte é difícil relacioná-los para identificar a origem de estrangulamentos. Para mitigar este problema propomos uma abordagem bem mais simples: usando a *gateway* de acesso aos micro-serviços registamos todos as invocações que lhes são feitas, bem como todas as respostas, extraindo assim o relacionamento entre serviços e o respetivo desempenho. Para validar este método, simulamos a invocação de serviços concretos duma implementação real de uma aplicação. Os resultados mostram que é possível extrair a informação de desempenho mais relevante no sistema a um baixo custo.

Palavras-Chave

micro-serviços, API gateway, monitoria de caixa-preta

Conteúdo

Acrónimos

API Application Programming Interface. 4

APM Application Performance Monitoring. 9

AWS Amazon Web Services. 18

CLI Command-line Interface. 15

CPU Central Processing Unit. 1

GB Gigabyte. 11

HTML Hypertext Markup Language. 6

HTTP Hypertext Transfer Protocol. 6

IPM Infrastructure Performance Monitoring. 9

MB Megabyte. 11

MS Microservice. 38

NCPA Nagios cross Platform Agent. 19

NRDP Nagios Remote Data Processor. 19

NRPE Nagios Remote Plugin Executor. 19

RAM Random Access Memory. 38

REST Representational State Transfer. 15

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Metodologias de arquitetura e desenvolvimento baseadas em micro-serviços têm, nos tempos correntes, vindo a tornar-se numa prática comum para o desenvolvimento de sistemas. Este novo paradigma teve como origem diversos fatores. Primeiro, o facto de os sistemas monolíticos serem complexos e difíceis de escalar, principalmente em sistemas de grande escala. Assim, surgiu a necessidade de desagregar estes sistemas em módulos orientados à função e escaláveis individualmente. Em segundo, surgiram tendências como *containerization* para a instalação e operação de sistemas. Adicionalmente, do ponto de vista de engenharia de software, este paradigma foi motivado pelas metodologias ágeis de trabalho com o foco em pequenas equipas responsáveis transversalmente pelo ciclo de vida do módulo. Estas razões fazem dos micro-serviços um paradigma que traz grandes vantagens no desenvolvimento de sistemas distribuídos de grande escala e altamente disponíveis.

Por outro lado, esta nova forma de desenvolver sistemas, trouxe novos desafios na área da monitoria e operação. Os sistemas legados monolíticos, eram mais fáceis de monitorizar. O âmbito da monitoria estava restrito ao conjunto de máquinas que podiam estar a provocar anomalias de desempenho ao monólito. Com este novo paradigma, os administradores têm que validar sistemas de arquitetura dinâmica, com centenas ou milhares de máquinas e serviços, alterados regularmente, de forma a identificar a origem e mitigar rapidamente qualquer anomalia. Este aumento de complexidade, torna a monitorização bem mais complicada para os operadores, trazendo um grau adicional de incerteza e risco.

Algumas das plataformas de monitoria existentes atualmente, podem também usar-se em micro-serviços. Nesta área temos as ferramentas de monitoria de sistemas como o **Nagios** [30] ou **Zabbix** [52], que analisam diversas métricas dos sistemas como a Central Processing Unit (CPU) e a memória. Além disso, permitem o envio de alarmes em resposta ao incumprimento de limiares, por exemplo, na ocupação duma CPU ou num tempo de resposta. No entanto fazem uso de instrumentação ou agentes para a coleta das diversas métricas.

Outras plataformas como o **Kibana** [24] permite a coleta de logs, incorporando a criação de gráficos e de *dashboards* para a visualização das métricas. Existem outras ferramentas com potencialidades semelhantes como o **Grafana** [18], que permitem a visualização e análise de dados tal como tempos de resposta dos serviços.

A abordagem de *tracing*, requer a instrumentação dos micro-serviços de forma a permitir a propagação pelo sistema de uma identificador. Este identificador de correlação permite determinar o fluxo de um pedido específico através de vários micro-serviços. Temos como exemplos deste tipo de ferramentas o **Zipkin** [53], o **Opentracing** [37] ou o **Dapper**

[47]. Apesar das potencialidades das ferramentas supra mencionadas, qualquer uma delas força a que exista instrumentação dos micro-serviços. Cada micro-serviço é responsável por enviar a informação para um ponto central de coleta, processamento e agregação dos dados. Por exemplo, caso quiséssemos determinar a causalidade entre invocações num fluxo de operação, seria necessário instrumentar todos os micro-serviços nessa cadeia.

Com base no que foi referido, temos um sistema em que cada módulo tem as funções bem definidas e delimitadas, mas em que a sua monitoria é transversal e espalhada por todo o sistema. Ou seja, embora as metodologias para a criação de um sistema distribuído tenham evoluído, as técnicas de análise e monitoria não evoluíram ao mesmo passo.

Neste documento apresentamos uma abordagem "caixa-negra", em que coletamos diversas métricas, tal como o tempo de resposta, a origem e destino dos pedidos efetuados entre os micro-serviços. É uma solução que não obriga a adaptações nos micro-serviços e, é desta forma, muito menos invasiva.

1.1 Motivação

A monitoria nos sistemas monolíticos era facilmente endereçada, pois estando os seus módulos na mesma máquina, a origem dos problemas era mais restrita. Sistemas baseados em micro-serviços surgiram do ecossistema criado por novas técnicas de desenvolvimento, como os métodos ágeis, e novas formas de instanciação do software como contentores. No entanto, a monitoria do sistema como um todo não acompanhou estas novas técnicas. Prova disso é o facto de empresas de referência na industria, tal como a Netflix, terem adotado soluções personalizadas e específicas para as suas necessidades, como o **Vector** [34] e o **Atlas** [31]. Isto revela que a monitoria de um sistema de micro-serviços é um tema complexo e de difícil resolução.

A técnica normal é instrumentar o máximo de camadas possíveis do sistema, desde as máquinas até à própria aplicação. Ficamos assim com um sistema de monitoria "pesado", com uma elevada manutenção e fortemente acoplado ao sistema a monitorizar.

1.2 Objetivos

O nosso principal objetivo é criar uma solução de monitoria que resolva os problemas da abordagem que requer instrumentação.

Pretendemos que a nossa solução seja desacoplada do sistema de micro-serviços a analisar e que reúna o máximo de informação possível sobre a aplicação de micro-serviços. Queremos fazer uma abordagem de "caixa-negra" onde seja possível inferir sobre o desempenho da aplicação, assim como a sua topologia.

1.3 Planeamento e Ciclo de Vida

Visto que isto é um trabalho exploratório nós não seguimos nenhuma metodologia de desenvolvimento de *software*. Tivemos reuniões duas vezes por mês para discutir o estado do trabalho e possíveis direções a tomar a partir desse ponto.

A figura 1.1 mostra o nosso planeamento para o primeiro semestre através de um diagrama

de *Gantt*:

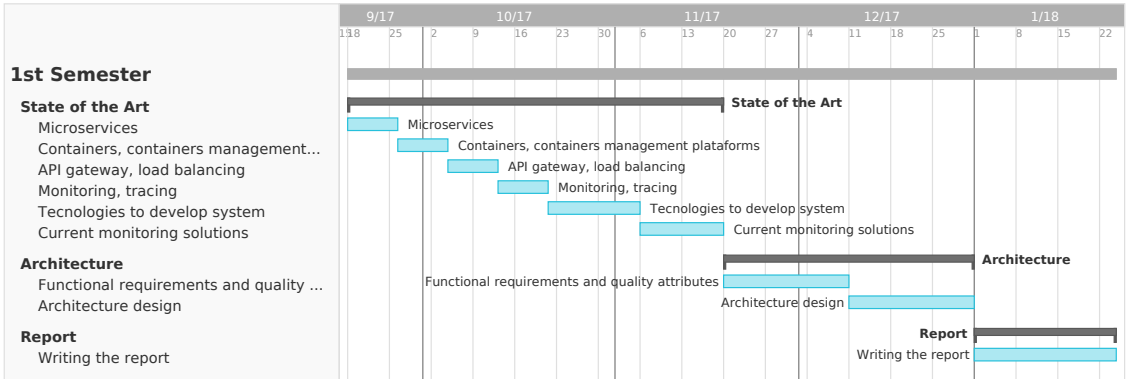

Figura 1.1: Diagram de *Gantt* do 1º semestre

No primeiro semestre começámos por fazer um estudo da arte, o qual demorou 9 semanas. Durante esse tempo aprendemos sobre diversos conceitos, como micro-serviços, contentores e técnicas de monitoria.

As próximas 3 semanas foram dedicadas à arquitetura. Primeiro tirámos requisitos funcionais e atributos de qualidade para a nossa plataforma, depois desenhámos a sua arquitetura.

As últimas 3 semanas foram dedicadas a escrever o relatório intermédio.

A figura 1.2 mostra o nosso planeamento para o segundo semestre.

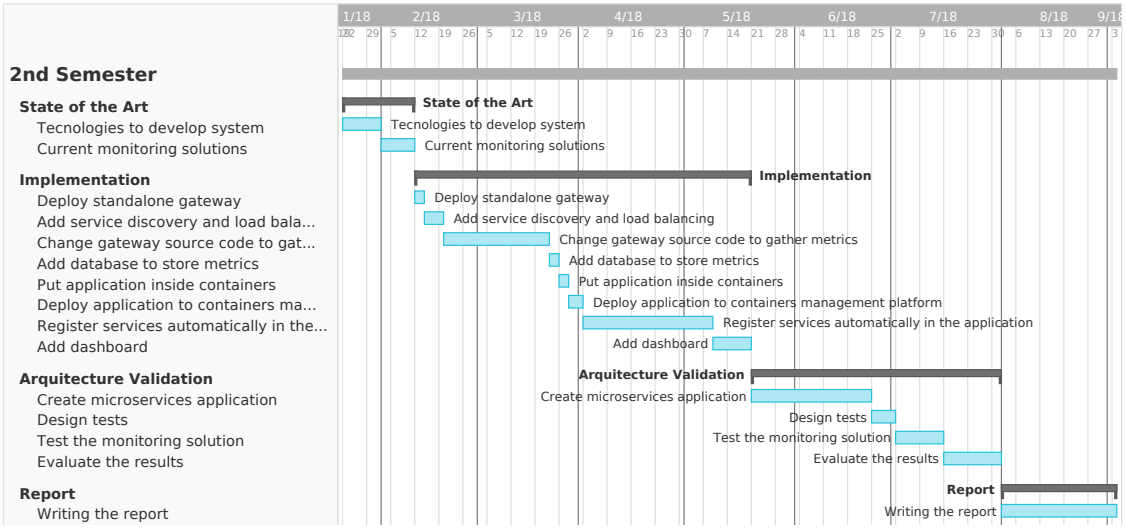

Figura 1.2: Diagram de *Gantt* do 2º semestre

As primeiras duas semanas do segundo semestre foram dedicadas à conclusão do estudo da arte. Por falta de tempo estas tarefas não foram totalmente realizadas no primeiro semestre. Pesquisámos sobre as ferramentas que iamos usar para implementar o nosso sistema e estudámos as soluções de monitoria já existentes.

Nos 3 meses seguintes implementámos a nossa plataforma de monitoria. Começámos por implementar a *gateway* e colocá-la a funcionar. Depois de isto estar feito adicionámos descoberta de serviços e um balanceador de carga. Estas tarefas demoraram um pouco

mais de 1 semana. Nas 5 semanas seguintes alterámos o código fonte da *gateway* de modo a conseguirmos recolher métricas dos pedidos que passavam por lá. Depois adicionámos uma base de dados para guardar estas métricas. Os próximos passos foram colocar todos estes componentes em contentores, e colocá-los numa plataforma de gestão de contentores. Durante as 5 semanas seguintes dedicámo-nos a resolver o problemas de registar serviços automaticamente na nossa plataforma. O último passo foi a adição uma *dashboard* para a visualização dos dados.

Os próximos 2 meses foram dedicados a validar a nossa arquitetura. Começámos por fazer uma aplicação de micro-serviços para efeitos de teste, e de seguida realizámos testes.

Por fim, no último mês escrevemos o relatório final.

1.4 Estrutura do Documento

A estrutura do documento é a seguinte: O capítulo 2 é dedicado ao estudo da arte. Começamos por explorar os conceitos ligados ao tema do nosso trabalho, como micro-serviços, contentores, Application Programming Interface (API) *gateway* e monitoria. De seguida analisamos as ferramentas usadas na criação da nossa plataforma e terminamos com a apresentação de tecnologias de monitoria já existentes.

No capítulo 3 descrevemos o problema com detalhe e o método que usamos para o resolver.

O próximo capítulo, 4, inclui a definição de requisitos funcionais e atributos de qualidade para a nossa plataforma. Terminamos com a apresentação da nossa proposta para a arquitetura.

No capítulo 5 descrevemos tudo o que foi implementado na plataforma de monitoria.

O capítulo 6 é dedicado à validação da nossa arquitetura. Começamos por apresentar a configuração experimental usada para os testes. De seguida realizamos os testes para validar a nossa arquitetura. No fim discutimos os resultados que conseguimos retirar com a solução de monitoria que construímos.

Acabamos no capítulo 7, onde retiramos conclusões e discutimos os passos que se podem seguir para trabalho futuro.

Capítulo 2

Estado da Arte

Nesta secção vamos começar por abordar conceitos relacionados com monitoria de micro-serviços, de seguida vamos explicar as tecnologias usadas para a implementação do sistema, e por fim vamos discutir as alternativas existentes no mercado no que toca a soluções de monitoria.

2.1 Conceitos

2.1.1 Micro-serviços

O estilo arquitetural baseado em micro-serviços tem vindo a crescer em popularidade nos últimos anos. É uma abordagem que consiste em desenvolver uma única aplicação usando um conjunto de pequenos serviços, em que cada um deles executa no seu processo e comunicam entre si usando usando mecanismos como Hypertext Transfer Protocol (HTTP). Estes serviços são construídos em torno da lógica de negócio e podem ser implantados independentemente.

Antes de entrarmos em mais detalhes sobre micro-serviços, é útil apresentar o estilo monolítico. Uma aplicação que segue o estilo monolítico é construída como uma única unidade. Vejamos o exemplo de aplicações *enterprise*: estas aplicações são geralmente divididas em três partes principais, uma *interface* gráfica do lado do utilizador (páginas Hypertext Markup Language (HTML) e *javascript* a correr no navegador de internet), uma base de dados (consiste em várias tabelas inseridas geralmente numa base de dados relacional) e um servidor. O servidor vai lidar com pedidos HTTP, executar operações lógicas, gerir toda a informação presente na base de dados e popular as páginas HTML do navegador de internet. Ora, este servidor é um monólito, um único executável lógico.

