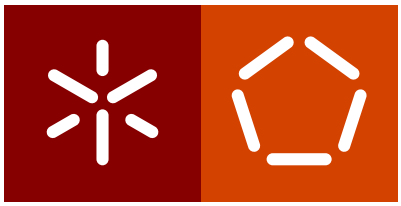


Universidade do Minho
Escola de Engenharia
Departamento de Informática

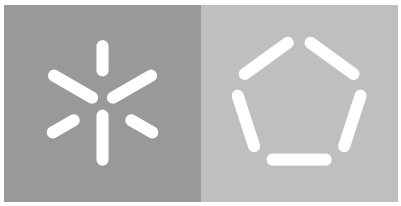
Daniel António Landeira da Silva

Monitorização e controlo de aplicações na nuvem

Master dissertation
Master's in Informatics Engineering

Dissertation supervised by
Prof. José Orlando Pereira

January 2021



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Daniel António Landeira da Silva

Monitorização e controlo de aplicações na nuvem

Master dissertation
Master's in Informatics Engineering

Dissertation supervised by
Prof. José Orlando Pereira

January 2021

DECLARAÇÃO DE INTEGRIDADE

Declaro ter atuado com integridade na elaboração do presente trabalho académico e confirmo que não recorri à prática de plágio nem a qualquer forma de utilização indevida ou falsificação de informações ou resultados em nenhuma das etapas conducente à sua elaboração.

Mais declaro que conheço e que respeitei o Código de Conduta Ética da Universidade do Minho.

AGRADECIMENTOS

Gostaria de começar por agradecer ao meu orientador de dissertação, o Prof. José Orlando Pereira, por todo o apoio demonstrado durante este processo. Esteve sempre pronto a tirar dúvidas, a passar o seu conhecimento, a discutir os vários problemas levantados e, acima de tudo, por acreditar ser possível atingir todos os objetivos a que nos propusemos, independentemente das adversidades impostas pelo ano 2020.

Queria agradecer aos meus colegas de curso, em particular ao Diogo Fernandes, ao Tiago Lameira e ao Pedro Silva, pela disponibilidade para troca de ideias e pela enorme motivação dada ao longo do processo.

Em especial, queria mostrar a minha gratidão à minha família pela o apoio e motivação constante, não só durante a elaboração da dissertação, mas ao longo de todo o meu percurso académico. A eles, estarei eternamente grato.

Não esquecendo aqueles que contribuíram e continuam a contribuir para campo em que esta dissertação se insere, gostaria de agradecer a todas as pessoas e entidades que trabalharam nos vários artigos e ferramentas mencionados ao longo da dissertação, em particular ao Francisco Neves pela criação do SYSQUERY. Sem o seu trabalho, não seria possível a elaboração e escrita desta dissertação.

ABSTRACT

Cloud orchestration systems, as Kubernetes, allow us to dynamically manage aspects such as location of components. This makes traditional resource-oriented monitoring systems inadequate. They also make it desirable that control mechanisms act directly on the orchestrator and not on individual components.

This project aims to design, develop and test an application for monitoring and control distributed database systems, solving the challenges posed by this new environment. This dissertation motivated by project H2020 CloudDB Appliance.

Keywords: Cloud, Monitoring, Control

RESUMO

Os sistemas de orquestração na nuvem, como o Kubernetes, permitem gerir dinamicamente aspetos como a (co-)localização de componentes e o número de instâncias de cada um. Isto faz com que os sistemas tradicionais de monitorização, orientados aos recursos físicos, sejam desadequados. Fazem também com que seja desejável que os mecanismos de controlo ajam diretamente sobre o orquestrador e não sobre os componentes individualmente.

Este projeto tem como objetivo projetar, desenvolver e testar uma aplicação para monitorização e controlo de sistemas de bases de dados distribuídas resolvendo os desafios colocados por este novo ambiente. Esta proposta de dissertação enquadra-se no projeto H2020 CloudDB Appliance.

Palavras-chave: Nuvem, Monitorização, Controlo

CONTEÚDO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE EXCERTOS DE CÓDIGO

LIST OF ABBREVIATIONS

API Application Programming Interface

KLM Keystroke Level Model

PID Process ID

PPID Parent Process ID

OS Operative System

IT Information technology

CI Continuous integration

CD Continuous delivery

IaaS Infrastructure-as-a-service

Paas Platform-as-a-service

1

INTRODUÇÃO

1.1 CONTEXTO

Um contentor é uma unidade de software que empacota o código e as respectivas dependências, para que a aplicação contida neste seja executada mais rapidamente e de forma mais confiável e segura[11]. O Kubernetes é um sistema de orquestração de contentores, que facilita a implantação, o dimensionamento e a gestão de aplicações em contentores, permitindo que os mecanismos de controlo sobre os contentores sejam executados sobre o orquestrador e não sobre os componentes individualmente, acrescentando assim uma camada de abstração [9].

Esta tecnologia é fundamental para as arquiteturas de microsserviços na implantação de aplicações. O uso de microsserviços é cada vez mais frequente e, segundo a Google, a maior parte das pesquisas sobre o tema microsserviços são impulsionados pela tecnologia, ou seja, sobre a tecnologia e como utilizá-la de maneira prática. Significa isto que perguntas como "O que são microsserviços"são muito menos frequentes [14].

Uma arquitetura baseada em microsserviços é uma arquitetura pensada para a nuvem que tem como objetivo produzir sistemas de *software* como um pacote de pequenos serviços, onde a implementação de cada serviço é independente da dos outros [14], podendo assim cada serviço ser independentemente desenvolvido e atualizado sem comprometer a integridade do ecossistema [16].

O uso desta arquitetura traz grandes benefícios tais como

- **Escalabilidade**: como os vários serviços são independentes uns dos outros, podemos redimensionar os serviços que são necessários ao invés de escalar a aplicação inteira;
- **Implantação e manutenção facilitada**: podemos tratar o código de cada serviço separadamente e depois fazer a implantação apenas daquela parte;
- **Isolamento**: erros ou alterações num dos servidos são independentes para o resto da aplicação.

Como em qualquer aplicação multi-camada, as arquiteturas de microsserviços integram sistemas de base de dados. Uma base de dados é uma coleção de dados computorizada

para ser facilmente acedida, administrada e atualizada. Para manter a alta disponibilidade tem de haver múltiplos nós de base de dados e em diferentes sítios. Para manter a mesma informação em todos os nós da base de dados, que se denominam de réplicas, tem de se sincronizar a informação entre elas. Para aumentar a escalabilidade de uma de base de dados pode-se particionar a informação entre as réplicas, isto é, uma réplica não tem a informação inteira de uma base de dados, mas apenas uma parte desta. Este processo é conhecido como particionamento.

1.2 PROBLEMAS

Tal como no exemplo de uma base de dados ou de uma orquestração baseada em microsserviços no geral, a sua monitorização é importante para detetar ou antecipar erros. Quanto mais rápido se detetar e/ou antecipar erros, menos prejuízo haverá.

Uma abordagem interessante é o SYSQUERY que permite a monitorização e supervisão dos recursos utilizados e as interações entre-processos. Num cenário de *clustering* no Kubernetes, esta abordagem irá recolher informação sobre cada um dos sistemas e o tipo e volume dos dados trocados entre sistemas. Esta informação será interessante para tentar permitir o melhor controlo de base de dados distribuídas no *Kubernetes* e de aplicações no geral. Contudo, o SYSQUERY por si só não oferece a possibilidade de visualização dos resultados da monitorização numa interface web e não possui qualquer funcionalidade para manipulação dos ficheiros de configuração do Kubernetes. Apesar de existirem diversas ferramentas para a apresentação de informação em ambiente web, nenhuma delas está preparada para tratar e expor alguma da informação recolhida pelo SYSQUERY.

1.3 OBJETIVOS

O objetivo principal desta dissertação é o desenvolvimento de uma plataforma para monitorização e controlo de sistemas na nuvem, denominada CloudOverWatch que tem como principais funcionalidades:

- Recolher e guardar informação de uma orquestração Kubernetes;
- Conseguir visualizar a informação recolhida em ambiente web;
- Interagir com cada um dos processos;
- Alterar diferentes parâmetros dos diversos controladores diferentes;
- Visualizar as alterações feitas à orquestração;

Além disso será importante comparar a aplicação desenvolvida com as diferentes alternativas existentes no mercado.

1.4 ESTRUTURA DO DOCUMENTO

Para além deste capítulo o documento contém também os seguintes capítulos:

- **Capítulo 2 - Estado da arte -** no qual são abordados os temas mais importantes relacionados com a dissertação a ser desenvolvida
- **Capítulo 3 - Desenvolvimento -** é exposto o problema, a proposta de solução e a sua implementação;
- **Capítulo 4 - Casos de Estudo / Resultados -** são apresentados vários resultados em diferentes cenários das diferentes propostas de solução do mercado e da desenvolvida.
- **Capítulo 5 - Conclusão -** é apresentada uma pequena conclusão do que já foi e será feito.
- **Apêndices / Anexos -** é onde são apresentadas várias informações complementares da dissertação.

2

ESTADO DA ARTE

Neste capítulo são apresentados e discutidos vários temas relevantes no âmbito da dissertação, analisando uma das formas mais comum, nos dias de hoje, de organizar e controlar aplicações em ambiente nuvem. Posteriormente discute-se como é feita a monitorização dessa mesma abordagem.

2.1 ORQUESTRAÇÃO

2.1.1 *Contentor e Imagens Docker*

Os contentores são pacotes de software que contêm tudo o que é preciso para um microsserviço correr, e.g. para um contentor com um microsserviço de *PostgreSQL*, vai conter o código, dependências e bibliotecas necessárias [12]. Assim, mesmo que duas ou mais pessoas tenham máquinas completamente diferentes, a aplicação vai correr de igual modo para todos.

Um contentor instancia uma imagem, logo, para se ter um contentor de PostgreSQL tem de se instanciar uma imagem PostgreSQL. Existem algumas alternativas relativamente à criação das mesmas, mas geralmente são usadas imagens Docker, que é o tipo de imagem que é usado no âmbito desta dissertação.

Considere-se o seguinte exemplo de código que configura uma imagem [1]:

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

Lista 2.1: Código exemplo de uma imagem Docker

Uma imagem Docker é organizada em camadas, em cada um daqueles comandos representa uma instrução que criará uma camada nova:

- **FROM** : criada uma camada da imagem *ubuntu:15.04*
- **COPY** : adiciona ficheiros do directório atual do utilizador

- **RUN** : faz *build* da aplicação, através do comando *make*
- **CMD** : específica que comando corre no contentor, neste caso o ficheiro app.py

Na Figura 1, é possível observar que ao criar o contentor para albergar a imagem, estamos a acrescentar uma nova camada no topo das outras camadas.