Usar um servidor monólito é uma abordagem natural para um sistema deste tipo. Toda a lógica para o tratamento de um pedido executa num único processo, permitindo usar funcionalidades da linguagem para dividir a aplicação em classes ou funções. É depois possível escalar o monólito horizontavelmente, executando várias instâncias atrás de um balanceador de carga. [26]

No entanto, esta abordagem carrega muitas limitações. Consideremos o cenário em que este servidor se transforma num monólito de enormes dimensões. Isto vai exigir uma interação muito mais complexa entre as equipas de desenvolvimento. Visto que a aplicação está a executar num único processo, caso exista uma falha no sistema toda a aplicação vai

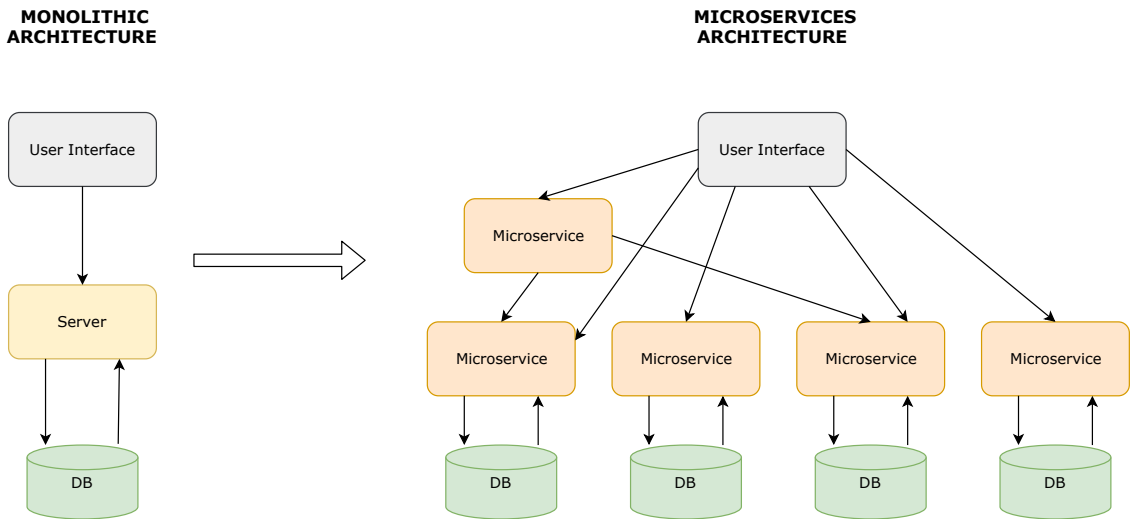

Figura 2.1: Abordagem de monólito e de micro-serviços

abaixo. Estamos também presos às decisões tecnológicas feitas no início, pois é extremamente caro em termos de tempo e dinheiro reescrever toda a aplicação. [43]

Os micro-serviços surgiram para fazer face a estes problemas. Eles permitiram a divisão dos módulos presentes no monólito em diferentes serviços, cada um com a sua arquitetura e lógica de negócio. Esta transformação está representada na figura 2.1.

Uma arquitetura de micro-serviços não tem uma definição formal, no entanto tem um conjunto de caraterísticas que deve seguir. Essas caraterísticas vão ser apresentadas seguidamente: [27]

- **Descentralizados -** Arquiteturas de micro-serviços são sistemas distribuídos com gestão de dados descentralizada. Em vez de usarem uma base de dados única, cada um tem a sua base de dados com a respetiva informação. Os micro-serviços também são descentralizados no sentido em que são desenvolvidos e geridos.
- **Poliglotas -** Cada micro-serviço é construido consoante as suas necessidades. As equipas de desenvolvimento têm liberdade para escolher a ferramenta que melhor resolve o problema. Isto resulta numa abordagem heterogénea, onde são escolhidas diferentes linguagens de programação, base de dados ou ferramentas.
- **Independentes -** Serviços diferentes de uma arquitetura de micro-serviços podem ser alterados ou melhorados independentemente sem afetar o funcionamento de outros serviços.
- **Têm uma responsabilidade -** Cada micro-serviço deve ter uma única responsabilidade, um domínio dentro da aplicação geral. Deve fazer só uma coisa independentemente do tamanho que possa tomar. Não há um tamanho definido que todos os micro-serviços devam seguir.
- **Caixa negra -** Cada micro-serviço é desenhado como uma caixa negra, ou seja, escondem os detalhes da sua complexidade dos outros serviços. A comunicação entre serviços é realizada por *APIs* bem definidas.

As grandes vantagens dos micro-serviços são de facto a agilidade de processos e o potencial

de escalabilidade. Os micro-serviços permitem a existência de equipas pequenas e independentes por cada serviço. Estas equipas atuam dentro de um contexto pequeno e bem definido, o que faz com que os ciclos de desenvolvimento sejam mais pequenos. A figura 2.2 representa os dois tipos de desenvolvimento: à esquerda equipas pequenas trabalham em vários serviços e à direita temos uma grande equipa que trabalha num monólito. Os micro-serviços também têm mais potencial no que toca à escalabilidade. Estes serviços podem ser escalados horizontavelmente e de forma ótima, ou seja, só são replicados os serviços necessários. Enquanto que no monólito, toda a aplicação necessitava de ser replicada.

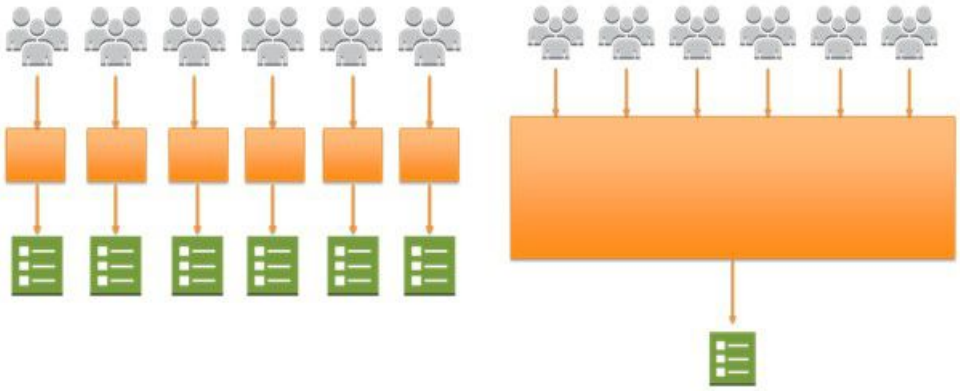

Figura 2.2: Agilidade de processos do monólito e micro-serviços [27]

Os maiores desafios deste estilo arquitetural advêm do facto dos micro-serviços serem essencialmente um sistema distribuido. Um desses problemas é a falta de *standards* no que toca à monitoria.

2.1.2 Monitoria

Monitoria consiste no ato de observar e seguir operações e atividades de utilizadores, aplicações ou serviços de rede num sistema computacional. Permite supervisionar os processos gerais que são realizados no sistema, de modo a dar informação relevante aos administradores. [28]

Aplicações desenvolvidas usando micro-serviços precisam de ser monitorizadas pelas mesmas razões que qualquer outro sistema. Primeiro, devido a potenciais falhas que possam acontecer no sistema, segundo por questões de desempenho. Sistemas complexos, incluindo os monólitos, conseguem continuar a funcionar num estado degradado que tem impacto no desempenho. Isto normalmente indica que uma possível falha no sistema está iminente. Monitorizar o comportamento dos sistemas pode alertar os administradores antes que o sistema falhe por completo.

Monitorizar sistemas ao longo do tempo também pode produzir informação valiosa. Os dados podem ser analisados em busca de padrões em caso de falhas do sistema, de modo a ser correlacionado com eventos específicos. [48]

No entanto, monitorizar arquiteturas de micro-serviços é diferente de monitorizar monólitos. Os monólitos são implementados usando um único executável e usam uma única linguagem de programação. Para monitorizar este tipo de aplicações usa-se uma combinação de ferramentas de Infrastructure Performance Monitoring (IPM) e Application Performance Monitoring (APM). As ferramentes de APM permitem uma análise bastante profunda de problemas relacionados com o código, enquanto que ferramentas de IPM permitem relacionar a infraestrutura com a aplicação. [3]

Por outro lado, os micro-serviços são implementados usando uma família de serviços independentes. Cada serviço tem um função específica para desempenhar e podem comunicar com outros serviços. Consequente, para monitorizar uma arquitetura de micro-serviços, além de se observar o desempanho que cada serviço individualmente, também é necessário observar as interações entre todos os serviços da aplicação. Uma técnica bastante usada para monitorizar micro-serviços é o *tracing*.

Tracing **[44]**

O *tracing* é um método usado para monitorizar aplicações, especialmente aplicações compostas por micro-serviços. Esta técnica permite localizar falhas e causas para a degradação do desempenho, no entanto requer a adição de instrumentação ao código da aplicação. A instrumentação fornece informação de modo a que o administrador consiga analisar quer o desempenho quer o fluxo de operações da aplicação.

Esta técnica é parecida com APM. Uma ferramenta organiza, processa e gera visualizações das métricas dos pedidos, criando uma imagem de como a aplicação se comporta. No entanto vai mais além, é possível seguir os pedidos através de cada serviço. Isto dá uma ideia bastante clara sobre o comportamento de cada instância.

O *tracing* usa unidades básicas denominadas de *traces* e *spans*. Um *trace* representa todo o percurso de um pedido enquanto ele passa pelos serviços do sistema. Um *span* é uma atividade ou operação que ocorre num serviço específico do sistema. Cada *trace* é composto por um conjunto de *spans*. Através da análise de cada *span* e dos *traces* é possível obter uma imagem global do percurso do pedido.

No entanto estes dados não são processados em tempo real, são armazenados numa local-

ização central e analisados sob demanda. Esta técnica permite um conhecimento profundo do comportamento da aplicação, mas é difícil de implementar e de manter.

2.1.3 Contentores

Os contentores apareceram para resolver o problema de executar o *software* de forma confiável quando movido de um ambiente computacional para outro. Quer seja desde um ambiente de desenvolvimento para um ambiente de teste, ou de um sistema operativo para outro. [45]

Um contentor é um pacote de *software* leve e autónomo que contém tudo o que é necessário para executar uma aplicação: código, ferramentas e bibliotecas do sistema e definições. [49]

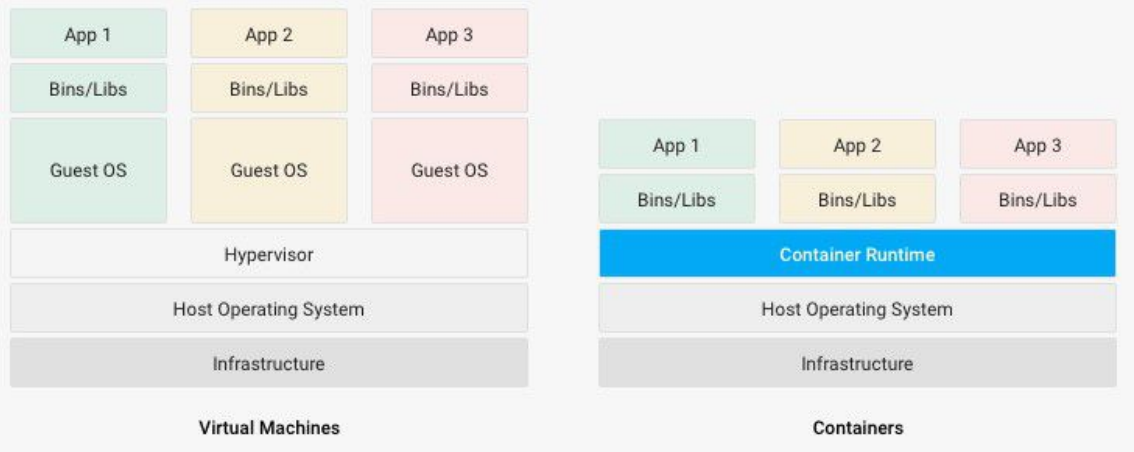

Figura 2.3: Virtual Machines vs Contentores [7]

São uma abordagem mais leve quando comparados com máquina virtuais. Podemos ver as suas diferenças através da figura 2.3. Ambas utilizam virtualização para tirar vantagem dos recursos físicos das máquinas, sendo a maior diferença entre eles o nível de virtualização. As diferenças são apresentadas com mais detalhe na tabela 2.1.

Tabela 2.1: Diferenças entre contentores e máquinas virtuais [1][22]

Contentores	**Máquinas Virtuais**
Tamanho na ordem dos MBs	Tamanho na ordem dos GBs
Virtualizam o sistema operativo	Virtualizam o *hardware*
Contentores no mesmo servidor partilham o sistema operativo do anfitrião	Máquinas virtuais no mesmo servidor têm sistemas operativos diferentes
Partilham o *kernel* do sistema operativo do anfitrião para aceder ao *hardware*	Usam hipervisores para partilhar e gerir o *hardware*

Em resumo, os contentores têm as seguintes caraterísticas: [7]

- **Ambiente consistente -** Os contentores dão capacidade ao programador de criar ambientes isolados de outras aplicações. Garantem consistência independentemente do ambiente onde são implantados.

- **Executam em qualquer lugar -** Os contentores podem ser executados em praticamente qualquer lugar, facilitando o desenvolvimento e implantação em diferentes sistemas operativos, em máquinas virtuais, centros de dados ou na *cloud*.
- **Isolamento -** Os contentores virtualizam recursos de CPU, memória, disco e rede ao nível do sistema operativo, oferencendo ao programador uma visualização isolada logicamente de outras aplicações.

Em suma, os contentores são bastante adequados para implementar uma arquitetura baseada em micro-serviços, onde cada serviço pode ser implantado no seu respetivo contentor e facilmente ligados entre si.

Plataformas de Gestão de Contentores

As plataformas de gestão de contentores ajudam os utilizadores a observar contentores induviduais, incluindo as suas versões e ligações. Estas soluções melhoram o potencial dos contentores, simplificando alguns aspetos, tais como alocação de recursos e escalabilidade. Estas plataformas têm as seguintes caraterísticas chave: [5]

- **Consistência -** As plataformas de gestão de contentores oferecem consistência de duas maneiras: primeiro devido à independência de recursos, ou seja, os contentores podem ser alterados, iniciados ou eliminados sem afetar nenhum componente da aplicação, e segundo, permitem controlar versões.
- **Eficiência -** As plataformas de gestão de contentores permitem poupar tempo aquando o desenvolvimento e escalamento de aplicações. A capacidade de fazer alterações e adicionar funcionalidades sem perturbar todo o eco-sistema de uma aplicação promove o rápido desenvolvimento.
- **Segurança -** Uma aplicação isolada é tipicamente uma aplicação mais segura. Usando as plataformas de gestão de contentores é possível ligar os serviços com aplicações exteriores com um risco reduzido no que toca a segurança.

2.1.4 API *Gateway*

Enquanto que nos sistemas monólitos os clientes apenas comunicam com um serviço, o próprio monólito, numa arquitetura baseada em micro-serviços, um cliente pode comunicar com vários serviços. No entanto, a interação direta dos clientes com os serviços tem potenciais problemas: [8]

- O código do cliente fica mais complexo, pois é necessário saber a localização de todos os serviços.
- O cliente e os micro-serviços ficam acoplados, pois o cliente precisa de saber a decomposição de cada um dos serviços.
- Uma operação única pode requerer várias chamadas aos serviços, o que adiciona mais latência.
- Os serviços com *endpoints* públicos são um risco de segurança.

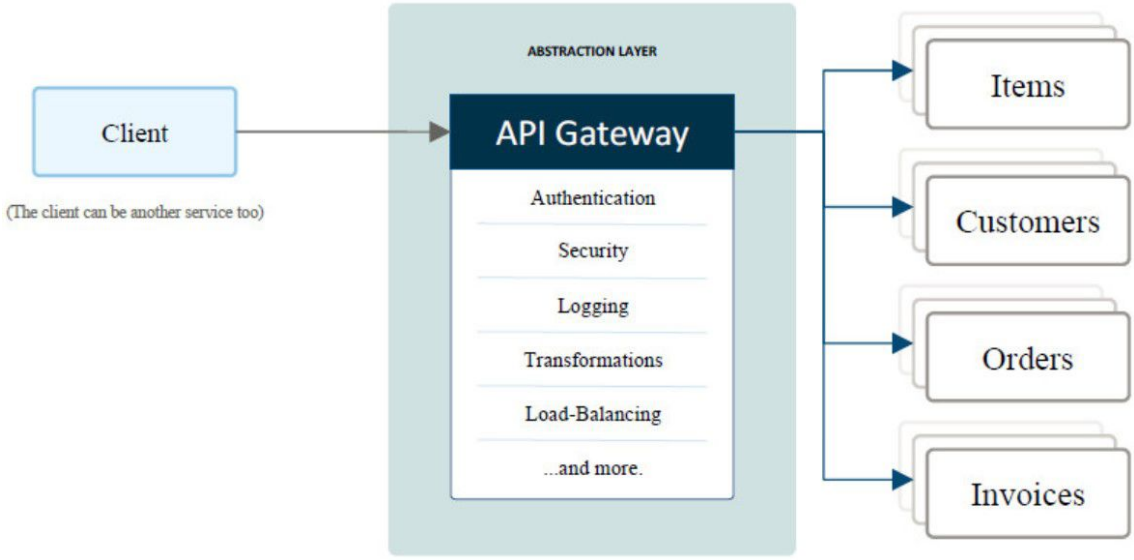

Figura 2.4: API *Gateway* [23]

A *gateway* é uma camada de abstração que se situa no meio dos clientes e dos serviços. As suas funcionalidades podem ser agrupados nos seguintes pontos: [8]

- **Redirecionamento -** A *gateway* redireciona os pedidos dos clientes para um ou mais serviços. Isto permite desacoplar os clientes dos micro-serviços, pois desta maneira os clientes só precisam de conhecer um *endpoint*, a *gateway*.
- **Agregração -** A *gateway* pode ser usada para agregar múltiplos pedidos num único. Isto acontece quando uma operação requerer múltiplas chamadas aos serviços. Desta maneira o cliente faz um pedido à *gateway*, que por sua vez faz vários pedidos a todos os serviços necessários, agrega os resultados, e envia tudo de volta para o cliente.
- **Consolidação -** O uso da *gateway* permite descarregar funcionalidades dos serviços para a *gateway*. É útil para consolidar funções num único lugar, em vez de serem implementadas por vários serviços, tal como autenticação ou autorização.

Visto que a *gateway* é um único ponto de entrada para os serviços, pode também realizar outras tarefas, tal como limitação de taxa dos clientes (*throttling*), *caching* de respostas, compressão de dados, monitorização ou balanceamento de carga.

Balanceamento de Carga

Um balanceador de carga é um componente que distribui tráfico de rede através de um aglomerado de servidores. Posiciona-se entre o cliente e os servidores, aceita tráfico de rede e aplicacional e distribui esse tráfico para os servidores usando vários algoritmos. Ao fazer balanceamente de carga é possível reduzir a carga de um servidor individual, e previne que um serviço se torne num único ponto de falha. Portanto, melhora a disponibilidade e responsividade de uma aplicação. [50]

Um balanceador de carga pode usar vários algoritmos, de modo a definir o comportamento a seguir para escolher o serviço adequado e redirecionar os pedidos dos clientes. Alguns algoritmos usados são apresentados na tabela 2.2:

Tabela 2.2: Algoritmos de balanceamento de carga [50]

Algoritmo	**Descrição**
Round Robin	Escolhe o serviço sequencialmente
Least Connections	Escolhe o service com menos conexões ativas
Least Response Time	Escolhe o serviço que tem menos conexões ativas e que tem a média de tempo de resposta mais baixa
Least Bandwidth	Escolhe o serviço que atualmente está a servir menos tráfico, medido em megabits por segundo (Mbps)
Least Packets	Escolhe o serviço que recebeu menos pacotes dentro de um determinado intervalo de tempo

2.2 Tecnlologias usadas na aplicação

Docker [9]

Docker é um projeto de código aberto que oferece soluções para desenvolver *software*, os contentores. Permite guardar o código de um serviço e as suas dependências num pequeno pacote chamado imagem. A imagem é posteriormente usada para gerar uma instância da aplicação, o contentor. Na figura 2.5 está representada a arquitetura do Docker, e vai ser analisada seguidamente. [10]

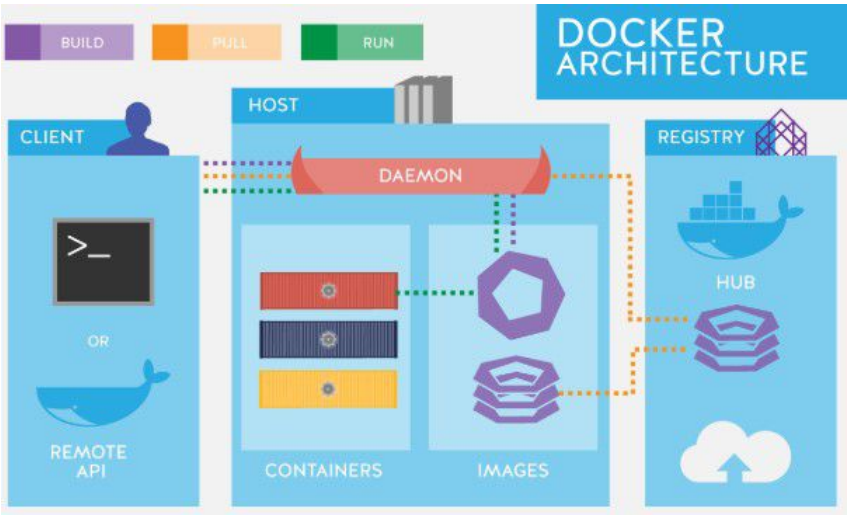

Figura 2.5: Arquitetura do Docker

O Docker usa o modelo cliente-servidor, e é constituido pelos seguintes componentes:

- **"Docker daemon" -** O *daemon* é responsável por todas ações relacionadas com os contentores, e recebe comandos através da Command-line Interface (CLI) ou uma Representational State Transfer (REST) API.
- **"Docker Client" -** O cliente é usado pelos utilizadores para comunicar com o Docker.
- **"Docker Objects" -** Os objetos são usados para montar a aplicação. São os seguintes:
  - **"Images" -** As imagens são um modelo somente de leitura. São usadas para armazenar e enviar aplicações.
  - **"Containers" -** Os contentores são ambientes encapsulados onde as aplicações são executadas. É definido por uma imagem e opções de configuração.
- **"Docker Registries" -** Os registos são localzações onde é possível guardar ou baixar imagens.

Docker Swarm [12]

O Docker Swarm é uma plataforma de gestão de contentores. A versão 1.12.0 e seguintes versões do Docker permitem ao programador implantar contentores no chamado modo "Swarm", ou seja, em vários nós [12]. A figura 2.6 representa a arquitetura do Docker Swarm, e consiste em "managers" e "workers". O utilizador pode declarar o estado dos vários serviços usando ficheiros de configuração YAML.

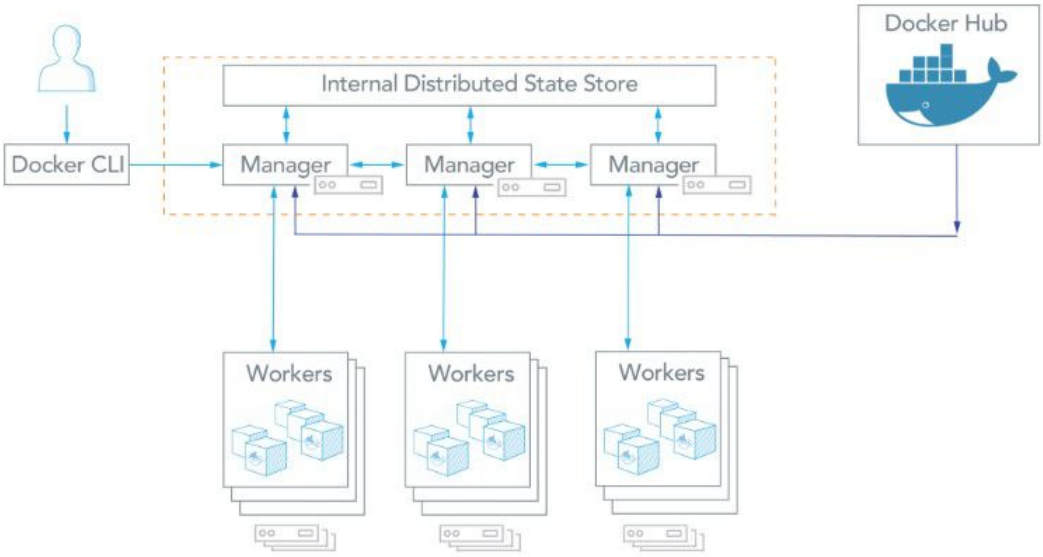

Figura 2.6: Arquitetura do Docker Swarm [25]

Seguidamente serão apresentados alguns termos relacionados com o Docker Swarm [25]:

- **"Node" -** São instâncias do Swarm. Podem ser distribuidos localmente ou em *clouds* públicas.
- **"Swarm" -** É um aglomerado de "nodes".
- **"Manager Nodes" -** Os "manager nodes" recebem definições dos serviços do utilizador e enviam trabalho para os "worker nodes". Os "manager nodes" também podem realizar o trabalho de "worker nodes".
- **"Worker Nodes" -** Recolhem e executam tarefas dos "manager nodes".
- **"Service" -** Especifica a imagem do contentor e o número de instâncias.
- **"Task" -** É uma unidade atómica de um "Service" escalonado num "worker node".

Zuul [35]

Zuul é um projeto de código aberto da Netflix, e serve de porta de entrada (*gateway*) de todos os pedidos dirigidos à sua aplicação de *streaming*. O Zuul usa diferentes tipos de filtros de modo a ser possível adicional funcionalidade à *gateway*. Este filtros podem ser usados para adicionar as seguintes funcionalidades: [46]

- Autenticatição e autorização

- Monitoria
- Encaminhamento dinâmico
- Testes de stress
- Respostas estáticas

O ciclo de vida dos pedidos que entram no zuul está representado na figura 2.7. Os tipos de filtros por defeito que podem ser usados estão explicados seguidamente:

- **"Pre Filters" -** Filtros executam antes do pedido ser encaminhado para o servidor.
- **"Routing Filters" -** Filtros manipulam o encaminhamento do pedido para o servidor.
- **"Post Filters" -** Filtros executam depois do pedido ter sido encaminhado para o servidor.
- **"Error Filters" -** Filtros executam quando um erro ocorre num dos filtros anteriores.

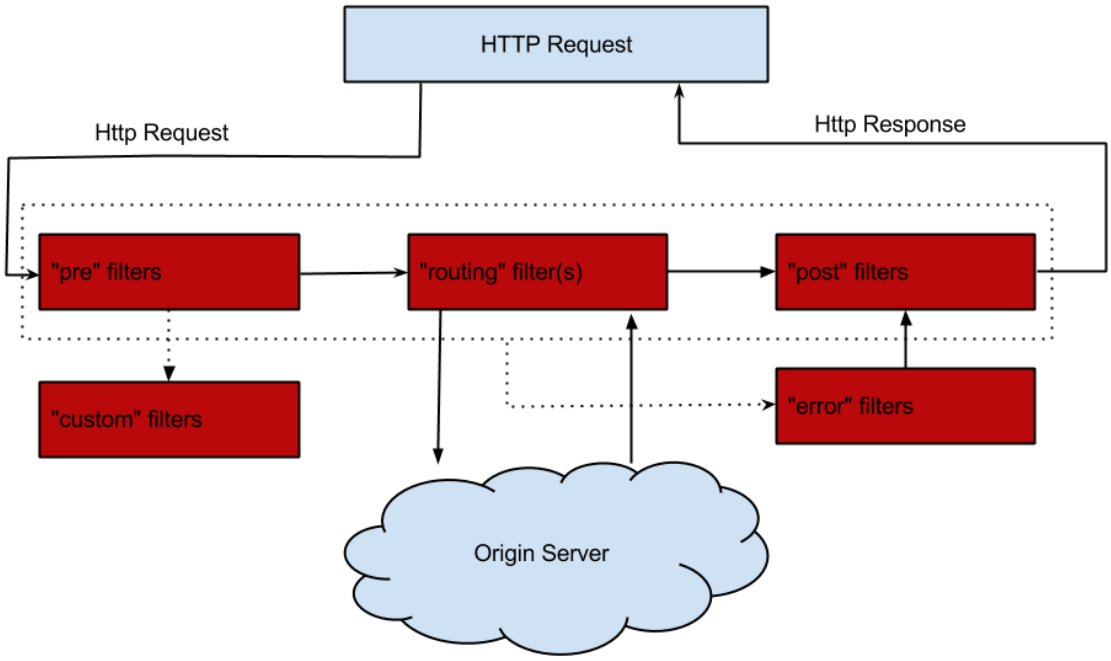

Figura 2.7: Ciclo de vida dos pedidos [20]

Ribbon [33]

Ribbon é um projeto de código aberto da Netflix, e fornece balanceadores de carga para comunicar com aglomerados de servidores. Este balanceadores de carga têm as seguintes componentes: [51]

- **"Rule" -** componente lógico que determina qual o servidor que é escolhido de uma lista.