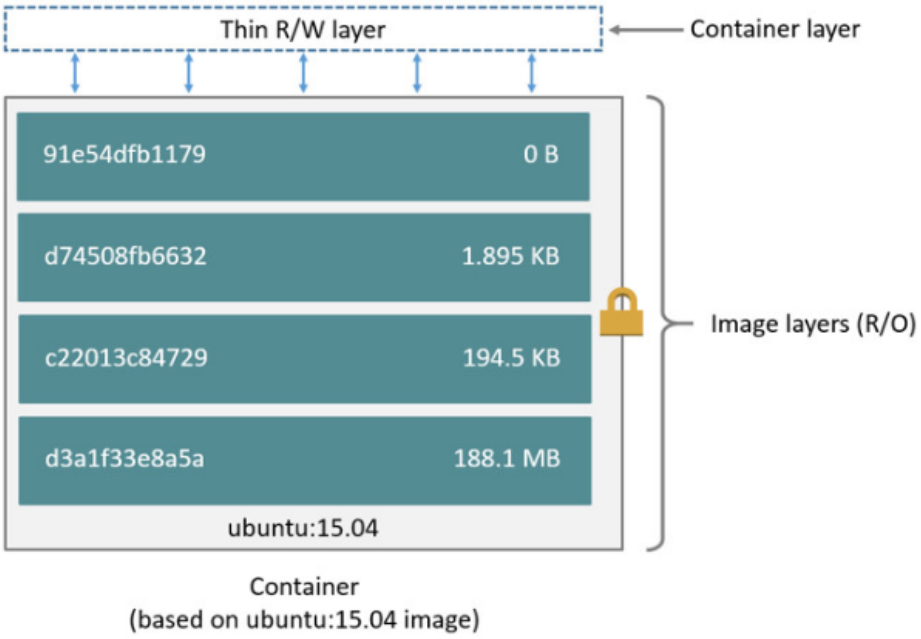


Figura 1: Exemplo das camadas da imagem e contentor [1]

Todas as mudanças (como escrever novos ficheiros, alterar ficheiros ou apagar ficheiros, etc) realizadas durante a execução do contentor serão apagadas quando o contentor for eliminado, mas as restantes camadas, pertencentes ao grupo "Image Layers"serão as mesmas. O contentor pode ser assim resumido a uma instância com estado de uma imagem. Quando é eliminado perde toda a nova informação, que no caso de um contentor com uma imagem PostgreSQL, seria perder toda a informação da base de dados, a não ser que esta informação seja persistida na máquina que alberga o Kubernetes.

Quantos mais contentores uma aplicação tiver, mais complexa se tornará controlar e organizá-la, por isso é necessário possuir uma ferramenta que permita gerir e orquestrar os contentores de uma aplicação, e é ai que entra o Kubernetes.

2.1.2 *Kubernetes*

O Kubernetes é um sistema de orquestração de contentores, que facilita a implementação, o dimensionamento e a gestão de aplicações em contentores, permitindo que os mecanismos

de controlo sobre os contentores sejam executados sobre o orquestrador e não sobre os componentes individualmente, acrescentando assim uma camada de abstração.

A principal razão pela qual foi optado por o Kubernetes a outros sistemas de orquestração foi o facto de o SYSQUERY ter sido desenhado a pensar no Kubernetes.

O Kubernetes executa a sua carga de trabalho ao colocar os contentores dentro de *pods* - unidade mais básica do Kubernetes - e os *pods* dentro de *nodes* - máquina física ou virtual. Um *pod* permite que um ou vários contentores corram dentro dele e cada *pod* encontra-se dentro de um *node*, que também permite que vários *pods* corram dentro dele. Os *nodes* podem ser uma máquina virtual ou física dependendo da orquestração e, cada nó, tem os serviços necessários para um *pod* conseguir correr.

A arquitetura base do Kubernetes está resumida na Figura 2.

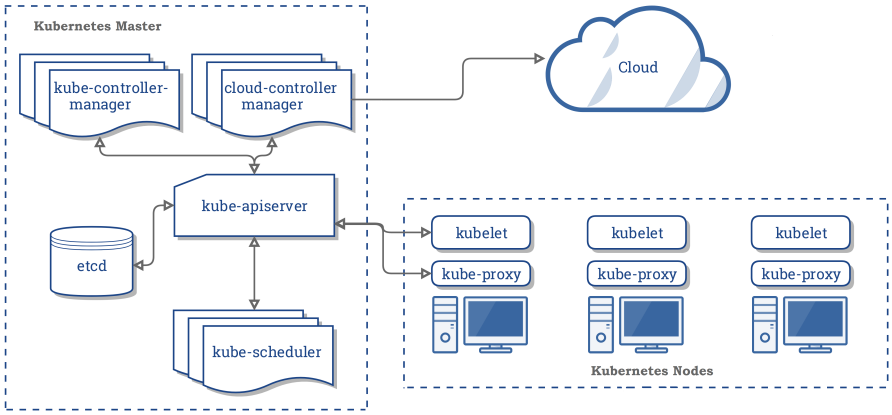


Figura 2: Arquitetura base do Kubernetes [8]

Como é demonstrado na Figura 2, a arquitetura do Kubernetes pode ser dividida em Kubernetes um *master* e em um ou vários nós (*Nodes*). O Kubernetes master é o componente principal de controlo, contendo os quatro componentes principais que têm como objetivo controlar e administrar os nós e contentores no sistema [15].

Os componentes do *master* são os seguintes:

- **etcd**: é um componente de armazenamento chave-valor consistente e de alta disponibilidade usado como repositório do Kubernetes para todos os dados do cluster [8].
- **scheduler**: componente que fica responsável pelo agendamento de cada *pod* a um dos nós do sistema [15].

- **API Server**: é responsável por receber comandos e manipular os dados para os objetos do Kubernetes, como *pods*. Os utilizadores podem enviar comandos para este componente atráves do *kubectl* (Kubernetes command line interface) [15].
- **Controller Manager**: componente que monitoriza o etcd e assim regula o estado do sistema.

Cada um dos nós tem vários componentes a correrem que ficam responsáveis por manter os *pods* a funcionar e criar um ambiente para os contentores correrem [8]. Os componentes de cada nós são os seguintes:

- **pods**: um *pod* é a unidade mais pequena, atómica, do Kubernetes. Quando é criado uma implantação no kubernetes, esta *implantação* cria um ou vários *pods* com contentores dentro de cada um dos *pods*, ao invés de criar os contentores diretamente. Cada um deste nós é enviado para um *node*.
- **kubelet**: é um agente que corre em cada *node* de um *cluster* e tem como função assegurar que os contentores estão a correr num *pod*;
- **container *Runtime***: é responsável pelo *runtime* dos contentores;

Para o bom funcionamento de um ambiente destes é preciso monitorizá-lo. O *Kubernetes* oferece um sistema de monitorização e gestão, mas que é insuficiente, pois tem em conta apenas a utilização de CPU para alterar o sistema. O utilizador pode definir um dos algoritmos disponíveis pelo Kubernetes, por exemplo quando a utilização do CPU de um *pod* ultrapassar os cinquenta por cento, o Kubernetes automaticamente acrescenta uma réplica. Existem muitas outras métricas que são importantes a ter em conta, como por exemplo, memória ou espaço no disco, que neste caso não são tidas em conta pelo sistema de monitorização, tornando-o assim obsoleto. Além disto também não em conta as ligações entre os vários *pods*. As alternativas a esta monitorização base do Kubernetes encontram-se na secção 2.2.

2.1.3 *VMs*

Como alternativa ao uso de contentores e da sua orquestração existe também a possibilidade de usar VMs para a implantação de uma aplicação.

Ambas as alternativas permitem empacotar os vários componentes. A maior diferença reside em termos de escalabilidade e portabilidade.

Quando falamos em contentores, falamos em algo mais pequeno, na razão dos Megabytes. Cada um não permite empacotar mais que uma aplicação e todos os ficheiros que são necessários para a correr. E, são geralmente usados para empacotar funções únicas, como

por exemplo uma instância de Base de dados, que realizam tarefas específicas. A natureza leve dos contentores e do Sistema Operativo compartilhado entre os diferentes contentores, torna-os muito fáceis de mover entre ambientes.

Quando falamos em VMs falamos na ordem dos GigaByte e geralmente contêm o seu próprio OS, permitindo que se executem várias funções que consomem muitos recursos ao mesmo tempo. Basicamente, permite simular que temos vários computadores dentro de uma só máquina.

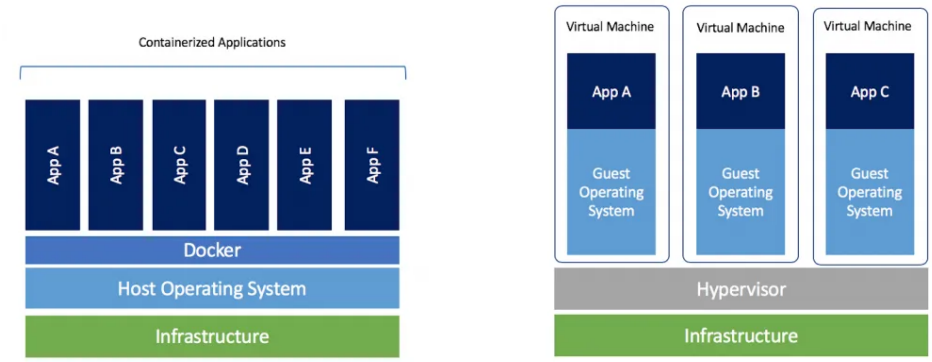


Figura 3: VMs vs Contentores [4]

Como é possível observar na Figura 3, a virtualização (uso de VMs) possui um software denominado *hipervisor* que cria e corre várias VMS e que separa os recursos de suas máquinas físicas para que possam ser particionados e dedicados às VMs [3].

Já os contentores contêm um microsserviço ou aplicativo e tudo que é preciso para este ser executado. Todas estas instruções e ficheiros são preservados num contentor através de uma imagem (ver secção 2.1.1). Como os contentores são pequenos, geralmente há centenas deles fracamente acoplados [3]. É por que quando maior for a aplicação mais importante é o uso de ferramentas de orquestração de contentores, para provisioná-los e geri-los.

De acordo com [3], os contentores são melhores quando queremos uma aplicação nuvem nativa, empacotar pacotes, incutir DevOps ou práticas de CI / CD num projecto ou mover projectos de IT escalonáveis numa área de IT diversificada que compartilha o mesmo sistema operacional. Já as VMs são melhores em cargas de trabalho tradicionais e monolíticas, quando se quer isolar ciclos de desenvolvimento arriscados, provisionar recursos de infra-estrutura (como redes, servidores e dados) ou executar um sistema operacional diferente dentro de outro sistema operacional.

Um dos problemas dos contentores face às VMs era a sua fraca performance, algo que está a ser ultrapassado. De acordo com o artigo "An Updated Performance Comparision of Virtual Machines and Linux Containers"[17], que avaliou a performance entre o uso de VMS e contentores em 2015, é comum se dizer que se queremos fazer uma IaaS devemos

usar VMs e se queremos fazer PaaS devemos usar contentores. Mas segundo este mesmo artigo esta afirmação não é necessariamente verdade porque não encontram nenhuma razão técnica para isto ser uma verdade absoluta com as mudanças que têm vindo sido feitas aos contentores, especialmente em casos que o uso de contentores para a construção de uma IaaS tem melhor performance.

2.1.4 *Outras Ferramentas de Automação*

Existem alternativas ao Kubernetes para a orquestração de contentores e, além disso, ainda existem outras ferramentas que têm um lugar bem presente no mercado da automação, mas que têm um papel diferente de ferramentas como o Kubernetes.