- **"Ping" -** Componente que executa em segundo plano que verifica a saúde dos servidores.
- **"Server List" -** Lista dos servidores disponíveis. Esta lista pode ser estática ou dinâmica.

Eureka [32]

Eureka é um projeto de código aberto da Netflix e é muito utilizado na *cloud* da Amazon Web Services (AWS). É um serviço baseado em REST e é usado principalmente para descoberta de serviços. A figura 2.8 representa a implementação do Eureka num cenário exemplo, neste caso a AWS:

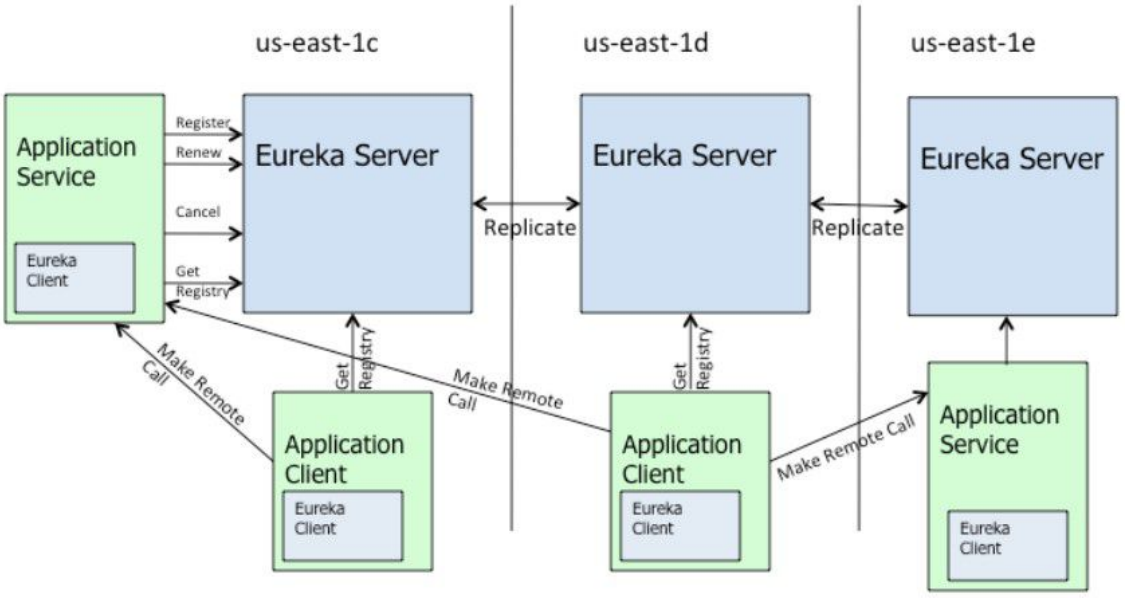

Figura 2.8: Arquitetura de alto nível do Eureka [15]

Geralmente a descoberta de serviços é replicada, neste caso um servidor Eureka está a ser replicado por cada zona, mas tal não é obrigatório. O que é obrigatório é os serviços terem de se registar no Eureka e tratar de toda a logística subsequente. Isto quer dizer que eles também têm de renovar o seu registo através do envio de *heartbeats*. Caso a renovação não seja feita, o serviço é eventualmente descartado. É de realçar que após um serviço se registar no Eureka, a sua informação é replicada por todos os nós do aglomerado. Os clientes do Eureka são normalmente os balanceadores de carga, que pretendem receber a lista de servidores disponíveis. [15]

Grafana

Grafana é um projeto de código aberto dedicado à construção de *dashboards*. Tem suporte total para *backends* como o Graphite [19], Prometheus [42], Elasticsearch [14], OpenTSDB [38] ou InfluxDB [21] e tem suporte parcial para MySQL [29] ou PostgreSQL [41].

A construção de *dashboard* é feita através de painéis. Cada painel é responsável por mostrar uma visualização de dados, como por exemplo, tabelas, gráficos de barras ou gráficos temporais. É também possível importar *plugins* desenvolvidos pela comunidade e que estão presentes no repositório oficial do Grafana. Estes aspetos fazem com que o Grafana seja altamente costumizável.

2.3 Soluções atuais de monitoria

Nagios

Nagios é uma aplicação grátis e de código aberto que monitoriza sistemas, redes e infraestruturas. Oferece alertas quando acontece um erro e alerta uma segunda vez quando o problema é resolvido. Oferece os seguintes serviços:

- monitoria da rede, infraestrura e aplicação.
- envio de alertas.
- apresenta os resultados na forma de gráficos.

Esta solução força a existência de agentes nos anfitriões para funcionar: Nagios Remote Plugin Executor (NRPE), Nagios Remote Data Processor (NRDP), Nagios cross Platform Agent (NCPA).

Zabbix

Zabbix é uma solução de monitoria de código aberto baseada no modelo servidor-cliente. Permite monitorizar a infraestrutura como um todo, desde pequenos processos aplicacionais até bases de dados. Para guardar a informação pode usar MySQL, PostgreSQL, SQLite, Oracle ou IBM DB2. Esta solução também é baseada em agentes. Oferece serviços como:

- monitoria da rede, infraestrura e aplicação.
- envio de alertas.
- apresenta os resultados na forma de gráficos.

New Relic [36]

New Relic é uma solução comercial especializada em monitorizar e analisar servidores e aplicações *web*. Esta solução oferece os seguintes serviços:

- monitoria da rede, infraestrura e aplicação.
- envio de alertas.
- apresenta os resultados na forma de gráficos.

Para esta solução funcionar é necessário instrumentar todos os serviços da aplicação.

Pinpoint [39]

Pinpoint é uma solução de código aberto para sistemas distribuídos de grande escala escrita em Java. Esta solução ajuda a analisar toda a estrutura de um sistema e a interligação de todos os seus componentes usando *tracing*. Oference os seguintes serviços:

- monitoria da rede, infraestrura e aplicação.
- apresenta os resultados na forma de gráficos.

Dynatrace [13]

Dynatrace é uma solução comercial que permite monitorizar todo o sistema, desde o *front-end* ao *back-end*, e da infraestrutura à *cloud* através do uso de agentes. Oferece os seguintes serviços:

- monitoria da rede, infraestrura e aplicação.
- envio de alertas.
- apresenta os resultados na forma de gráficos.

2.3.1 Comparação das soluções

A tabela 2.3 mostra a comparação entre as soluções de monitoria apresentadas anteriormente. É de realçar que todas usam agentes ou instrumentação do código.

Tabela 2.3: Comparação das soluções de monitoria

Solução / Caraterística	Nagios	Zabbix	New Relic	Pinpoint	Dynatrace
Solução é grátis	✓	✓		✓	
Monitoriza a infraestrutura	✓	✓	✓	✓	✓
Monitoriza a aplicação	✓	✓	✓	✓	✓
Monitoriza a rede	✓	✓	✓	✓	✓
Envia alertas	✓	✓	✓		✓
Apresenta os resultados na forma de gráficos	✓	✓	✓	✓	✓
Usa agentes ou instrumentação do código	✓	✓	✓	✓	✓

Capítulo 3

Metodologia Proposta

Neste documento nós atacamos o problema da monitoria de arquiteturas de micro-serviços. Em soluções verticais, a monitoria era mais simples, pois a aplicação não muda muito ao longo do tempo. Os sistemas de micro-serviços evoluiram a partir de novas metodologias de desenvolvimento, como as metodologias ágeis e novas técnicas de implementação, como os contentores. Os sistemas de monitoria são seguiram esta evolução e ainda são baseadas em técnicas de monitorização de sistemas monólitos. De facto, a monitoria é um problema mais complexo e difícil para sistemas altamente dinâmicos.

Nós analisamos o problema de monitoria através de uma perspetiva diferente. A abordagem típica consistia em instrumentar ou adicionar agentes no máximo de camadas possíveis, desde os anfitriões até à camada de aplicação, passando também pelo *middleware*. Infelizmente, esta abordagem involve mudar o código fonte da aplicação. Enquanto esta técnica cria múltiplos pontos de monitoria, também adiciona pontos de falha e manutenção adicionais, e consequentemente acoplam monitoria com lógica de negócio. Isto vai contra a metodologia de micro-serviços, que seguia a premissa de módulos pequenos orientados à função.

Para eliminar a necessidade de instrumentação, nós seguimos uma abordagem diferente. Sabendo que os micro-serviços recorrem a uma *gateway* para fazer descoberta de serviços e encaminhar pedidos, nós adicionamos a capacidade de coleta de métricas a esta *gateway*. A ideia passa por fazer com que a *gateway* recolha informação como tempos de resposta e endereços IP da origem e destino dos pedidos. Esta abordagem tem vantagens, como o desacoplamento do sistema de monitoria da aplicação de micro-serviços, sem prejudicar o potencial de escalabilidade do sistema, porque a *gateway* e serviços associados são escaláveis horizontavelmente.

Capítulo 4

Arquitetura do Sistema

Através da investigação realizada conseguimos chegar a uma arquitetura para o sistema de monitoria que preenche as nossas necessidades.

O passo seguinte consiste na definição e apresentação dos requisitos funcionais, que são funcionalidades que o nosso sistema deve suportar. O próximo passo prende-se com a definição de requisitos não-funcionais, também conhecidos como atributos de qualidade. Atributos de qualidade são um conjunto de atributos que o nosso sistema deve respeitar em determinadas situações. Por fim, vamos apresentar a nossa proposta para a arquitetura.

4.1 Requisitos Funcionais

De seguida vamos definir os requisitos funcionais do sistema de monitoria. Vamos também dar uma breve explicação sobre cada um deles, de modo a que fiquem bastante claros.

- **#RF01 - O código fonte do sistema de micro-serviços a monitorizar não deve ser alterado:** É muito importante desacoplar a solução de monitoria do sistema de microserviços, caso contrário vai ter de ser sujeita a uma elevada manutenção.
- **#RF02 - O sistema de monitoria deve ser capaz de retirar métricas como tempos de resposta, e endereços IP tanto da máquina invocada como da máquina invocadora:** Estas são as métricas mais importantes a retirar do sistema de micro-serviços. Os tempo de resposta vão ajudar-nos a inferir sobre a performance do sistema, enquanto que os endereços IP vão permitir associar duas máquinas a um determinado pedido. Esta última informação vai ser bastante útil para determinar a topologia do sistema de micro-serviços.
- **#RF03 - O sistema de monitoria deve disponibilizar os dados recolhidos:** A base de dados deve ser disponibilizada ao utilizador final, de modo a este construir visualizações de dados de acordo com as suas necessidades.
- **#RF04 - O sistema de monitoria deve apresentar a informação de modo visual com recurso a uma *dashboard*:** A informação deve ser apresentada em formato de gráficos e tabelas. A *dashboard* deve apresentar a seguinte informação:
  - Número de micro-serviços e respetivas instâncias registadas no sistema.

- Número de pedidos a entrar no sistema ao longo do tempo.
- Tempo de resposta dos pedidos ao longo do tempo.
- Distribuição dos pedidos.
- Topologia da aplicação de micro-serviços.
- Tempos médios de tempo de resposta por cada micro-serviço.

4.2 Atributos de Qualidade

Os atributos de qualidade vão ser apresentados usando diversos cenários, estes divididos pelas seguintes porções [4]:

- **Fonte:** entidade que gera o estímulo.
- **Estímulo:** condição que requer uma resposta aquando da sua chegada ao sistema.
- **Artefacto:** elementos do sistema que são estimulados.
- **Ambiente:** condições do sistema aquando da chegada do estímulo.
- **Resposta:** atividade realizada após a chegada do estímulo.
- **Medida:** quando a resposta ocorre, esta deve ser mensurável de modo a que o requisito possa ser testado.

4.2.1 Disponibilidade

Disponibilidade está relacionada com o tempo no qual o sistema está operacional. Se qualquer módulo não estiver operacional pode fazer com que se perca valiosa informação para avaliar o sistema de micro-serviços. No pior caso, poderia fazer com que os micro-serviços estivessem indisponíveis para o utilizador.

Tabela 4.1: Cenário de disponibilidade

Fonte	Código interno do sistema de monitoria ou interação humana
Estímulo	Algum componente do sistema de monitoria vai abaixo/fica indisponível ou é fisicamente comprometido
Artefacto	Processador ou memória da máquina em caso de o sistema de monitoria ir abaixo, ou um componente do sistema de monitoria em caso de algo ficar fisicamente comprometido
Ambiente	Operações habituais ou sistema de monitoria sobrecarregado
Resposta	O sistema de monitoria continua a funcionar normalmente e nenhuma interrupção é detetada
Medida	O sistema deve ter uma disponibilidade de 99.9%

4.2.2 Escalabilidade

Escalabilidade é bastante importante para o nosso sistema. Na eventualidade de o sistema receber um elevado número de pedidos numa curta janela de tempo, este não pode ser um ponto de estrangulamento. Caso contrário, iria comprometer o normal funcionamento dos micro-serviços.

Tabela 4.2: Cenário de escalabilidade

Fonte	Um ou múltiplos utilizadores
Estímulo	Um ou múltiplos pedidos para o sistema de micro-serviços
Artefacto	Sistema de monitoria
Ambiente	Operações habituais ou sistema de monitoria sobrecarregado
Resposta	Cada pedido é manipulado com sucesso
Medida	A análise de cada pedido não pode demorar em média mais do que 1 segundo quando comparado com uma *gateway* sem monitoria

4.2.3 Desempenho

O desempenho está relacionado com as acções de monitoria que são realizadas e que podem causar com a experiência global do utilizar os micro-serviços seja mais lenta. Sabendo à partida que este facto é inevitável, o nosso objetivo é que o sistema de monitoria não adicione muita sobrecarga.

Tabela 4.3: Cenário de desempenho

Fonte	Utilizadores
Estímulo	Utilizador faz um pedido ao sistema de micro-serviços
Artefacto	Sistema de monitoria
Ambiente	Em tempo de execução
Resposta	Cada pedido é analisado com sucesso
Medida	A análise de cada pedido não pode demorar em média mais do que 1 segundo quando comparado com uma *gateway* sem monitoria

4.2.4 Compatibilidade

Compatibilidade é uma atributo chave para o nosso sistema. Partindo do princípio que o sistema de micro-serviços segue os passos necessários para ser integrado na nossa solução, ver capítulo 5, o nosso sistema de monitoria deve funcionar com normalidade.