Como exemplo temos Ansible, Chef e Puppet, que ao contrário do Kubernetes não são exemplos de ferramentas de gestão da configuração automática para contentores, mas são ferramentas de gestão da configuração automática no geral, sem foco particular em contentores. Estas soluções - Ansible, Chef e Puppet - têm como objetivo automatizar a configuração de uma determinada máquina ou VM. Estes produtos não foram feitos para lidar com interações entre diferentes máquinas e microsserviços mas para manter a *CI/CD* de um projecto. Apesar de conseguirem configurar contentores, a sua prática não é recomendada. Por exemplo, não conseguimos dizer ao Chef para executar apenas um certo número de servidores de base de dados em um determinado momento e aumentá-los quando o uso da CPU atingir uma certa percentagem (e.g. 90%) e, em seguida, voltar a reduzi-los quando o uso da CPU for inferior a uma certa percentagem (20%). Nem o Chef nem o Ansible conseguem recuperar ou substituir em tempo real um contentor ou uma instância que foi desligada ou passou a não funcionar corretamente [2].

Em suma, estas ferramentas são usadas para controlar o ambiente à volta dos contentores e não aos contentores em si.

2.2 MONITORIZAÇÃO

Existem alternativas ao sistema de monitorização do Kubernetes que tornam a monitorização sobre as implantações do mesmo bastante melhor.

Um dos tipos de supervisão geralmente adotados um método que divide o processo de monitorização em três fases: uma primeira fase de recolha de informação sobre o sistema, numa segunda fase esta informação é guardada numa base de dados e por fim a informação é apresentada ao utilizador. Existem várias ferramentas para as diferentes fases do processo. Neste tipo de monitorização o objetivo é tirar o máximo de conclusões com base nas métricas recolhidas, como a utilização CPU e RAM.

Outro tipo de supervisão, que se baseia na monitorização orientada aos recursos, é a monitorização orientada à estrutura, que além de recolher estes recursos, tem um grande foco em demonstrar as conexões entre os diferentes processos, *pods* ou *hosts*, para facilmente perceber como a aplicação interage e, assim, mais rapidamente detetar possíveis problemas numa orquestração.

2.2.1 *Monitorização orientada aos recursos*

No processo de extração é comum o uso das ferramentas Prometheus[1] e Heapster[2], ambos sistemas de monitorização do desempenho e extração de informação compatíveis com o Kubernetes. O *Prometheus* permite também criar alertas baseados nas métricas recolhidas e possuí a sua própria interface. Os melhores resultados são obtidos quando combinado com outro sistema de apresentação de informação como o Grafana[3] [10].

No processo de armazenamento e procura é comum o uso de Elasticsearch[4] ou InfluxDb[5] e para o processo de apresentação de informação é comum o uso de Grafana ou Kibana[6], sendo estes últimos ambos produtos de software com o qual se consegue representar informação de monitorização, em ambiente web. O Elasticsearch e Kibana foram desenvolvidos para serem usados em conjunto, por isso é natural que se a escolha a nível de processo de armazenamento e procura for o Elasticsearch, a escolha natural para o processo de apresentação de informação recaia sobre o Kibana. Estes processos estão sintetizados na Figura 4.

1 https://prometheus.io/
2 https://github.com/kubernetes-retired/heapster
3 https://grafana.com/
4 https://www.elastic.co/pt/what-is/elasticsearch
5 https://www.influxdata.com/
6 https://www.elastic.co/pt/products/kibana

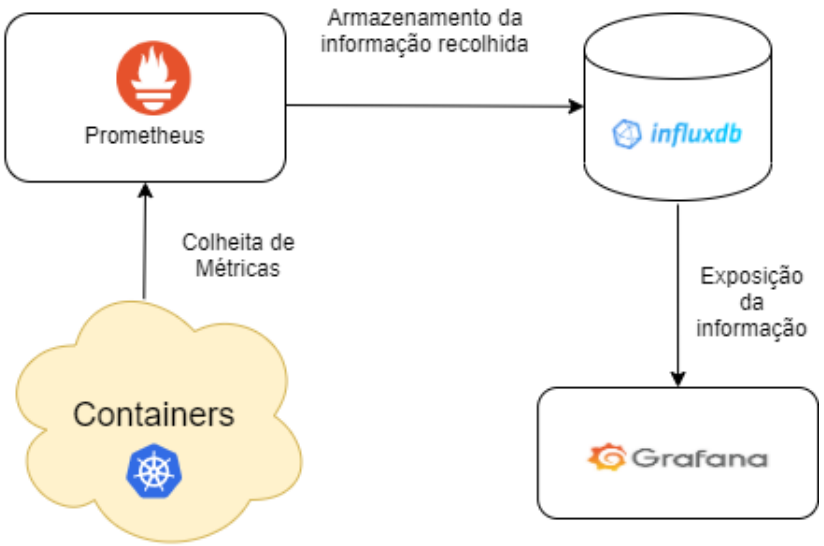


Figura 4: Exemplo típico de monitorização

Independentemente das ferramentas escolhidas o objetivo será ter uma melhor perceção do que está a acontecer no ambiente desenvolvido e utilizar essas mesmas informações para manipular e manter a melhor infraestrutura possível. Contudo nenhuma das ferramentas ou combinação das mesmas consegue detetar e quantificar a informação sobre a troca de dados entre vários processos quer dentro da mesma máquina quer entre máquinas. Para isso tem de se ter uma abordagem mais direcionada à estrutura.

2.2.2 *Monitorização orientada à estrutura*

Este tipo de monitorização difere da solução apresentada na Figura 4, principalmente na parte de exposição de informação, onde o foco não é mostrar as métricas sobre recursos recolhidos, mas sim a interação entre estes. Além disto, esta abordagem tende a também interagir diretamente com o Kubernetes, para alteração das configurações da orquestração, algo que não acontece com a abordagem anterior. Como exemplo, têm-se o WeaveScope.

O Weave Scope é uma ferramenta de administração e visualização de orquestrações *Docker* e *Kubernetes*. Esta ferramenta fornece uma visão sobre para uma aplicação bem como para toda a sua infraestrutura, e permite diagnosticar quaisquer problemas na orquestração, em tempo real e à medida que este está a ser implantado num provedor de nuvem.

O Weave Scope oferece uma vista sobre os processos, contentores, *pods* e *hosts* de uma orquestração, facilitando assim a descoberta de um problema [7]. As Figuras 5 e 6 demonstram alguns casos dessas mesmas vistas.

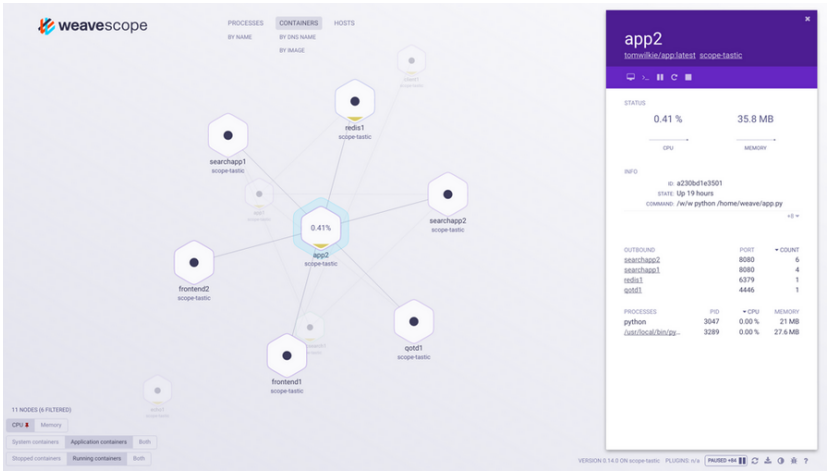


Figura 5: Vista dos contentores no Weave Scope

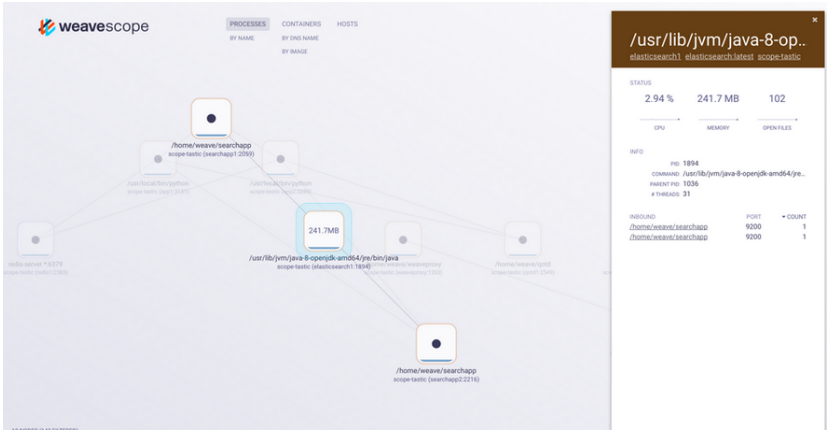


Figura 6: Mais detalhes do Weave Scope

Com o Weave Scope, o utilizador pode controlar todo o ciclo de vida de um contentor nos diferentes *hosts* do *cluster* a partir de uma única interface do usuário. Pode iniciar, parar, pausar e reiniciar contentores e executar comandos diretos na linha de comandos de cada uma das máquinas.

Apesar de se ter dito que o Weave Scope não consegue fazer o que o SYSQUERY consegue, Weave Scope é o que mais se aproxima do que se quer idealizar com esta dissertação, com o acréscimo de a aplicação desenvolvida disponibilizar a quantidade de dados trocados entre processos, *pods* e *hosts*. Não obstante, o Weave Scope é usado como base de comparação na parte de resultados - ver Capítulo 4.

Para que se conseguir quantificar a informação recolhida, é seguida a abordagem SYSQUERY [13].

Através da abordagem descrita em [13] é possível detetar os dados trocados entre os vários serviços existentes nos *pods* e entre *pods* e depois quantificá-la, com um mínimo de sobrecarga sobre o sistema. Esta abordagem baseia-se no uso de eBPF[7] para interceptar a informação desejada e depois conseguir agregar esta informação ainda dentro do *kernel* do sistema, de forma a minimizar a quantidade de dados que são copiadas para o *user mode*. Esta informação é guardada em Neo4j[8], para depois serem executadas queries sobre a mesma para representação gráfica da comunicação. Contudo o SYSQUERY por si só não possui nenhuma ferramenta quer para a visualização de dados quer para a manipulação de ficheiros de configuração de Kubernetes. Como nenhumas das ferramentas e/ou produtos de software acima descritos está preparado para interpretar alguma da informação recolhida pelo SYSQUERY, é apresentada uma solução para o problema, no capítulo seguinte.

O SYSQUERY recolhe vários dados sobre cada um dos processos e estes são recorrentemente mencionados ao longo desta dissertação, por isso seria necessário sumarizar os atributos mais importantes desses dados recolhidos.

- **pid** : número do processo;
- **rss** : memória RAM usada;
- **host** : instância em que o processo correu;
- **starttime e endttime** : timestamps de quando um processo começou e terminou de correr, que posteriormente são usados para calcular a percentagem de CPU;
- **cmd** : comando usado para correr processo;
- **comm** : tipo de processo a correr;
- **cont** : string de contém o uid do processo e o seu contentorID;

Além do SYSQUERY existem outras abordagens/ferramentas que têm implementações similares, mas que não conseguem alcançar o mesmo que o SYSQUERY quer porque precisam de informação prévia do sistema (e.g, não conseguem detetar os processos automaticamente), como é o caso de D-Trace e Magpie, ou porque, apesar de conseguirem automaticamente detetar os processos, não conseguem medir a informação trocada como é o caso do Weave Scope[9] [13]. Tal como o SYSQUERY, estas alternativas são monitorizações orientadas à estrutura.

7 eBPF - Extended BerkeleyPacket Filter - é uma tecnologia cada vez mais popular, para executar programas que passam do espaço do utilizador para o *kernel*
8 https://neo4j.com/
9 https://www.weave.works/oss/scope/

3

DESENVOLVIMENTO

No capítulo anterior foi identificado a abordagem que é usada, na aplicação desenvolvida, para monitorizar as orquestrações de Kubernetes e também as limitações desta abordagem: não possui uma parte para visualização da informação nem forma para alterar os ficheiros de configuração de Kubernetes. Neste capítulo, é discutida a proposta de solução que visa colmatar esses mesmos defeitos, as diferentes decisões tomadas e discutida a sua implementação, quer da aplicação e API quer de outras alterações feitas, através de texto, excertos de código e imagens. Para o desenvolvimento da aplicação e da correspondente API - para comunicar e modificar uma orquestração em Kubernetes - foram feitas inúmeras alterações ao SYSQUERY e ao Neovis, acabando por estas alterações fazerem parte do desenvolvimento da aplicação e, assim sendo, também serão discutidas neste capítulo.

3.1 PROPOSTA DE SOLUÇÃO

A proposta de solução passa por, depois de conseguir recolher a informação ser possível expor esta informação para o utilizador e utilizá-la de alguma forma, como por exemplo, conseguir alterar os sistemas de configuração do Kubernetes.

Na Figura 7, está desenhada a proposta de solução que tem quatro ações/funções principais:

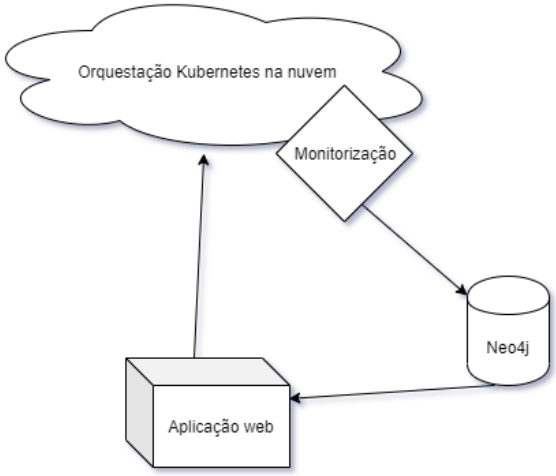


Figura 7: Desenho básico da solução do sistema

- A monitorização de uma orquestração em Kubernetes;
- A salvaguarda da informação recolhida numa base de dados de grafos, Neo4j;
- Visualização da informação recolhida em tempo real, desenvolvida em VueJS[1].
- Possibilidade de alteração dos ficheiros de configuração do Kubernetes dos contentores da orquestração, de acordo com informação recolhida;

Esta arquitetura permite com a mesma facilidade que se possa perceber que existe um problema numa orquestração também seja possível atuar para mitigar esse problema e, depois, ver os resultados dessa atuação. Esta solução é ótima em casos em que, por exemplo, se detete um tráfego anormal de dados e a partir daí atuar rapidamente de forma muito simples. Por exemplo, numa orquestração em que tem vários *pods* a comunicar uns com o outro e de um momento para o outro começa a haver um maior fluxo de dados para um *pod* apesar do número de conexões ou novos processos não ter aumento significativamente - possivelmente os utilizadores fizeram ações mais complexas. A aplicação ao detetar isto, dá a entender ao utilizador que poderá haver um possível problema. Assim o utilizador pode rapidamente tentar resolver o problema, através da comunicação direta existente entre a aplicação com o Kubernetes, por exemplo, aumentando o número de réplicas do *pod* em sobrecarga. Depois é feita automaticamente uma nova monitorização, para perceber se o que foi feito é suficiente para colmatar o problema.

Comparativamente com outras soluções teria-se de esperar por um aumento de outras métricas (como uso de CPU ou RAM) e as ações tomadas poderiam já ser mais tardias. O WeaveScope também só detetaria mais tarde já que o número de ligações não aumentou

1 https://vuejs.org/

significativamente, e esta aplicação não consegue detetar a quantidade de tráfego trocado, apenas o número de ligações.

3.2 SYSQUERY

O SYSQUERY recolhe vários dados sobre os variados processos presentes na orquestração a ser analisada. Esta informação é crucial para o desenvolvimento de toda a aplicação, porém, precisa de ser ligeiramente alterada, para ter um melhor enquadramento com o utilizador final.

O SYSQUERY cria uma rede onde cada processo se liga a uma *socket* que por sua vez se liga a um outro processo. Claro podem existir múltiplas ligações entre os processos, onde o que difere é a *socket* utilizada por isso a *query* utilizada junta todas as sockets numa só, de forma a ter uma única ligação entre os nós com o número total de bytes trocados.

$$(p1 : Process) - [r1 : Connectedto] - > (s1 : Socket) < - [r2 : Connectedto] - (p2 : Process)$$

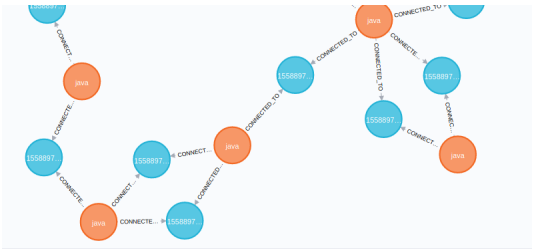
O facto de existir um nó (Socket) entre cada um dos *pods*, fazia com que rapidamente a experiência visual fosse desagradável com o número acrescido de nós na rede, tornando, assim, a sua eliminação uma prioridade para uma melhor User Experience (UX).

Para isso tem de se modificar o código para eliminar esses nós intermédios. A abordagem eleita foi que quando o SYSQUERY acaba de criar toda a rede de nós, de seguida, passa a criar uma nova relação que junta todos as *Sockets* entre dois nós numa só ligação sem a *socket* entre os nós, através do método *merge* do Neo4j. Assim a relação *NoSocket* contém o número total de *bytes* trocados entre os dois processos.

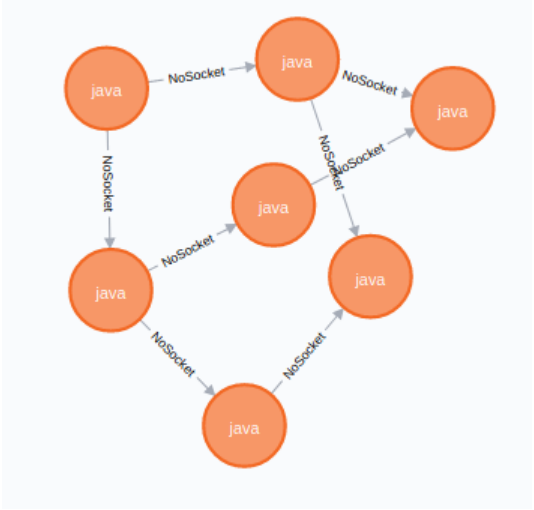
```
1 match n=(p1:Process)-[c1:CONNECTED_TO]-(s1:Socket)
2 match n1=(p2:Process)-[c2:CONNECTED_TO]-(s2:Socket)  where s1=s2 and p1<>p2 and c1.sent_bytes is
      not null and c2.sent_bytes is not null with sum(c1.sent_bytes+c2.sent_bytes) as str,p1,p2
3 merge (p1)-[arr:NoSocket2{sent_bytes:str}]-(p2) return type(arr)
```

Lista 3.1: Código para eliminar os nós intermidiários

A Figura 8 mostra a diferença entre a versão sem o código do SYSQUERY alterado e com o SYSQUERY alterado.



(a) Com Sockets



(b) Sem Sockets

Figura 8: Exemplo retirado diretamente do Neo4j - Com Sockets vs Sem Sockets

3.3 NEOVIS

O Neovis é um plugin que se liga a uma base de dados Neo4j e constrói, através de uma biblioteca chamada Vis.js, gráficos idênticos aos que encontramos no Neo4j. Assim é possível visualizar em ambiente web as bases de dados de Neo4j, através da rede de nós criada pelo *plugin*.

A versão oficial do Neovis, que é *open-source*, ainda tem muitas limitações e não usa por completo o Vis.js, por isso tiveram de ser feitas várias mudanças ao longo do tempo, para se ter a melhor configuração tendo em conta os objetivos definidos.

Assim, tem de ser criada uma nova versão do Neovis que juntamente com a aplicação implementada em VueJS, com as seguintes funcionalidades adicionais:

- Possibilidade de haver imagens associadas aos processos;
- Possibilidade de o utilizador organizar a visualização por instância ou por processo;
- Adicionado um evento de DoubleClick aos nós na rede;
- A forma como o atributo comunidade é definido foi modificada;
- Adicionada legenda para as várias instâncias;

3.3.1 *Comunidade e Legenda*

Um dos atributos que este *plugin* permite configurar é a comunidade. A comunidade atribui diferentes cores aos nós dependendo da sua comunidade. Tendo em conta a informação recolhida pelo SYSQUERY (ver secção 2.2.1), faz sentido que a comunidade seja o *host*. Porém este atributo é uma string (e.g. "instance-1") e o Neovis está à espera de um valor numérico.

Assim, é feita a primeira alteração ao Neovis, que agora passa a fazer *parse* do valor que recebe e é retirada a parte final para atribuir a cor aos grupos. Deste modo, fica possível atribuir a cada nó a respetiva cor dependendo da sua instância. Para estabilidade do código, é atribuído o valor zero no caso de a instância possuir um valor que seja inválido.

```
let temp = neo4jNode.properties[communityKey].split("-")
let number = temp[1]
if (neo4jNode.properties[communityKey]) {
  node.group = number || 0;
} else {
  node.group = 0;
}
```

Lista 3.2: Código para atribuição da cor ao nó

Apesar, de agora, com as diferentes a informação fosse mais apelativa, poderia ser ainda impercetível, para o utilizador, associar imediatamente que cada cor correspondia a uma instância. Seria necessário adicionar uma legenda. Assim, para novo nó adicionado à rede passou a ser retirada a sua instância e guardada num *array*, no caso ainda de não existir, ficando no final com um *array* com o número de todas as instâncias.

Com esta informação, só falta adicionar os novos nós à rede de forma estática, isto é, sempre no mesmo sítio. Os nós que pertencem à rede são dispersos aleatoriamente na rede e posteriormente atraídos uns pelos outros, através da força de atração (discutido na secção 3.3.4). Desta forma, a sua posição muda a cada *refresh*, enquanto que estes novos nós teriam de estar sempre no mesmo sítio, ou seja, teriam de ser adicionadas coordenadas específicas para cada um deles e como não tinham ligações com outros nós, a força de atração era nula. Para os diferenciar ao criar o nó foi adicionado um novo atributo a cada nó, *fake*.