Tabela 4.4: Cenário de compatibilidade

Fonte	Sistema de monitoria
Estímulo	Adição do sistema de micro-serviços a monitorar
Artefacto	Sistema de monitoria
Ambiente	Em construção
Resposta	O sistema de monitoria deve funcionar com normalidade
Medida	Se o sistema de monitoria funciona ou não

4.3 Arquitetura do Sistema

De seguida vamos apresentar a nossa arquitetura para o sistema, tendo em conta os requisitos funcionais apresentados na secção 4.1 e os atributos de qualidade apresentados na secção 4.2.

Decidimos apresentar a nossa arquitetura de forma hierárquica, com diferentes níveis de abstração segundo o modelo C4 de Simon Brown [6]. O primeiro nível de abstração diz respeito ao diagrama de contexto do sistema, e representa a interação entre sistemas e utilizadores. Este nível é bastante útil pois permite dar um passo atrás e ver a questão de uma perspetiva geral. O segundo nível de abstração denomina-se de diagrama de contentores e descreve a arquitetura de alto nível do sistema. Mostra também as tecnologias mais relevantes que foram escolhidas para cada contentor, bem como a forma de comunicação entre eles. No terceiro nível de abstração, temos os diagramas de componentes, que não é mais do que uma amplificação de cada contentor mencionado anteriormente. Cada contentor é decomposto em vários componentes lógicos de acordo com a sua funcionalidade.

É de realçar que o autor propõe um último nível de abstração, diagrama de classes, que nós decidimos descartar.

4.3.1 Diagrama de Contexto do Sistema

Na figura 4.1 está representado o diagrama de contexto do sistema. É possível ver as interações do sistema de monitoria com todas as entidades com as quais comunica externamente.

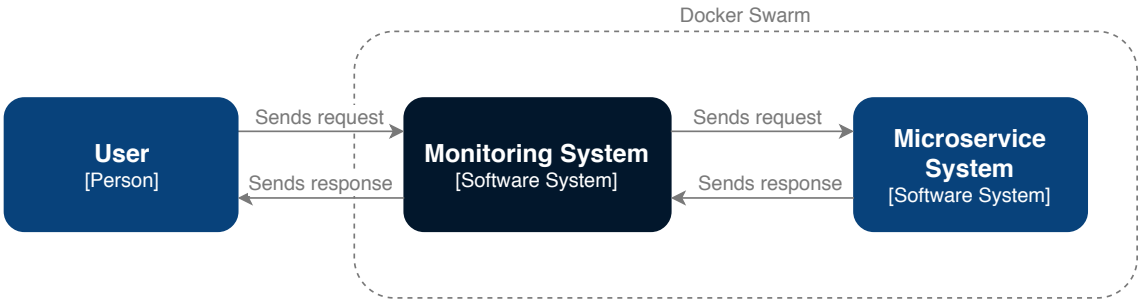

Figura 4.1: Diagrama de contexto do sistema

O sistema de monitoria é como um espécie de intermediário entre o utilizador e o sistema de micro-serviços. Visto de fora o que faz é redirecionar pedidos de utilizadores e as suas devidas respostas dos micro-serviços.

É de realçar que tanto o sistema de monitoria e os micro-serviços estão a correr dentro de uma rede do **Docker Swarm**, uma plataforma de gestão de contentores. Esta decisão permite-nos automatizar muitas tarefas, as quais vão ser analisadas com pormenor seguidamente, o que faz com que a nossa solução seja mais fácil de operar para o utilizador final.

4.3.2 Diagrama de Contentores

Uma vez que em sistemas distribuídos, particularmente nos baseados em micro-serviços, é comum resolver o problema de descoberta de serviços com uma *gateway*, torna-se barato e bastante prático observar o sistema a esse nível. Para tal utilizámos dois componentes da **Netflix**: **Zuul** e **Eureka**, responsáveis por *gateway* e descoberta de serviços, respetivamente. É com base nestes serviços que conseguimos colecionar medições fora do caminho crítico. A figura 4.2 mostra todos os contentores, bem como a tecnologia utilizada em cada um deles.

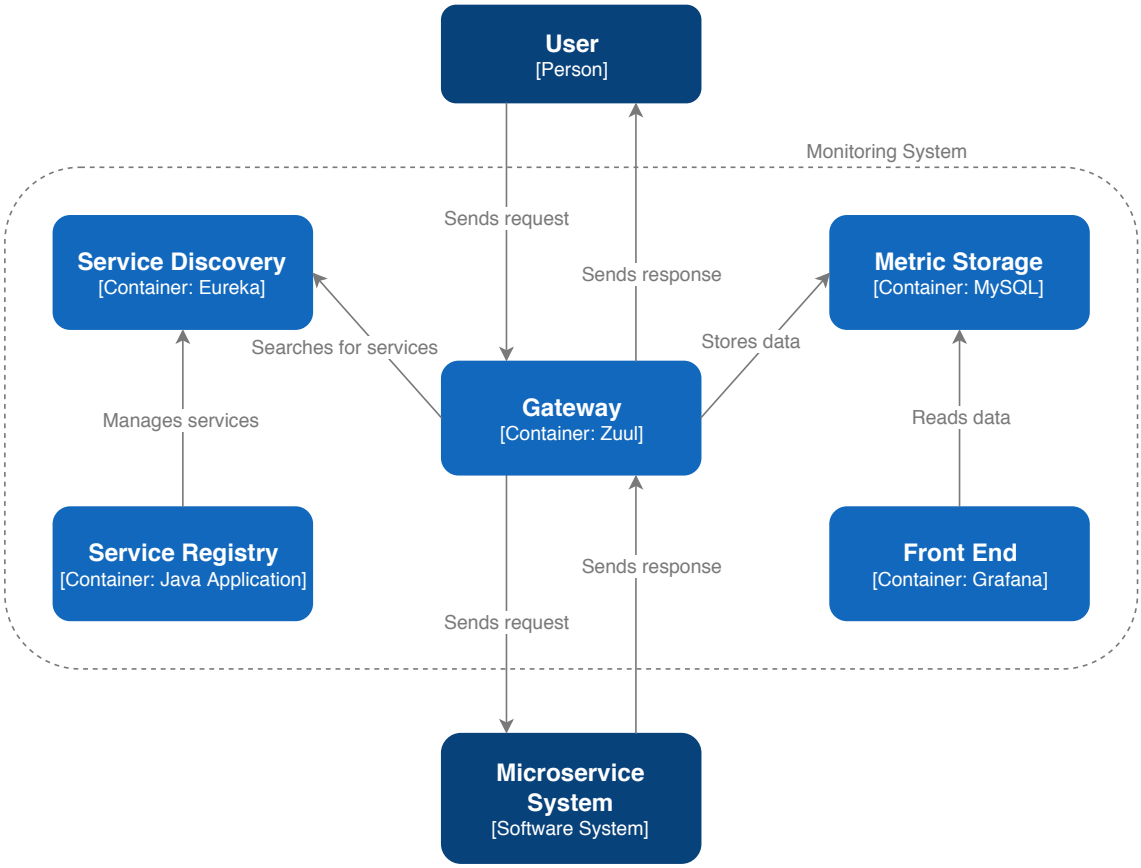

Figura 4.2: Diagrama de contentores

Utilizamos um contentor responsável por fazer o registo de serviços, "Service Registry". Isto permite automatizar por completo o processo de registo de micro-serviços, bem como remover a necessidade de instrumentação. O registo é feito através de meta-dados disponibilizados na descrição dos contentores, e pelo gestor de contentores. Isto obriga a que todos os micro-serviços a monitorar sejam colocados dentro de contentores.

Para cada pedido, seja proveniente de clientes ou entre micro-serviços são guardadas métricas temporais em relação a dimensões de origem e destino, as quais serão apresentadas no capítulo 6. Para guardar esta informação usamos uma base de dados relacional baseada em **MySQL**. A nossa visão passa por dar acesso ao utilizador a esta base de dados, para que seja possível criar *dashboards* adaptadas às suas necessidades. No entanto, a nossa solução de monitoria apresenta já uma *dashboard* embutida, indo buscar informação à base de dados supra mencionada. Vamos discutir a informação apresentada no nosso *Front End* também no capítulo 6.

4.3.3 Diagramas de Componentes

O registo de serviços da nossa solução é completamente automático, o que nos dá uma enorme vantagem, visto que não precisamos de instrumentar os micro-serviços. A sua arquitetura está representada com detalhe na figura 4.3.

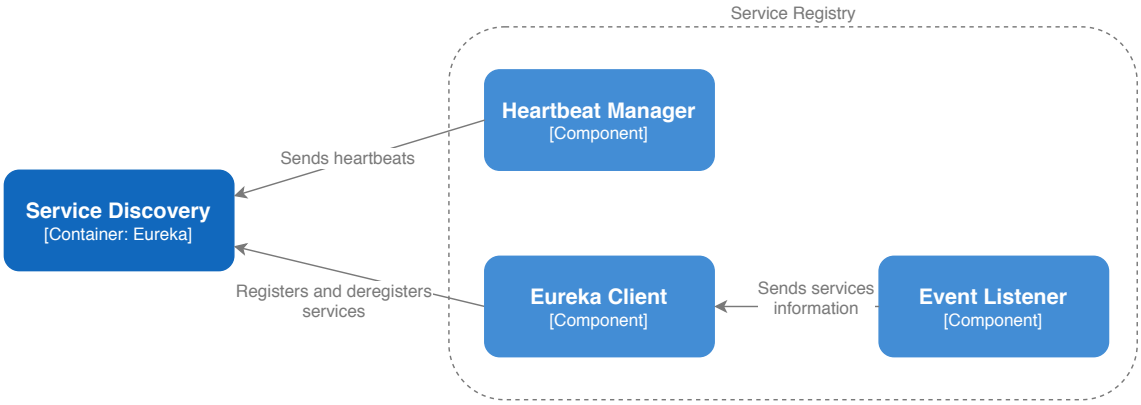

Figura 4.3: Diagrama de componentes do registo de serviços

O nosso componente "Event Listener" é notificado quando há alguma alteração dos contentores presentes na rede do **Docker Swarm**. Conseguimos saber quando um contentor é criado ou eliminado, o que nos permite registar ou desregistar os micro-serviços na descoberta de serviços.

O **Eureka** é por design um serviço passivo, ou seja, têm de ser os clientes a fazer todas as operações. A nossa solução vai de encontro a esta arquitetura. Temos dois componentes subscritos diretamente à REST API do **Eureka**, o "Eureka Client" e o "Heartbeat Manager". O primeiro é responsável por registar e desresgistar os serviços presentes na rede, enquanto que o segundo trata de enviar *heartbeats* de todos os serviços atualmente registados. Caso a nossa solução não enviasse *heartbeats* os serviços registados no **Eureka** iriam eventualmente ser descartados.

Por fim, na figura 4.4 está representado com mais detalhe o coração da nossa solução, a *gateway*. Esta serve como ponto centralizado do nosso sistema, pelo qual entram todos os pedidos aos micro-serviços. Os pedidos são balanceados através do **Ribbon**, a solução da Netflix para resolver o problema do balanceamento de carga, e que se integra perfeitamente com as restantes tecnologias da Netflix usadas no nosso sistema. Por defeito é usado o algoritmo *Round Robin*, mas este é possível de ser alterado de forma manual nas configurações da *gateway*.

Antes do pedido ser balanceado, o pedido passa primeiro por um filtro. Neste ponto é recolhido o tempo de chegada ao nosso sistema e é guardado no cabeçalho do pedido. Quando chega a resposta dos micro-serviços, esta passa ainda por outro filtro, no qual é retirada toda a informação do pedido. Os dados são recolhidos e tratados de forma assíncrona através de uma fila. Garantimos que esta fila bloqueie para leitura, mas nunca para escrita, de modo a não sobrecarregar o sistema. Finalmente, os dados são guardados na base de dados mencionada anteriormente.

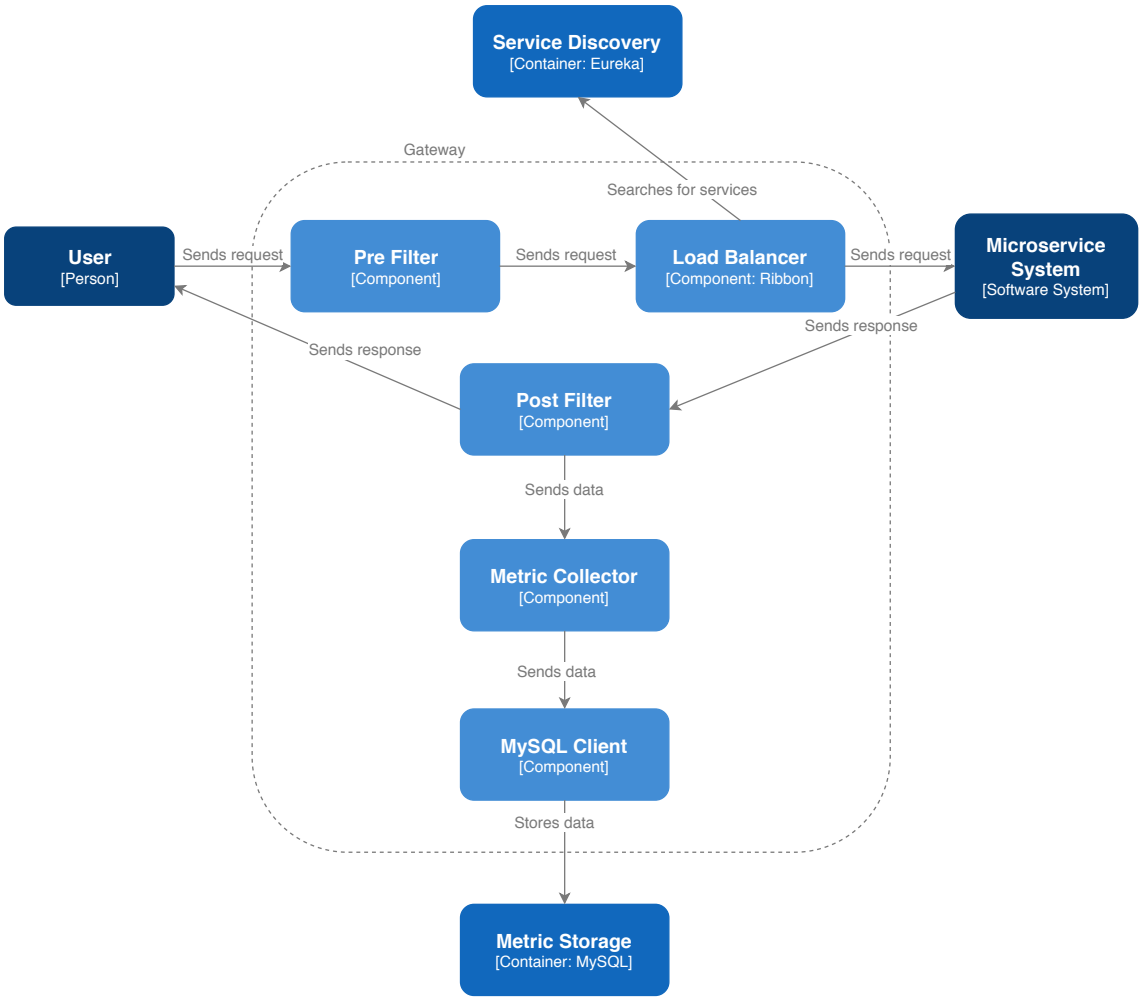

Figura 4.4: Diagrama de componentes da *gateway*

É relevante mencionar que todos os componentes envolvidos na monitoria são horizontavelmente escaláveis, por isso não é prejudicado quer o desempenho, quer a disponibilidade da aplicação. Como removemos toda a instrumentação dos micro-serviços, todos os processos responsáveis por extração e tratamento de dados são realizados fora do caminho crítico, e por conseguinte não sobrecarregam o sistema.