Deste modo, aquando a criação da rede, é verificado se o nó é *fake* ou não, isto é, se o nó que está a ser adicionado faz parte dos nós que pertencem à rede ou são nós criados posteriormente com o intuito de fazerem quer parte ou da legenda ou do grupo de nós de atração (será discutido mais à frente, na secção 3.5) e, assim, dispor os nós corretamente. Enquanto que os nós que fazem a rede estão dispersos à sorte na janela estes novos nós têm

em conta o tamanho máximo da janela e são dispostos nessa extremidade, como é possível observar na Figura 9.

```
1 var mynetwork = document.getElementById('viz');
2 var x =  mynetwork.clientWidth;
3 var y = - mynetwork.clientHeight;
4 var step=120;
5 var a=parseInt(i)
6 var mult=step*(i-1);
```

Lista 3.3: Código para dispersão de nós

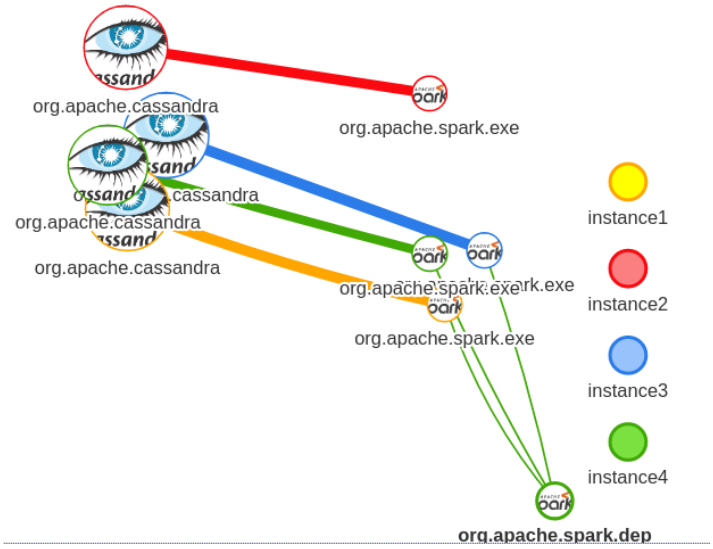


Figura 9: Legenda no CloudOverWatch

3.3.2 *DoubleClick*

O Neovis disponibiliza toda a informação que um nó possui quando passamos o rato por cima de um nó.

Contudo, por vezes, pode se tornar difícil de ler, por isso seria interessante que quando o utilizador clicasse duas vezes no nó um modal surgisse e mostrasse a informação de uma forma mais organizada. Não obstante, foi referido que se quer modificar os atributos do controlador de um *pod*, portanto, neste modal faz sentido haver uma secção, onde é possível editar atributos do *pod*:

- réplicas - número de réplicas de um *pod*;

- limite de CPU e memória - limite máximo que um contentor pode usar de um recurso - de CPU ou memória;
- *request* de CPU e memória - é os recursos que um contentor é garantido ter; ajuda o Kubernetes a decidir para que *node* um *pod* deve ir.

A unidade usada para definir o limite ou *request* de cpu é millicpu. Por exemplo, de definirmos um request de um *pod* como **250m**, estamos a dizer que para cada *pod* daquele tipo terá de haver 1/4 de um core disponível para ele correr. Já a memória é medida em bytes.

Para conseguir este objetivo a configuração do Neovis tem de ser alterada para que através de eventos disponibilizados pelo Vis.js, se consiga adicionar o evento doubleclick, que passa a informação necessária ao modal e fá-lo surgir. É também necessário verificar se o nó clicado faz parte da legenda, que em caso afirmativo, nenhum modal é aberto.

```
var config = {
    [...]
      Process: {
        caption: "cmd",
        size: "cputime",
        community: "host",
        clickEvent: properties => {
          var cmd = properties.properties
          if(properties.properties.fake==false){
            view.show(properties.properties.cont)
            view.choosenNode=properties.properties.cmd
            view.fill(properties.properties)
          }
        }
      }
},
```

Lista 3.4: Código para preencher e abrir modal

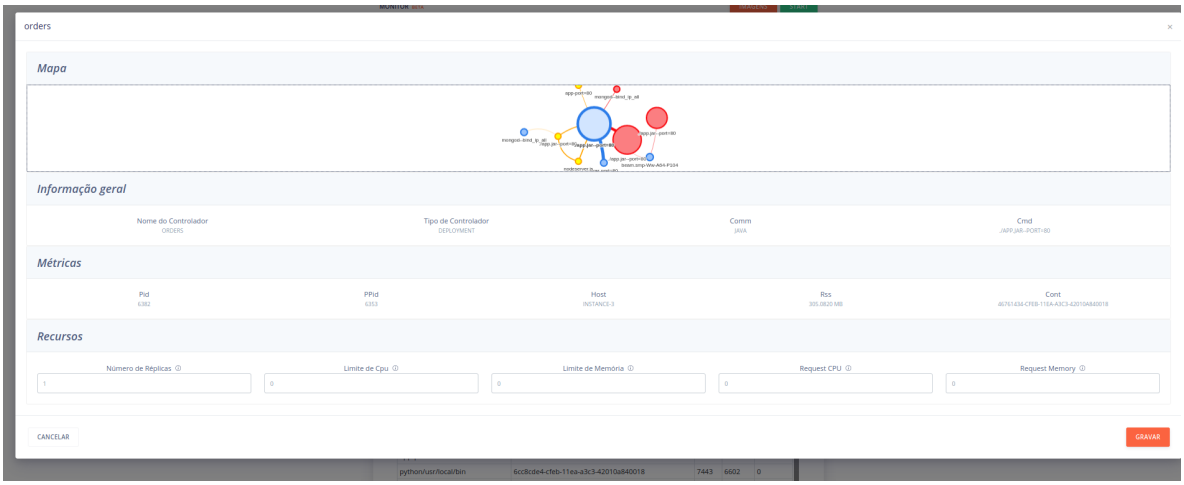


Figura 10: Modal apresentado para cada nó

Por fim, no modal é adicionado um mapa, que é uma sub-rede da principal, que contém apenas os processos a que o processo que o utilizador clicou está ligado.

3.3.3 *Imagem em processos*

O Neovis disponibiliza a possibilidade de adicionar texto por debaixo do nó, de forma o identificar (que neste caso é o atributo cmd). Esta solução foi adotada, mas, quanto maior for a rede mais difícil será identificar o tipo de processo a que o nó corresponde.

O Vis.js tem um atributo para os nós que é *image*. Com este atributo é possível atribuir uma imagem a cada nó de forma a o identificar.

Para a identificação dos processos tem de existir um array que contém uma chave (e.g. Postgres) e um url para uma imagem. Para cada nó é verificado se o cmd dele contém algumas das palavras chave e, em caso positivo, o processo além de ter a cor da instância passa também a ter uma imagem representativa do seu processo, como se pode observar na Figura 12.

```
1  //neovis.js
2  node.image=assertImage(array,node.originalProperties.cmd)
3
4  //image.js
5  export function assertImage(array,name){
6      for(var key in array){
7          if(name.includes(key)){
8              return array[key]
9          }
10     }
```

```
11      return false
12
13  }
```

Lista 3.5: Código que atribui uma imagem a um processo

Como não se pode cobrir todos os processos possíveis, o utilizador tem a possibilidade de adicionar novos processos com a respetiva imagem, como está representado na Figura 11.

Assim, sempre que o **cmd** do processo contém uma das chaves, a imagem do nó é modificada para a associada à chave. De notar que o cmd de um nó pode ser muito grande e com um maior e mais específico número de chaves-valor, um cmd pode conter mais que uma chave. Na eventualidade de isto acontecer, será sempre atribuído a primeira chave-valor encontrada.

Figura 11: Modal para acrescentar imagem de um processo

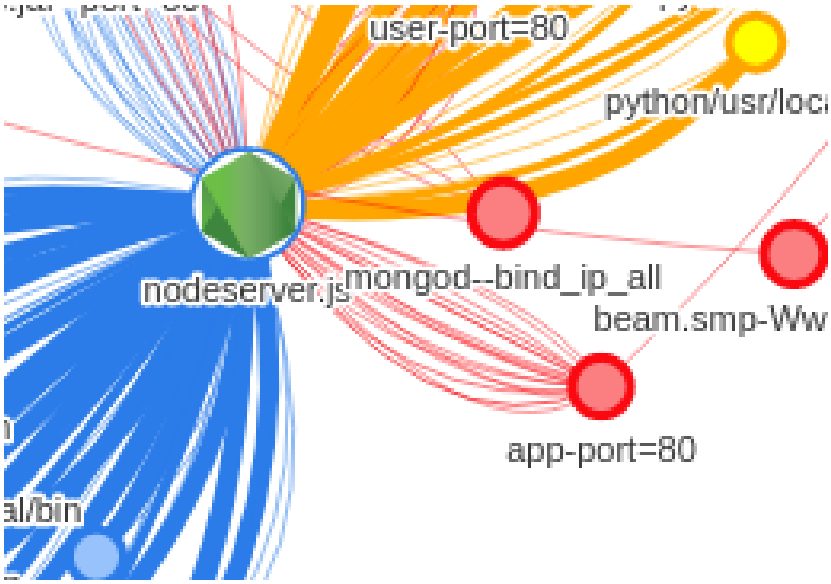
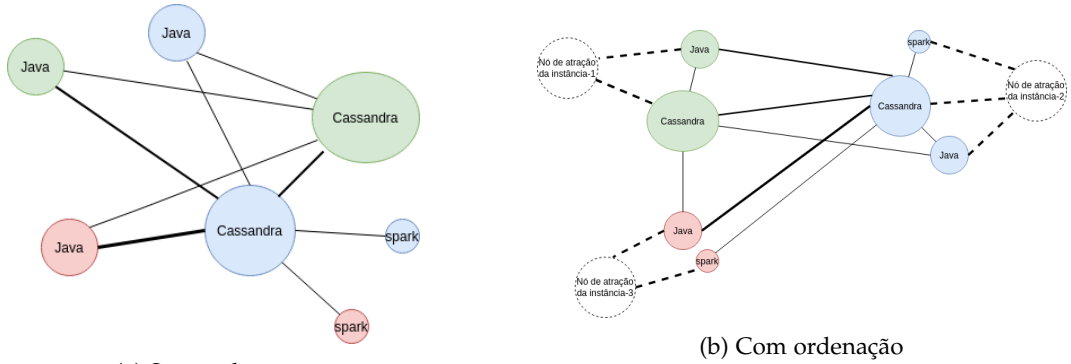


Figura 12: Exemplo de processo com imagem

3.3.4 *Dispersão de nós em processos*

Como foi referido em 3.3.1 , os nós são dispostos aleatoriamente, mas são posteriormente atraídos seguindo um conceito, existente no vis.js, denominado força de atração. Essa força depende do número de nós que estão ligadas a um nó (quanto maior o número de nós ligados a um nó, maior será a sua força de atração), onde os nós com maior força de atração têm tendência a estar mais no meio da rede. Apesar de haver certa forma uma organização na aleatoriedade, do ponto de vista de percepção de informação por parte do utilizador, não significa que seja necessariamente bom. Seria mais favorável, por exemplo, que os diferentes processos estivessem juntos de acordo com a sua instância.

Para se conseguir organizar por instância tem de se ter em conta a física dos nós, a tal força de atração. A disposição dos mesmos depende das ligações com os outros nós, para tal tem de existir um nó por cada instância que tenha uma força de atração muito grande, isto é, todos os nós daquela instância tinham de se ligar a este. E do ponto de vista do utilizador, este nó tinha de ser invisível - esta parte é bastante fácil de implementar já que uma das propriedades dos nós presentes no Vis.JS é *hidden*, que quando a **true** o nó fica invisível. Na figura 13 encontra-se um exemplo teórico do que se quer obter.



(a) Sem ordenação

(b) Com ordenação

Figura 13: Exemplo teórico das diferenças de com e sem nó de atração

Assim, o primeiro passo é criar para instância um nó nas extremidades da janela (para maior facilidade de organização). Para o conseguir, ao criar os nós tem de ser também criado um objeto que guarda a instância e o respetivo nó.

```
1 var str= node.originalProperties.cmd.split(" ")
2 var temp = str[0]
3 if(this.info[temp]==false){
4     this.info[temp]=[node.id]
5 }else{
6   var a = this.info[temp]
7   var b = false;
```

```
8   if(a.length!=0){
9     for(var n=0;n<a.length;n++){
10      if(node.id==a[n]){
11        b=true;
12        break;
13      }
14    }
15    if(b==false){
16      this.info[temp].push(node.id)
17    }
18  }
19 }
```

Lista 3.6: Código para dispersão de nós

No final ficámos com algo como:

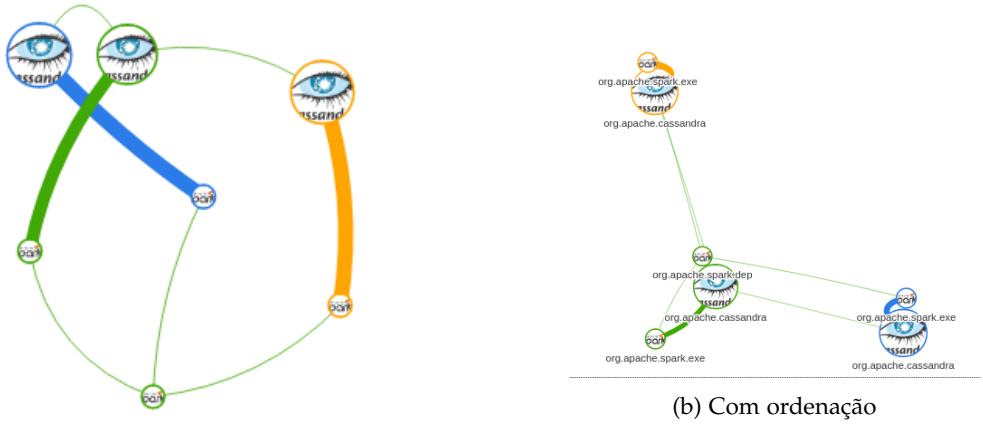
```
1 {
2 1:[1,2,3,4]
3 2:[5,6,7.8]
4 3:[9,10,11,12]
5 4:[13,14,15,16]
6 }
```

Lista 3.7: Exemplo do objeto criado

Agora, já se possui toda a informação para a criação das ligações, em que cada nó no array criava uma aresta entre si e o nó da sua instância.

```
1 createEdge(i,id,process){
2 [...]
3     var a = "instance-"+i
4       var arr = this.info[a]
5       for(var n=0;n<arr.length;n++){
6         var buff=arr[n]
7         var edge= {"id":99+buff,"from":id1,"originalProperties":{"sent_bytes":{"low":132+buff,"
              high":0}},"value":132+buff,"label":"","hidden":true}
8
9         this._addEdge(edge)
10    }
11 [...]
```

Lista 3.8: Código para criação de arestas



(a) Sem ordenação

(b) Com ordenação

Figura 14: Diferenças do uso ou não uso de ordenação por instâncias

Como podemos ver na Figura 14, é notória a diferença, entre a vista organizada e a vista desorganizada, sendo muito mais fácil de perceber se um possível problema, reside entre instâncias ou dentro das mesmas. É de notar que ficou também muito similar com o exemplo teórico (Figura 13), como se queria.

Outra vista que foi pensada foi a vista por processos. Ou seja, cada nó dispersa-se na rede de acordo com o seu processo e não com a sua instância, sendo usado o atributo cmd como comparação e não o atributo *host*. A metodologia a usar é a mesma, ao criar os nós cria-se também um objeto que contém o atributo cmd recolhido pelo SYSQUERY e os processos que continham um cmd idêntico.

```
{
    "java":[1,2,3,4]
    "postrgres":[5,6,7.8]
    "cassandra":[9,10,11,12]
}
```

Lista 3.9: Exemplo do objeto criado para ordenação por processo

Posteriormente, são criadas as arestas entre os nós invisíveis e os nós correspondentes. Na figura 15 pode-se ver o resultado final.

Assim passa a existir um novo atributo na renderização denominado *view* que pode variar entre:

- normal : vista por instâncias;
- processo: vista por processos.

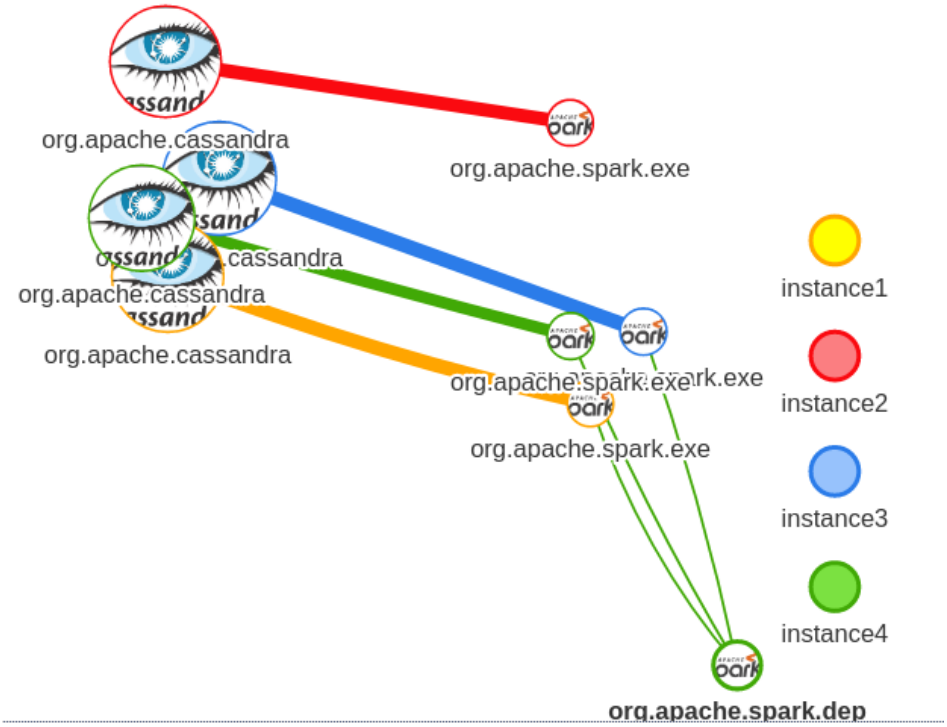


Figura 15: Exemplo do uso da ordenação por processos

3.4 APLICAÇÃO

Neste capítulo são sumarizadas todas as funcionalidades da aplicação, e ao fazê-lo, é introduzido o elemento final que completa o ciclo Kubernetes-Aplicação-Kubernetes, isto é, como a aplicação comunica com a orquestração Kubernetes, para manipular os recursos atribuídos aos *pods*. As funcionalidades vão ser apresentadas ao nível da aplicação e não tão extensivamente ao nível do *back-end*. As funcionalidades que requeiram uma descrição mais pormenorizada a esse nível, serão apresentadas nas secções seguintes.

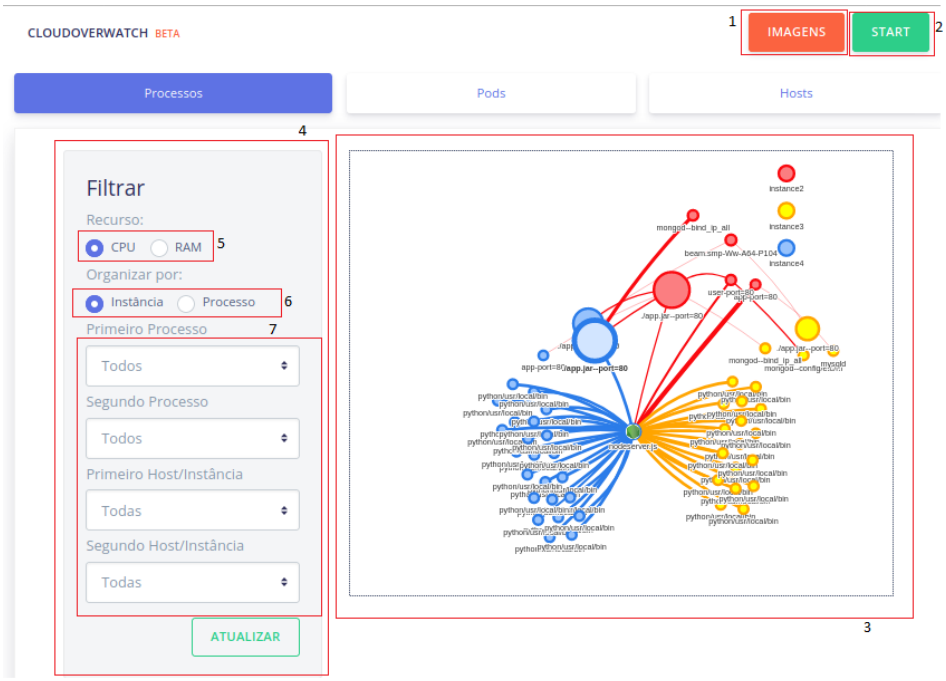


Figura 16: Vista principal da aplicação

Na Figura 16 está representada a vista principal da aplicação, que contém quase todas as funcionalidades ao nível dos *pods*.

- 1: Botão que abre modal para atribuir uma imagem a um processo;
- 2: Botão que começa e pára a monitorização sobre a orquestração Kubernetes:
- 3: Quadro principal onde tem temos todas as configurações possíveis;
- 4: Secção onde se encontra a rede de nós;
- 5: O volume do nó pode ser definido quer pela RAM ou CPU usados durante o tempo que o sistema teve a ser monitorizado;
- 6: Vista pode ser ordenada por instância ou processo;
- 7: Pode personalizar os nós que se quer ver ao mudar o número de processos e/ou instâncias que queremos ver.

Na figura 17 está representada a tabela dos processos, que contém a informação mais sintetizada.