Capítulo 5

Implementação

Nesta secção vamos abordar a componente prática do nosso trabalho, especificando tudo o que foi implementado incluindo algumas alterações realizadas na arquitetura.

Foi feita uma implementação não intrusiva utilizando a plataforma de gestão de contentores **Docker Swarm**. O código fonte e as instruções de utilização da ferramenta estão disponíveis no **GitHub** [16] como código aberto. A ferramenta é bastante fácil de instalar num sistema que tenha o **Python** e o **Docker** instalados. A ferramenta de monitoria necessita de uma rede de *overlay* [11], portanto o sistema deve também ter esta rede devidamente configurada para garantir o correto funcionamente da nossa abordagem.

Seguidamente, o utilizador deve configurar a solução de acordo com as suas necessidades utilizando o nosso ficheiro de configurações. O único parâmetro que precisa de ser obrigatoriamente configurado é a rede de *overlay*. O resto dos parâmetros não necessitam de ser preenchidos, sendo utilizados os valores por defeito.

Para usar a nossa solução de monitoria é necessário baixar o repositório do **GitHub**, definir a rede de *overlay* no nosso ficheiro de configurações e executar o *script* de instalação. Este *script*, que utiliza a linguagem **Python**, gera e corre automaticamente o ficheiro *docker-compose* necessário para o **Docker Swarm**.

Depois destes passos o sistema de monitoria está montado, falta portanto iniciar a aplicação de micro-serviços a monitorizar. Para tal é necessário que cada micro-serviço esteja dentro de um contentor. Depois basta criar um ficheiro de configurações *docker-compose* para colocar a aplicação a correr no **Docker Swarm**. É de realçar que a aplicação deve estar na mesma rede do sistema de monitoria. Para informação adicional é favor visitar o nosso repositório do **GitHub**.

Na secção 4 referimos que todos os componentes do sistema de monitoria são horizontavelmente escaláveis. No entanto, na nossa implementação atual apenas o registo de serviços está replicado. Replicar todos os componentes obrigava a um grande esforço, e consequentemente tempo que não tinhamos à nossa disposição. O registo de serviços foi a exceção à regra, pois era necessária a sua replicação para o funcionamento do sistema. Dado que a implementação do **Docker Swarm** pode ter várias máquinas, é necessário um componente de registo de serviços em cada uma delas, para conseguirmos apanhar todas as alterações no estado dos contentores.

O módulo responsável por guardar as métricas, "metric storage", também sofreu alterações. Tinhamos previsto usar uma base de dados **MySQL**, mas devido a algumas limitações presentes no **Grafana** tivemos de adicionar outra base de dados. O **Grafana**

suporta **MySQL** mas não totalmente, o que resultou em problemas na construção de alguns gráficos. Optámos então por adicionar um nova base de dados, o **InfluxDB**. Esta é uma base de dados baseada em séries temporais, e está otimizada para a manipulação de dados indexados por tempo ou intervalos de tempo. Os dados guardados nas duas bases de dados são idênticos e podem ser consultados na tabela 5.1.

Tabela 5.1: Métricas coletadas

Métrica	**Tipo**
Tempo de início	Long
Tempo de fim	Long
Duração	Long
IP de origem	String
Porta de origem	String
Serviço de destino	String
Instância de destino	String
IP de destino	String
Porta de destino	String
Método	String
URL	String
Código de resposta	String

Para cada pedido, independentemente da sua origem, outro micro-serviço ou um cliente, guardamos métricas relacionadas com a origem e o destino do pedido.

Além de gráficos com médias e quartis, como *box-plots*, esta informação bruta permite-nos criar informação de alto nível sobre o sistema. Por exemplo, é possível extrair a topologia e caracterizar o nível de interação entre os serviços dinamicamente. No módulo de *Front End* usámos o **Grafana**, uma solução altamente costumizável.

Em suma, na tabela 5.2 estão presentes todos os contentores físicos usados pela nossa ferramenta. É de realçar que uma vez que seja devidamente instalada, qualquer aplicação colocada dentro da mesma rede *overlay* do **Docker Swarm** vai usar automaticamente a nossa *gateway*, descoberta de serviços e devida monitorização.

Tabela 5.2: Contentores usados pelo ferramenta

Contentor	**Descrição**
Eureka	Descoberta de serviços
Zuul	*Gateway*
Registo de Serviços	Gere ciclo de vida dos contentores introduzidos pelo utilizador
InfluxDB	Base de dados baseada em séries temporais
MariaDB	Base de dados relacional
Grafana	*Dashboard*

Capítulo 6

Validação da Arquitetura

Validação tem um papel preponderante no desenvolvimento e implementação de uma arquitetura, e pode ser visto como o último passo para a conclusão de um projeto.

6.1 Configuração Experimental

Nesta secção vamos explicar a configuração que está por trás dos testes efetuados para validar a nossa arquitetura.

Para efetuar a monitoria foi utilizada a ferramenta descrita na secção 5 e que está disponível como código aberto no **GitHub**. A outra componente experimental é a aplicação que nos permite testar o nosso método de monitoria. A aplicação implementada está relacionada com música e tem cinco micro-serviços com funções bem definidas. De forma geral a aplicação permite aos seus clientes gerir utilizadores, listas de músicas e as próprias músicas. A sua arquitetura de alto nível é apresentada na figura 6.1.

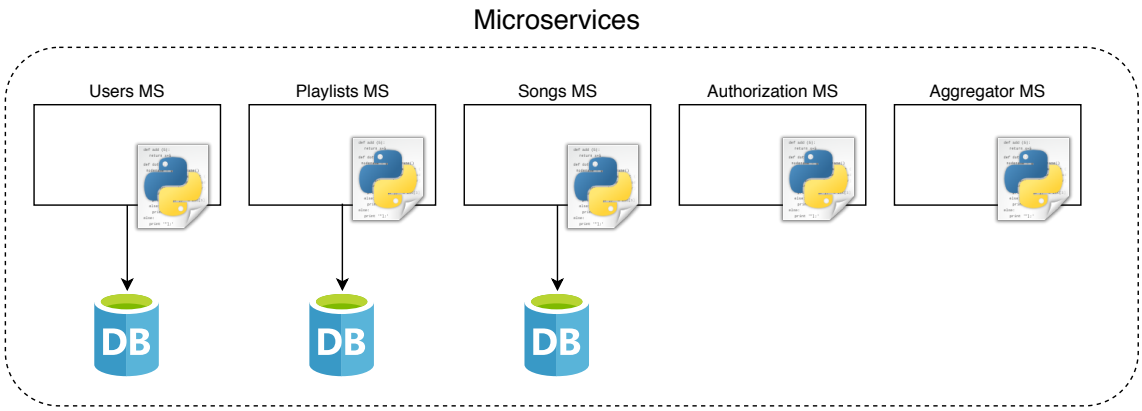

Figura 6.1: Aplicação de micro-serviços usada como teste

Na tabela 6.1 estão identificados os pontos de acesso a cada micro-serviço, bem como os métodos usados na invocação e uma breve descrição. Esta aplicação também está no **GitHub** [17] como código aberto. O repositório contém ainda instruções para utilizar a aplicação sem estar ligada ao sistema de monitoria.

Tabela 6.1: Micro-serviços e respetivas funcionalidades disponibilizadas

Microserviço	**Funcionalidade**	**Método**	**Descrição**
Authentication MS	/	GET	Validar se MS está operacional
	/login	POST	Criação de *token* caso as credenciais estejam corretas
Users MS	/	GET	Validar se MS está operacional
	/login	POST	Validar credenciais de login
	/users	GET	Consultar um utilizador
	/users	POST	Criar um utilizador
	/users/{id}	DELETE	Eliminar um utilizador
	/users/{id}	PUT	Atualizar um utilizador
Playlists MS	/	GET	Validar se MS está operacional
	/playlists	GET	Retornar listas associadas a um utilizador
	/playlists	POST	Criar uma lista
	/playlists/{id}	GET	Retornar uma lista
	/playlists/{id}	DELETE	Eliminar uma lista
	/playlists/{id}	PUT	Atualizar uma lista
	/playlists/songs/{id}	DELETE	Remover uma música da lista
	/playlists/songs/{id}	GET	Consultar todas as músicas da lista
	/playlists/songs/{id}	POST	Adicionar uma música à lista
Songs MS	/	GET	Validar se MS está operacional
	/songs	GET	Retornar música
	/songs	POST	Criar uma música
	/songs/convert/{id}	GET	Converter uma música de extensão **mp3** para **wav**
	/songs/criteria	GET	Retornar uma lista de músicas com base numa expressão
	/songs/{id}	DELETE	Apagar uma música
	/songs/{id}	PUT	Atualizar uma música
Aggregator MS	/	GET	Validar se MS está operacional
	/playlists/songs/{id}	GET	Retornar toda a informação de todas as músicas de uma lista

É importante referir que cada micro-serviço da aplicação das músicas foi inicializado com duas réplicas (instâncias). O sistema completo de teste é apresentado na figura 6.2.

Todo o *software* foi instalado numa implementação do **Docker Swarm** com duas máquinas virtuais. Cada máquina tinha como sistema operativo o **Ubuntu 16.04**, 2 cores virtuais e 4 GB de Random Access Memory (RAM).

Para simular carga no sistema foi usado o **Apache JMeter** [2]. As configuração utilizadas são especificadas na secção 6.2. Cada cliente efetuou o seguinte fluxo de operações em ciclo: 1) Criar utilizador; 2) Autenticação; 3) Obter utilizador; 4) Atualizar utilizador;

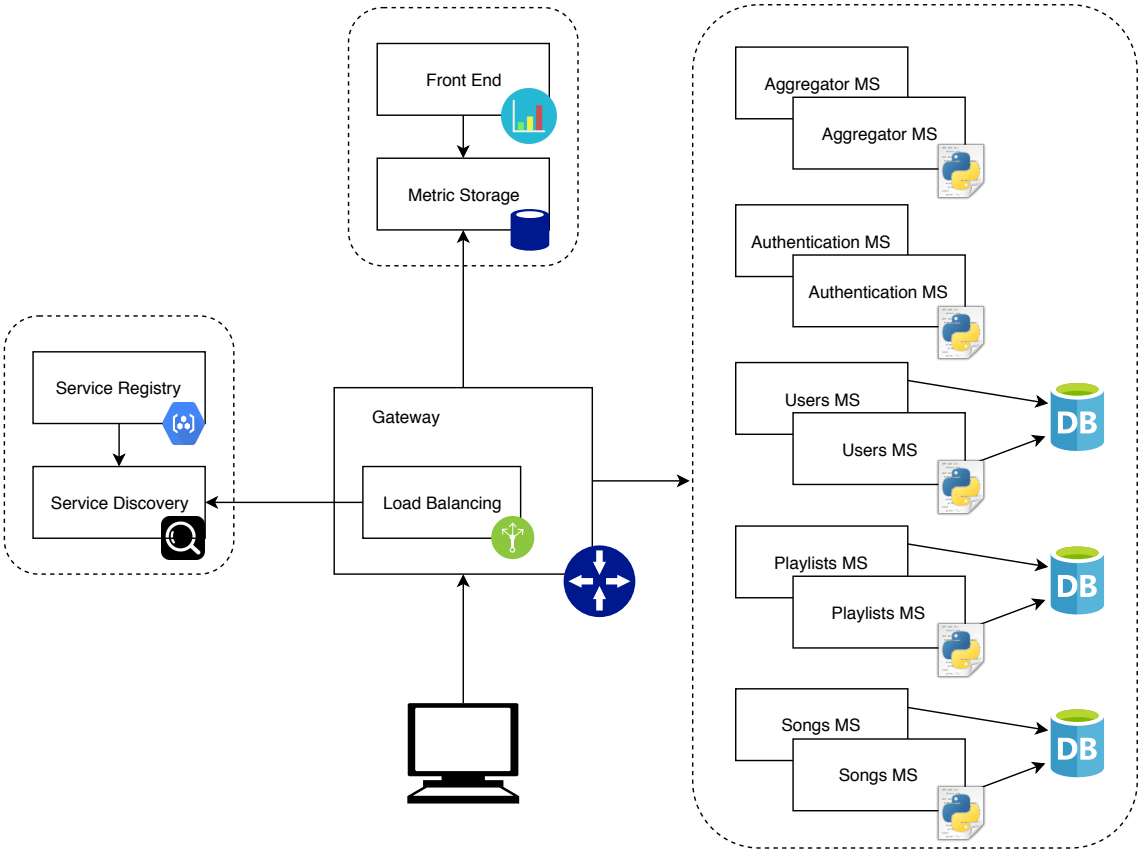

Figura 6.2: Arquitetura do sistema de teste

5) Adicionar música; 6) Consultar música; 7) Atualizar música; 8) Converter música; 9) Adicionar lista de músicas; 10) Obter lista de músicas; 11) Atualizar lista de músicas; 12) Adicionar lista à música; 13) Obter música da lista; 14) Obter todas as músicas de uma lista; 15) Remover música da lista; 16) Remover lista de músicas; 17) Remover música; 18) Remover utilizador.

O nosso objetivo com esta experiência é bastante claro: numa primeira fase é validar a nossa arquitura, e numa segunda fase é compreender os limites, vantagens e desvantagens da nossa abordagem de monitoria de "caixa-preta".

6.2 Testes

Nesta secção vamos analisar se conseguimos cumprir os requisitos funcionais e atributos de qualidade. De referir que os testes realizados baseam-se em colocar carga no sistema, como foi descrito na secção 6.1. Foram usados os seguintes parâmetros de configuração:

- **Clientes (*threads*):** número máximo de clientes a interagir com o sistema ao mesmo tempo.
- **Período de lançamento:** intervalo de tempo necessário para introduzir um novo cliente no sistema.
- **Tempo de teste:** tempo que o teste demorou a executar.