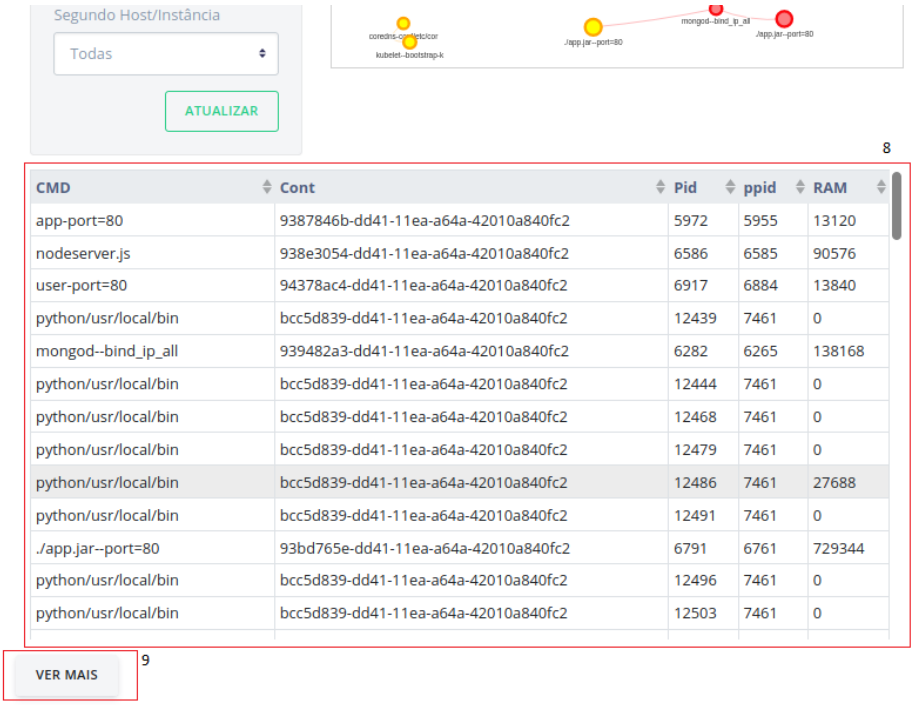


Figura 17: Vista principal da aplicação - Tabela de processos

- 8: Tabela que contém a informação mais importante de cada um dos processos;
- 9: Botão para abrir modal que contem mais informação sobre cada um dos processos.

Na figura 18 está representada a vista principal ao nível dos *pods*.

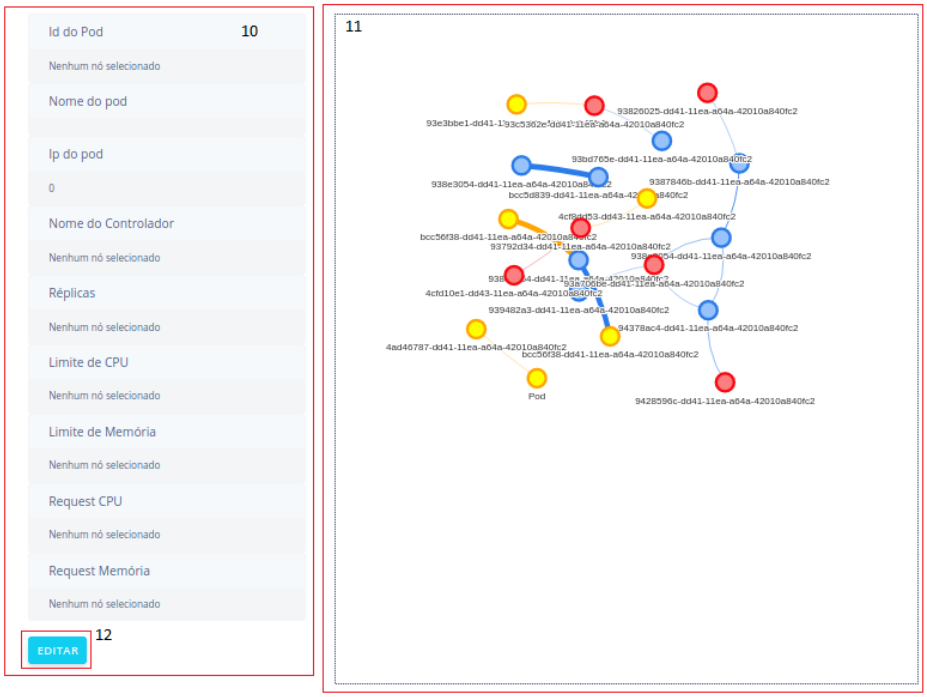


Figura 18: Vista principal da aplicação - *Pods*

- 10: Quadro que contém a informação sobre o *pod* selecionado;
- 11: Botão que abre modal para atribuir uma imagem a um processo;
- 12: Secção onde se encontra a rede de nós, dos *pods*.

Na figura 19 está representada a tabela dos *pods*, que contém a informação mais sintetizada.

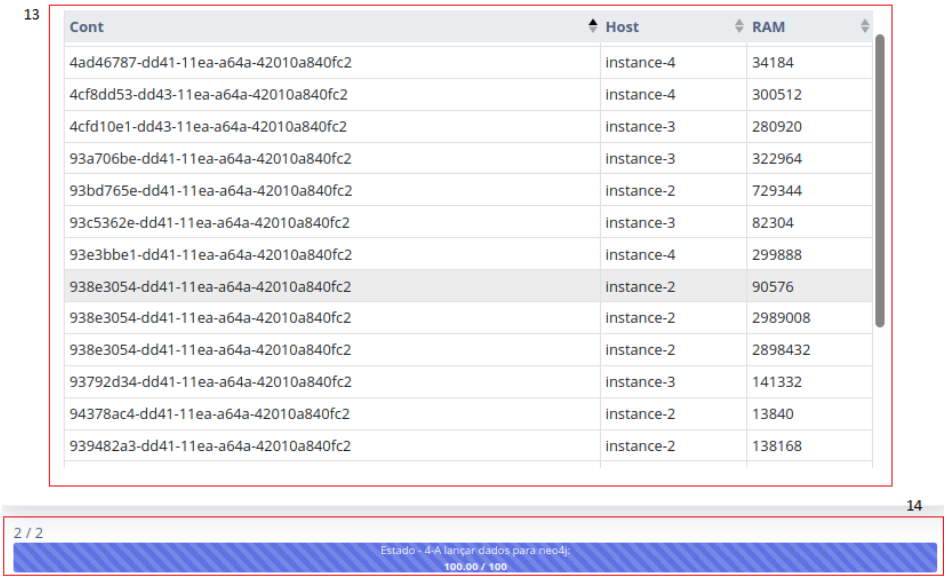


Figura 19: Vista principal da aplicação - Tabela de *pods*

- 13: Tabela que contém a informação mais importante de cada um dos *pods*;
- 14: Barra de progresso das alterações ao Kubernetes (e.g. mudar o número de réplicas).

Na figura 20 está representada a vista principal ao nível dos *hosts*.

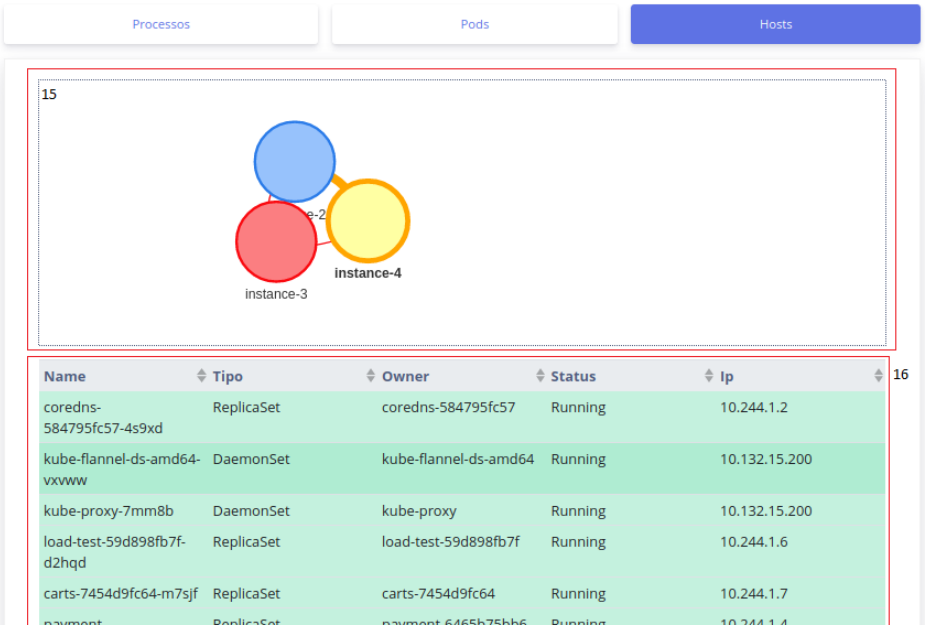


Figura 20: Vista principal da aplicação - *Hosts*

- 15: Secção onde se encontra a rede de nós, dos *hosts* ;
- 16: Tabela que contém a informação mais importante de cada um dos *pods* ligados a um *host* e mostra o estado atual dele.

3.4.1 *Definir instância e processo*

Na secção 3.3.4 é referido que é possível personalizar a vista ao nível dos processos, escolhendo as instâncias e/ou processos que queremos ver, sendo importante perceber como isso acontece. Além disso, foi referido em 3.2, que foram retiradas as sockets, passando a haver um novo tipo de ligação denominado NoSocket. Assim, importa perceber como é que se consegue ter as vistas personalizadas neste novo tipo de ligação, usando o Neo4j.

A query usada para carregar a rede inteira de nós é a seguinte: *match p=(p1)-[n:NoSocket]-(p2) return p*, isto é, faz *match* dos nós que estejam ligados pela NoSocket e retorna p, a rede. O utilizador pode editar esta rede para a sua preferência ou necessidade, usando o painel lateral disponível da secção dos processos (presenta na Figura 16 - ponto 7).

Ao organizarmos por instância ou por processo estamos a modificar a query, para que tenha em conta essas particularidades. Existem inúmeras hipóteses, mas para perceber olhemos para os seguintes exemplos:

- *match p=(p1)-[n:NoSocket]-(p2) where p1.host contains "instance-1"and p2.contains "instance-2"return p*: retorna só os *pods* que estão ligados entre a instância 1 e instância 2
- *match p=(p1)-[n:NoSocket]-(p2) where p1.host contains "instance-1"and p2.cmd contains "python"return p*: retorna os processos da instância 1 que se ligam a processos *python* de qualquer instância.

3.4.2 *Monitorização*

A aplicação permite que a qualquer altura seja possível fazer uma monitorização manual do sistema, isto é, definir quando a monitorização começa e quando acaba, pelo tempo que o utilizador quiser. O utilizador começa por clicar em **Start** (Figura 16 - ponto 2) e a aplicação comunica com a API, que por sua vez vai enviar e iniciar os agentes para a orquestração, de forma a começar a monitorizar a orquestração. Quando o utilizador clica em **Stop** (no mesmo botão em clicou *Start*), é enviado um novo pedido à API que vai fazer parar a monitorização e começar a ser gerada a nova versão da base de dados, que passado uns segundos vai atualizar a página, para apresentar o estado do sistema durante a monitorização.