O único parâmetro que variou durante os testes foi o número de clientes. O período de lançamento e o tempo de teste foi sempre o mesmo, de 120 segundos e 10 minutos respetivamente.

RF01: O código fonte do sistema de micro-serviços a monitorizar não deve ser alterado

Para validar este requisito funcional não foi necessário introduzir carga no sistema. Todos os micro-serviços da aplicação das músicas estavam em contentores, e a partir do momento que foram adicionados à rede do **Docker Swarm** foram apanhados pelo nosso registo de serviços. Toda a monitorização foi feita a partir da *gateway*. Consequentemente, não foi feita qualquer alteração ao código fonte dos micro-serviços.

RF02: O sistema de monitoria deve ser capaz de retirar métricas como tempos de resposta, e endereços IP tanto da máquina invocada como da máquina invocadora

Para validar este requisito foi introduzida carga no sistema com 10 clientes. Na figura 6.3 está representado um excerto da base de dados **MariaDB**. Através da análise da figura é possível verificar que estamos a recolher o tempo de resposta de cada pedido, bem como o endereço IP das duas máquinas intervenientes.

start_timestamp	end_timestamp	duration	source_ip	source_port	destiny_microservice	destiny_instance	destiny_port	destiny_ip	method	url	status_code
1534706779929	1534706781311	1382	10.255.0.2	63884	music-users	acec498edf94	5000	10.0.0.149	POST	/users	201
1534706781787	1534706781854	67	10.0.0.159	58034	music-users	6d4381140fc9	5000	10.0.0.150	POST	/login	200
1534706781367	1534706781870	503	10.255.0.2	63884	music-auth	42808fcf1322	5003	10.0.0.159	POST	/login	200
1534706781931	1534706781969	38	10.255.0.2	57055	music-users	acec498edf94	5000	10.0.0.149	GET	/users	200
1534706782002	1534706782034	32	10.255.0.2	57055	music-users	6d4381140fc9	5000	10.0.0.150	PUT	/users/1	200
1534706782072	1534706782503	431	10.255.0.2	57055	music-songs	f75d7c7440ae	5001	10.0.0.153	POST	/songs	201
1534706782534	1534706782589	55	10.255.0.2	57055	music-songs	b201d9d1611b	5001	10.0.0.152	GET	/songs/criteria	200
1534706782623	1534706782653	30	10.255.0.2	57055	music-songs	f75d7c7440ae	5001	10.0.0.153	PUT	/songs/1	200
1534706782681	1534706783650	969	10.255.0.2	57055	music-songs	b201d9d1611b	5001	10.0.0.152	GET	/songs/convert/1	200
1534706783681	1534706783918	237	10.255.0.2	57055	music-playlists	e5de1f38a112	5002	10.0.0.156	POST	/playlists	201
1534706783970	1534706784008	38	10.255.0.2	58305	music-playlists	e24c2de93be4	5002	10.0.0.155	GET	/playlists	200
1534706784038	1534706784068	30	10.255.0.2	58305	music-playlists	e5de1f38a112	5002	10.0.0.156	PUT	/playlists/1	200
1534706784115	1534706784130	15	10.0.0.155	60650	music-songs	f75d7c7440ae	5001	10.0.0.153	GET	/songs	200
1534706784095	1534706784141	46	10.255.0.2	58305	music-playlists	e24c2de93be4	5002	10.0.0.155	POST	/playlists/songs/1	200
1534706784170	1534706784196	26	10.255.0.2	58305	music-playlists	e5de1f38a112	5002	10.0.0.156	GET	/playlists/songs/1	200
1534706784434	1534706784451	17	10.0.0.161	41724	music-playlists	e24c2de93be4	5002	10.0.0.155	GET	/playlists/songs/1	200
1534706784532	1534706784557	25	10.0.0.161	41726	music-songs	b201d9d1611b	5001	10.0.0.152	GET	/songs	200
1534706784224	1534706784628	404	10.255.0.2	58305	music-aggr	e2afcf702d3d	5004	10.0.0.161	GET	/playlists/songs/1	200
1534706784655	1534706784685	30	10.255.0.2	58305	music-playlists	e5de1f38a112	5002	10.0.0.156	DELETE	/playlists/songs/1	200
1534706784713	1534706784736	23	10.255.0.2	58305	music-playlists	e24c2de93be4	5002	10.0.0.155	DELETE	/playlists/1	200
1534706784762	1534706784781	19	10.255.0.2	58305	music-songs	f75d7c7440ae	5001	10.0.0.153	DELETE	/songs/1	200
1534706784808	1534706784844	36	10.255.0.2	58305	music-users	acec498edf94	5000	10.0.0.149	DELETE	/users/1	200
1534706784885	1534706784906	21	10.255.0.2	58305	music-users	6d4381140fc9	5000	10.0.0.150	POST	/users	201
1534706784978	1534706784995	17	10.0.0.158	50264	music-users	acec498edf94	5000	10.0.0.149	POST	/login	200

Figura 6.3: Métricas colecionadas

RF03: O sistema de monitoria deve disponibilizar os dados recolhidos

Para validar este requisito funcional não foi necessário introduzir carga no sistema. Por defeito no nosso ficheiro de configurações as bases de dados não são disponibilizadas ao

utilizador. No entanto basta alterar um parâmetro no ficheiro para isso deixar de acontecer. Para mais detalhes é favor consultar o nosso repositório no **GitHub**.

RF04: O sistema de monitoria deve apresentar a informação de modo visual com recurso a uma *dashboard*

Para validar este requisito foi introduzida carga no sistema com 10 clientes. A *dashboard* do sistema de monitoria vai ser apresentada na figura 6.4 e os seus componentes vão ser discutidos seguidamente.

Figura 6.4: *Dashboard* baseada em Grafana

- **[1]** Este botão permite gerir a janela temporal de todos os gráficos simultaneamente. Tanto conseguimos ter informação sobre os últimos cinco minutos, ou sobre os últimos dois anos. Também é possível escolher uma opção para atualizar os dados da *dashboard* de x em x tempo.

- **[2]** Nesta secção temos os *dropdowns* que são usados para mostrar informação mais específica. Podemos ver informação sobre qualquer micro-serviço, ou as suas respetivas instâncias. Apesar de não ser tão importante como os anteriores, também é possível analisar informação consoante os métodos usados nos pedidos ou o código da sua resposta.

- **[3]** Este painel dá informação sobre o número de micro-serviços registados no sistema dentro do intervalo de tempo selecionado.

- **[4]** Este painel dá informação sobre o número total de instâncias dos micro-serviços registadas no sistema dentro do intervalo de tempo selecionado.

- **[5]** Este painel correlaciona os micro-serviços com o nome das instâncias. Apresenta ainda informação adicional sobre endereços ip e portos.

- **[6]** Este painel dá informação sobre o número de pedidos a chegar ao sistema em cada unidade de tempo.

- **[7]** Este painel da informação sobre a distribuição dos tempos de resposta dos pedidos por cada unidade de tempo. Zonas com cores mais escuras representam uma maior concentração de pedidos.

- **[8]** Este painel dá informação sobre a distribuição dos pedidos de forma geral dentro do intervalo de tempo selecionado.

- **[9]** Este painel dá informação sobre o tempo médio dos tempos de resposta dentro do intervalo de tempo selecionado.

Em suma, foram cumpridos quase todos os requisitos excepto apresentar a topologia do sistema. Para apresentar a topologia iamos recorrer a um *plugin* de diagramas para o **Grafana** [40], mas até à altura de escrita do documento este apresentava um defeito que não havia sido corrigido.

Disponibilidade

Não foi possível validar este atributo de qualidade. No entanto foram utilizadas algumas estratégias de modo ao sistema não prejudicar a disponibilidade da solução a aplicação existente (micro-serviços a funcionar com uma *gateway standard*):

- **Deteção de falhas -** O **Docker Swarm** garante-nos deteção de falhas em cada um dos contentores da nossa solução. Esta deteção é transparente à nossa aplicação.
- **Recuperação de falhas -** Está relacionado com a deteção de falhas, pois quando o **Docker Swarm** deteta que algum contentor foi abaixo, este está configurado a reiniciá-lo imediatamente.
- **Prevenção de falhas -** Nós prevenimos falhas através da redundância, evitando pontos únicos de falha. É realizado através da replicação de todos os nossos contentores. Contudo, como foi discutido na secção 5, na nossa implementação apenas um contentor foi replicado.

Desempenho e Escalabilidade

Estes atributos de qualidade estão relacionados, logo vão ser validados da mesma maneira e usando os mesmos testes. Para introduzir carga no sistema foi utilizado um número variável de clientes, 10, 50 e 200. Para cada um destes testes foram utilizados 3 cenários:

- **Cenário 1 -** Os pedidos foram feitos diretamente aos micro-serviços, sem ser necessário passarem pela *gateway*.
- **Cenário 2 -** Os pedidos foram feitos através de uma *gateway* que não faz monitoria. Foi implementado o **Zuul** sem nenhum tipo de modificações.
- **Cenário 3 -** Os pedidos foram feitos através da nossa solução de monitoria.

Os resultados são apresentados com detalhe nas tabelas 6.2, 6.3 e 6.4. O gráfico da figura 6.5 compara a média dos tempos de resposta de todos os testes.

É de realçar que não foi possível acabar o teste do do cenário 3 com uma carga de 200 clientes. Isto deveu-se essencialmente a falta de memória na máquina. A *gateway* quando retira as métricas brutas dos pedidos, coloca esta informação numa fila. A fila ficou tão grande que a 5 minutos do tempo de teste a máquina ficou sem recursos.

Tabela 6.2: Comparação de desempenho com máximo de 10 clientes

	Cenário 1		**Cenário 2**		**Cenário 3**	
Micro-serviço	**Média (ms)**	**Desvio Padrão (ms)**	**Média (ms)**	**Desvio Padrão (ms)**	**Média (ms)**	**Desvio Padrão (ms)**
Users MS	22.978	9.785	29.409	29.322	33.389	34.440
Authentication MS	37.959	12.470	58.125	104.077	62.129	95.338
Aggregator MS	151.764	19.343	215.521	347.321	233.867	355.785
Songs MS	222.076	432.495	220.848	410.938	227.871	411.145
Playlists MS	28.093	15.700	39.030	56.755	44.258	45.914
	Média total do cenário 1: 92.574 ms ± 97.958 ms					
	Média total do cenário 2: 112.586 ms ± 189.682 ms					
	Média total do cenário 3: 120.302 ms ± 188.524 ms					

Tabela 6.3: Comparação de desempenho com máximo de 50 clientes

	Cenário 1		**Cenário 2**		**Cenário 3**	
Micro-serviço	**Média (ms)**	**Desvio Padrão (ms)**	**Média (ms)**	**Desvio Padrão (ms)**	**Média (ms)**	**Desvio Padrão (ms)**
Users MS	159.711	334.778	52.735	197.257	50.532	52.199
Authentication MS	83.818	183.540	91.930	278.899	88.416	110.767
Aggregator MS	258.032	247.204	938.864	1245.162	1027.840	1075.723
Songs MS	412.271	566.555	257.774	432.136	258.489	425.658
Playlists MS	305.038	428.866	622.565	833.142	675.569	725.895
	Média total do cenário 1: 243.774 ms ± 352.188 ms					
	Média total do cenário 2: 392.773 ms ± 597.319 ms					
	Média total do cenário 3: 420.169 ms ± 467.983 ms					

Devido ao facto de não ter sido possível completar todos os testes com sucesso, não é possível validar nenhum dos atributos de qualidade. O nosso objetivo era encontrar o ponto em que os micro-serviços ou o sistema de monitoria se tornasse num ponto de estrangulamento. No caso de o sistema de monitoria se tornasse num ponto de estrangulamento não passaria nos testes, no caso de ser ao contrário, os micro-serviços serem um ponto de estrangulamento e a diferença entre os tempos fosse menos de 1 segundo, então conseguiríamos validar os atributos de qualidade.

No entanto vale a pena referir que o sistema de monitoria estava a passar nos testes, pois a diferença entre uma solução que usa uma *gateway standard* e a nossa solução, que usa uma *gateway* costumizada com coleta de métricas de pedidos, era em média menos do que 1 segundo.

Conseguimos ir mais além, comparando a nossa solução de monitoria com uma aplicação que recebe pedidos diretamente, a diferença também era em média menos do que o 1 segundo. Era interessante verificar até que ponto o uso do conceito da *gateway* influenciaria a média dos tempos de resposta.

Tabela 6.4: Comparação de desempenho com máximo de 200 clientes

	Cenário 1		Cenário 2		Cenário 3 *	
Micro-serviço	**Média (ms)**	**Desvio Padrão (ms)**	**Média (ms)**	**Desvio Padrão (ms)**	**Média (ms)**	**Desvio Padrão (ms)**
Users MS	219.549	381.401	845.589	457.714	636.086	2020.582
Authentication MS	120.890	218.294	961.543	520.727	905.658	2595.527
Aggregator MS	1513.007	1402.810	709.410	459.022	1369.995	4386.846
Songs MS	466.992	588.884	846.031	457.522	657.723	2622.813
Playlists MS	1647.042	1557.439	745.286	489.498	1118.819	3314.028
	Média total do cenário 1: 793.496 ms ± 829.765 ms					
	Média total do cenário 2: 821.571 ms ± 476.896 ms					
	Média total do cenário 3: 937.656 ms ± 2987.959 ms					

** Não foi possível concluir o teste, pois a máquina foi abaixo*

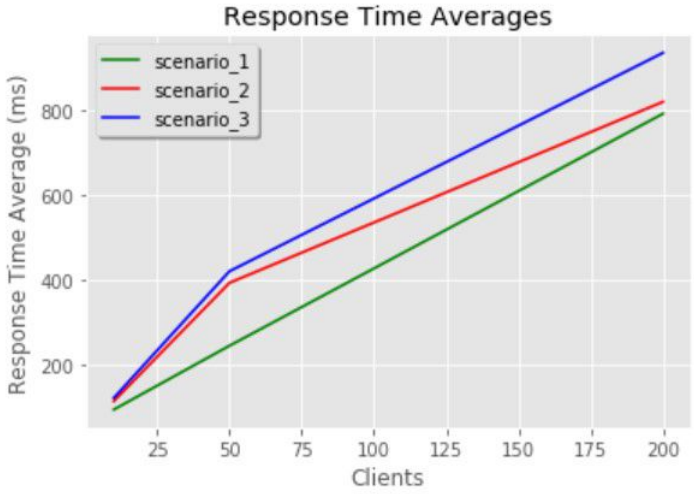

Figura 6.5: Médias dos tempos de resposta consoante o número de clientes

Compatibilidade

Para validar este atributo de qualidade não foi necessário introduzir carga no sistema. Visto que usamos o **Docker Swarm** e um módulo que automaticamente regista todos os micro-serviços presentes na rede na nossa descoberta de serviços, é seguro dizer que qualquer aplicação de micro-serviços, desde que esteja dentro de contentores, é compatível com a nossa solução de monitoria.

6.3 Discussão de Resultados

Nesta secção vamos apresentar os resultados da utilização da nossa técnica de monitoria, baseada apenas na extração de dados a partir da *gateway*. Esta técnica permitiu-nos extrair dados brutos da interação entre micro-serviços, e por conseguinte, criar um conjunto de gráficos com informação relevante para os administradores sem utilização de instrumentação. Nas figuras 6.6, 6.7, 6.8, 6.9 e 6.10 estão presentes várias visualizações da *dashboard* consoante o micro-serviço selecionado. Estas visualizões foram retiradas da *dashboard* descrita na secção 6.2.

Como foi mencionado na secção 6.2 não foi possível adicionar visualizações da topologia dos serviços. Para efeitos de discussão a figura 6.11 contém o grafo da aplicação de micro-serviços que foi feito manualmente.

Figura 6.6: *Dashboard* com o serviço "Aggregator MS" selecionado

Figura 6.7: *Dashboard* com o serviço "Authentication MS" selecionado

Figura 6.8: *Dashboard* com o serviço "Playlists MS" selecionado

Figura 6.9: *Dashboard* com o serviço "Songs MS" selecionado

Figura 6.10: *Dashboard* com o serviço "Users MS" selecionado

Em primeiro lugar, é importante conhecer os tempos de resposta de cada um dos micro-serviços. Mais detalhadamente, além de saber quais os micro-serviços que são mais lentos, é também necessário perceber se um serviço é chamado muitas ou poucas vezes e qual a distribuição dos seus tempos de resposta.

A ideia geral é que todas estas visualizações permitem dar uma ideia sobre a capacidade

do sistema, tempos de resposta por micro-serviço e a sua importância relativa. Assim, temos uma boa visão do sistema como um todo.

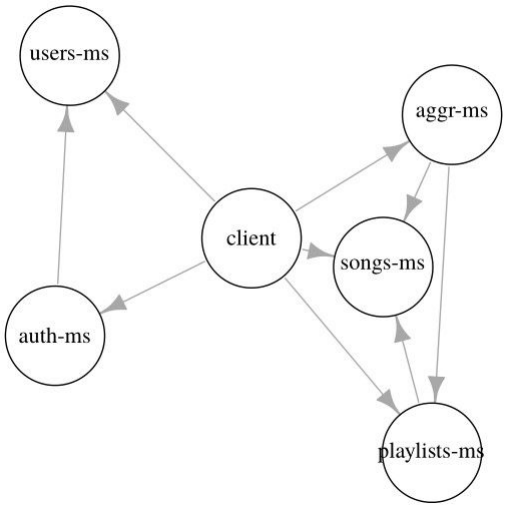

Figura 6.11: Grafo da aplicação de micro-serviços

Olhando para os dados da aplicação exemplo, segundo o gráfico que mostra os tempos médios de resposta por micro-serviço, gráfico esse representado dentro do retângulo vermelho das figuras 6.6 a 6.10, verificamos que os serviços "Aggregator MS" e "Songs MS" são os que apresentam maior latências, de 58ms e 55ms respetivamente. Os restantes micro-serviços têm latências que variam de 10ms a 16ms. Olhando agora para o grafo da figura 6.11 vemos que o serviço "Songs MS" é invocado por vários serviços incluindo o cliente, mas nunca invoca nenhum serviço. Logo toda a latência deste serviço é causada por ele mesmo. Por outro lado, ao analisar o serviço "Aggregator MS" vemos que este é apenas invocado pelo cliente, e por sua vez invoca vários serviços, incluido o "Songs MS". Podemos concluir então que as elevadas latências no serviço "Aggregator MS" muito provavelmente são causadas pelo serviço "Songs MS". Ora, isto significa que o serviço "Songs MS" é o maior candidato a receber uma intervenção por parte dos operadores.

Podemos ir mais além e tentar descobrir se o problema advém de alguma das suas instâncias. Na figura 6.12 conseguimos correlacionar o serviço com a sua respetiva instância. Depois disto podemos olhar para as visualizações de cada uma destas instâncias, as quais representadas nas figuras 6.13 e 6.14.

Services Information

Ip address	Instance	Port	Microservice
10.0.0.149	acec498edf94	5000	music-users
10.0.0.150	6d4381140fc9	5000	music-users
10.0.0.152	b201d9d1611b	5001	music-songs
10.0.0.153	f75d7c7440ae	5001	music-songs
10.0.0.155	e24c2de93be4	5002	music-playlists
10.0.0.156	e5de1f38a112	5002	music-playlists
10.0.0.158	83d9492027de	5003	music-auth
10.0.0.159	42808fcf1322	5003	music-auth
10.0.0.161	e2afcf702d3d	5004	music-aggr
10.0.0.162	e94a118bcce0	5004	music-aggr

Figura 6.12: Informação sobre micro-serviços

Ao analisar as duas figuras ao pormenor, é possível verificar que ambas são muito parecidas, com latências iguais de valor 55ms, o que faz entender que de facto o problema não é de nenhuma instância em particular, mas do serviço em geral.

Figura 6.13: *Dashboard* com a instância "b201d9d1611b" selecionada

Figura 6.14: *Dashboard* com a instância "f75d7c7440ae" selecionada

Comparando com as metodologias usadas em sistema de micro-serviços com instrumentação e agentes, esta tem algumas vantagens e desvantagens. Uma das desvantagens em relação à instrumentação é que não temos a granularidade para perceber por onde passam os pedidos, não havendo por isso informação exata da causalidade dos pedidos entre micro-serviços. Relativamente às vantagens, além de este método ser menos instrusivo, não tem a sobrecarga dos agentes ou código da instrumentação e como consequência fica o código desacoplado da insfraestrutura de monitoria. No caso de um sistema já em produção, o custo de implantação fica restrito à configuração da plataforma, não requerendo alterações ao sistema.

Capítulo 7

Conclusão e Trabalho Futuro

A correta monitoria e operação eficiente levantam grandes desafios aos administradores de sistemas. Com o novo paradigma dos micro-serviços esta tarefa tornou-se ainda mais complexa, devido à elasticidade e dinânica do sistema. A maioria da monitoria não foi desenhada para este tipo de sistemas, algo que tem levado as maiores empresas tecnológicas a criar soluções *in-house*, o que revela a falta de *standards* na área.

Neste documento abordámos o problema por outro prisma, sem recurso a instrumentação ou agentes. Tentámos analisar os limites de uma monitoria "caixa-negra", usando o facto de ser necessário um serviço de descoberta e encaminhamento entre micro-serviços. As evidências que recolhemos mostram que esta abordagem fornece informação relevante sobre o sistema com um esforço mínimo na integração no sistema. Adicionalmente, não estamos associados a nenhum tipo de sistema operativo ou linguagem, sendo até possível incorporar esta abordagem em sistema legados.

Para trabalho futuro, existem várias direções a seguir. Em primeiro lugar, replicar todas as componentes do nosso sistema, de modo a ser altamente disponível e escalável. Segundo, melhorar a nossa *dashboard*, para termos mais e melhor informação para ajudar os administradores a encontrar anomalias. Em último lugar, era bastante útil gerar modelos capazes de prever a ocupação dos micro-serviços.

Referências

[1] *An introduction to container technology*. 2017. URL: https://blog.kumina.nl/2017/03/an-introduction-to-container-technology/ (visited on 2018-08-23).

[2] *Apache JMeter*. URL: http://jmeter.apache.org/ (visited on 2018-08-18).

[3] Peter Arijs. *Comparing Microservices and Monolithic Applications From the Perspective of Monitoring*. 2016. URL: https://dzone.com/articles/comparing-microservices-and-monolithic-application (visited on 2018-08-29).

[4] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. third edition. Addison-Wesley Professional, 2012.

[5] *Best Container Management Software*. URL: https://www.g2crowd.com/categories/container-management (visited on 2018-08-24).

[6] Simon Brown. *Software architecture and the C4 model*. 2014. URL: http://www.codingthearchitecture.com/2014/08/24/c4_model_poster.html (visited on 2018-08-12).

[7] *CONTÊINERES NO GOOGLE*. URL: https://cloud.google.com/containers/ (visited on 2018-08-23).

[8] *Designing microservices: API gateways*. 2017. URL: https://docs.microsoft.com/en-us/azure/architecture/microservices/gateway (visited on 2018-08-25).

[9] *Docker*. URL: https://www.docker.com/ (visited on 2018-08-24).

[10] *Docker Alternatives*. URL: https://www.aquasec.com/wiki/display/containers/Docker+Alternatives+-+Rkt%5C%2C+LXD%5C%2C+OpenVZ%5C%2C+Linux+VServer%5C%2C+Windows+Containers (visited on 2018-08-24).

[11] *Docker overlay network*. URL: https://docs.docker.com/network/overlay/ (visited on 2018-08-16).

[12] *Docker Swarm*. URL: https://docs.docker.com/engine/swarm/ (visited on 2018-08-24).

[13] *Dynatrace*. URL: https://www.dynatrace.com/ (visited on 2018-08-28).

[14] *Elasticsearch*. URL: https://www.elastic.co/ (visited on 2018-08-28).

[15] *Eureka at a glance*. URL: https://github.com/Netflix/eureka/wiki/Eureka-at-a-glance (visited on 2018-08-28).

[16] *Github - monitoring_ms*. URL: https://github.com/fabiopina/monitoring_ms (visited on 2018-08-16).

[17] *Github - music_ms*. URL: https://github.com/fabiopina/music_ms (visited on 2018-08-18).

[18] *Grafana*. URL: https://grafana.com/ (visited on 2018-08-28).

[19] *Graphite*. URL: https://graphiteapp.org/ (visited on 2018-08-28).

[20] *How it Works*. URL: https://github.com/Netflix/zuul/wiki/How-it-Works (visited on 2018-08-28).

[21] *InfluxDB*. URL: https://www.influxdata.com/ (visited on 2018-08-28).

[22] *Introduction to Containers: Concept, Pros and Cons, Orchestration, Docker, and Other Alternatives*. 2016. URL: https://medium.com/flow-ci/introduction-to-containers-concept-pros-and-cons-orchestration-docker-and-other-alternatives-9a2f1b61132c (visited on 2018-08-23).

[23] Joab Jackson. *The Role of API Gateways in Microservice Architectures*. 2017. URL: https://thenewstack.io/api-gateways-age-microservices/ (visited on 2018-08-25).

[24] *Kibana*. URL: https://www.elastic.co/products/kibana (visited on 2018-08-28).

[25] *Kubernetes vs Docker Swarm*. 2017. URL: https://platform9.com/blog/kubernetes-docker-swarm-compared/ (visited on 2018-08-24).

[26] James Lewis and Martin Fowler. *Microservices*. 2014. URL: https://martinfowler.com/articles/microservices.html#footnote-etymology (visited on 2018-08-22).

[27] *Microservices on AWS*. URL: https://docs.aws.amazon.com/aws-technical-content/latest/microservices-on-aws/introduction.html (visited on 2018-08-22).

[28] *Monitoring Software*. URL: https://www.techopedia.com/definition/4313/monitoring-software (visited on 2018-08-25).

[29] *MySQL*. URL: https://www.mysql.com/ (visited on 2018-08-28).

[30] *Nagios*. URL: https://www.nagios.org/ (visited on 2018-08-28).

[31] *Netflix Atlas*. URL: https://github.com/Netflix/atlas (visited on 2018-08-28).

[32] *Netflix Eureka*. URL: https://github.com/Netflix/eureka (visited on 2018-08-28).

[33] *Netflix Ribbon*. URL: https://github.com/Netflix/ribbon (visited on 2018-08-28).

[34] *Netflix Vector*. URL: https://github.com/Netflix/vector (visited on 2018-08-28).

[35] *Netflix Zuul*. URL: https://github.com/Netflix/zuul (visited on 2018-08-28).

[36] *New Relic*. URL: https://newrelic.com/ (visited on 2018-08-28).

[37] *Opentracing*. URL: http://opentracing.io/ (visited on 2018-08-28).

[38] *OpenTSDB*. URL: http://opentsdb.net/ (visited on 2018-08-28).

[39] *Pinpoint*. URL: https://github.com/naver/pinpoint (visited on 2018-08-28).

[40] *Plugin diagrama do grafana*. URL: https://grafana.com/plugins/jdbranham-diagram-panel (visited on 2018-08-17).

[41] *PostgreSQL*. URL: https://www.postgresql.org/ (visited on 2018-08-28).

[42] *Prometheus*. URL: https://prometheus.io/ (visited on 2018-08-28).

[43] Chris Richardson. *Introduction to Microservices*. 2015. URL: https://www.nginx.com/blog/introduction-to-microservices/ (visited on 2018-08-22).

[44] Margaret Rouse. *distributed tracing*. URL: https://searchitoperations.techtarget.com/definition/distributed-tracing (visited on 2018-08-29).

[45] Paul Rubens. *What are containers and why do you need them?* 2017. URL: https://www.cio.com/article/2924995/software/what-are-containers-and-why-do-you-need-them.html (visited on 2018-08-23).

[46] Rafael Salerno. *Get to Know Netflix's Zuul.* 2016. URL: https://dzone.com/articles/spring-cloud-netflix-zuul-edge-serverapi-gatewayga (visited on 2018-08-28).

[47] Benjamin H. Sigelman et al. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure.* Technical report. Google, Inc, 2010.

[48] Dave Swersky. *The Hows, Whys and Whats of Monitoring Microservices.* 2018. URL: https://thenewstack.io/the-hows-whys-and-whats-of-monitoring-microservices/ (visited on 2018-08-26).

[49] *What is a Container.* URL: https://www.docker.com/resources/what-container (visited on 2018-08-23).

[50] *What is load balancing?* URL: https://www.citrix.com.br/glossary/load-balancing.html (visited on 2018-08-25).

[51] *Working with load balancers.* URL: https://github.com/Netflix/ribbon/wiki/Working-with-load-balancers (visited on 2018-08-28).

[52] *Zabbix.* URL: https://www.zabbix.com/ (visited on 2018-08-28).

[53] *Zipkin.* URL: https://zipkin.io/ (visited on 2018-08-28).