3.4.3 *Alteração de recursos*

É possível alterar vários recursos destinados a cada *pod*, ao alterador o controlador deste. O utilizador clica num dos nós e, como é possível observar nas Figuras 21 e 22, é gerado um modal com toda a informação sobre o processo e o controlador deste e onde é possível alterar o número de réplicas, o limite e *request* de CPU e de memória alocados para o controlador deste processo. A aplicação comunica com a API que por sua vez comunica com o Kubernetes, alterando assim o *yaml* correspondente. Ao mesmo tempo vai dando resposta ao utilizador de cada passo - estado - que está a acontecer através de uma barra de progresso (ver Figura 23). Os estados possíveis de uma alteração são os seguintes:

- **Estado 1** - *A executar modificações* - estado onde todas as modificações a um controlador estão a ser feitas;
- **Estado 2** - *A começar a monitorização* - passa para este estado quando todas as mudanças foram feitas; começa a monitorização;
- **Estado 3** - *A parar monitorização* - passa para este estado ao final de 40 segundos, ou seja, é feita uma monitorização de 40 segundos;
- **Estado 4** - *A lançar dados para neo4j* - passa para este estado depois da monitorização acabar para lançar todos os dados recolhidos para o neo4j.

A forma como é feita toda a comunicação será explorada na secção 3.5.

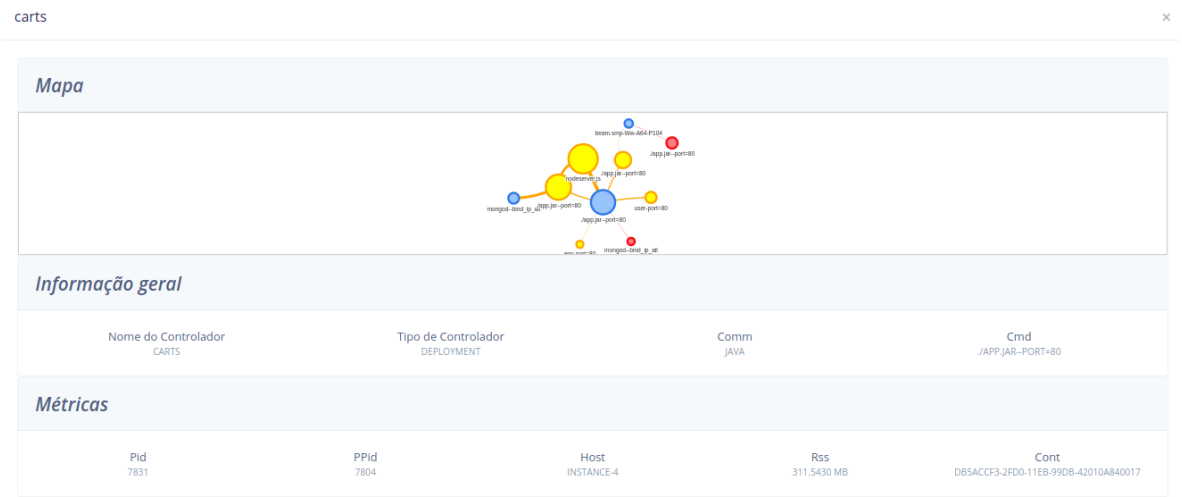


Figura 21: Modal de cada processos - informação

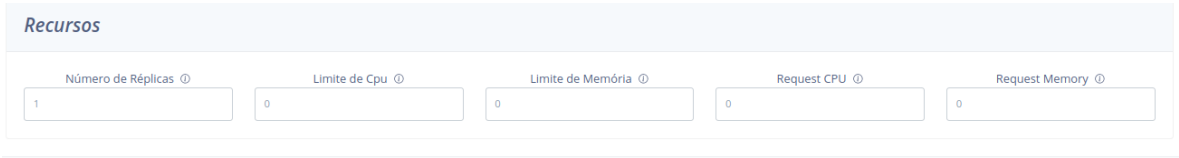


Figura 22: Modal de cada processo - alteração de recursos

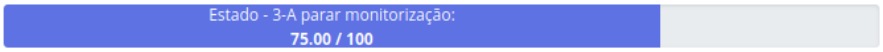


Figura 23: Barra de Feedback

3.4.4 *Vista sobre os Pods e Hosts*

A vista principal do CloudOverWatch é sobre todos os processos presentes na orquestração, os recursos usados aos mesmos e a troca de informação entre eles. Mas quanto maior for a aplicação maior será o número de processos a serem apresentados, o que faz com seja necessário apresentar outro tipo de vistas para ser mais fácil atuar sobre a orquestração.

A primeira vista a ser criada é uma vista sobre os *pods* da orquestração. Para isso passa a ser criada uma nova rede em que cada nó era a junção de todos os processos com o mesmo uid do *pod* e as arestas o somatório da informação trocada.

```
1 MATCH n=(p1:Process)-[ns1:NoSocket2]-(p2:Process) where p1.cont<> p2.cont with sum(ns1.
      sent_bytes) as str,p1.cont as over,p2.cont as over2
2 MERGE (p3:Pod{cont:over})
3 MERGE (p3:Pod{cont:over2})
4 MERGE (p3)-[novo:Pod{sent_bytes:str}]-(p4)
```

Lista 3.10: Query para criar vista sobre hosts

Como a criação da rede é feita através da informação recolhida pelo SYSQUERY, que só nos dá o *uid* do *pod* e não o nome dele, tem de ser apresentada na vista o *uid* dele em vez do nome dele. Para conseguir obter o nome desse *pod* é necessário clicar no *pod*, para que a aplicação faça um *request* ao Kubernetes para associar um nome ao *pod* e mostrá-lo no painel lateral da vista.

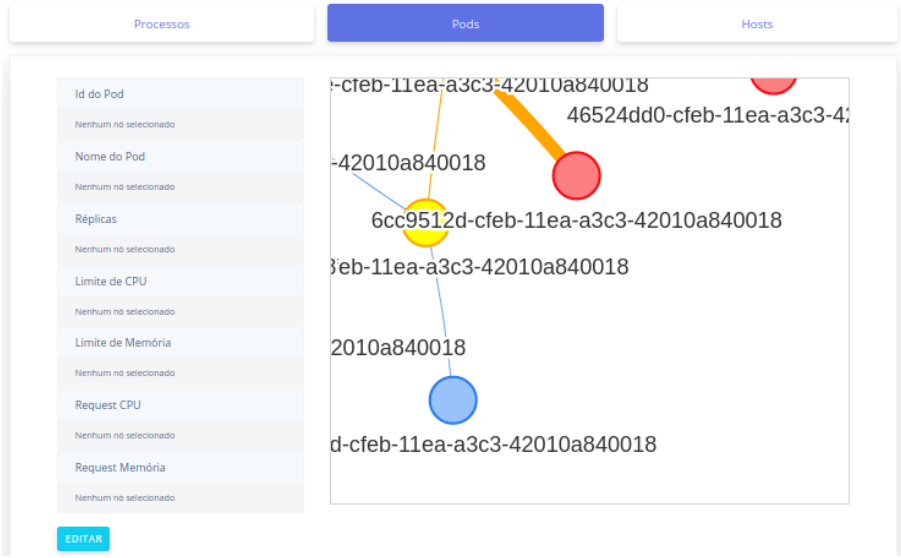


Figura 24: Vista dos pods

Da mesma forma, que o utilizador poderá querer ver a orquestração organizada por *pods* também poderá querer ver a troca de informação ao nível da instância, para perceber se algumas das máquinas estará, eventualmente, em sobrecarga ou quantos *pods* essa instância tem ativos ou com algum problema.

```
1 MATCH n=(h1:Host)-[hp1:HAS_PID]-(p1:Process)-[ns1:NoSocket2]-(p2:Process)-[hp2:HAS_PID]-(h2:Host
      ) where h1<>h2 with sum(ns1.sent_bytes) as sumatorio,h1,h2
2 MERGE (h1)-[novo:HostConnections{sent_bytes:suamtorio}]-(h2)
```

Lista 3.11: Query para criar vista sobre *pods*

Clicando em cada uma das diferentes instâncias, aparece em baixo do gráfico ao utilizador uma tabela que contém os diferentes *pods* que correm naquela máquina, mostrando o seu estado atual - verde se estiver a correr ou vermelho se houver algum problema-, o tipo de controlador associado ao *pod*, o seu *owner* e o ip **interno** do *pod*.

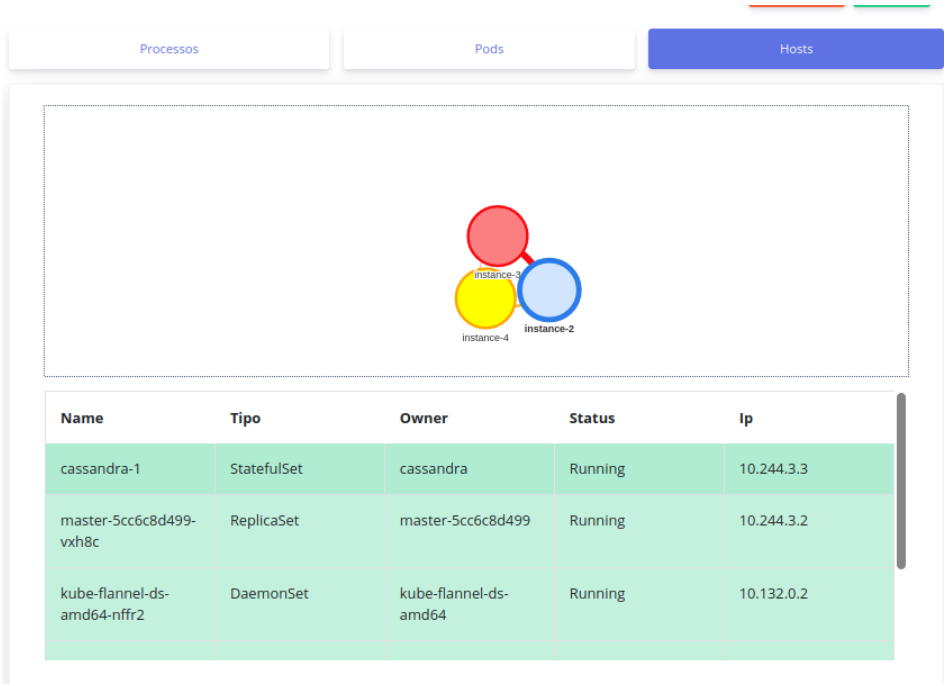


Figura 25: Vista dos hosts

Ao contrário da vista sobre os processos, estas duas vistas não possuem qualquer tipo de filtro, sendo que estas vistas são muito mais pequenas que a dos processos e não se justifica ter um filtro.

3.5 API

A API serve para fazer a ponte entre a aplicação e a orquestração de Kubernetes. Através dos diferentes *endpoints* da API podemos modificar o número de réplicas, limite e *request* de CPU e memória e começar/terminar uma monitorização ao sistema.

3.5.1 *Funcionamento geral*

A Figura 26 representa como é feita a comunicação com a API e com o resto da arquitetura da aplicação. A API serve para fazer a ponte entre a aplicação e a orquestração de Kubernetes. Através dos diferentes endpoints da API podemos modificar o número de réplicas, limite e *request* de cpu e memória e começar/terminar uma monitorização ao sistema.

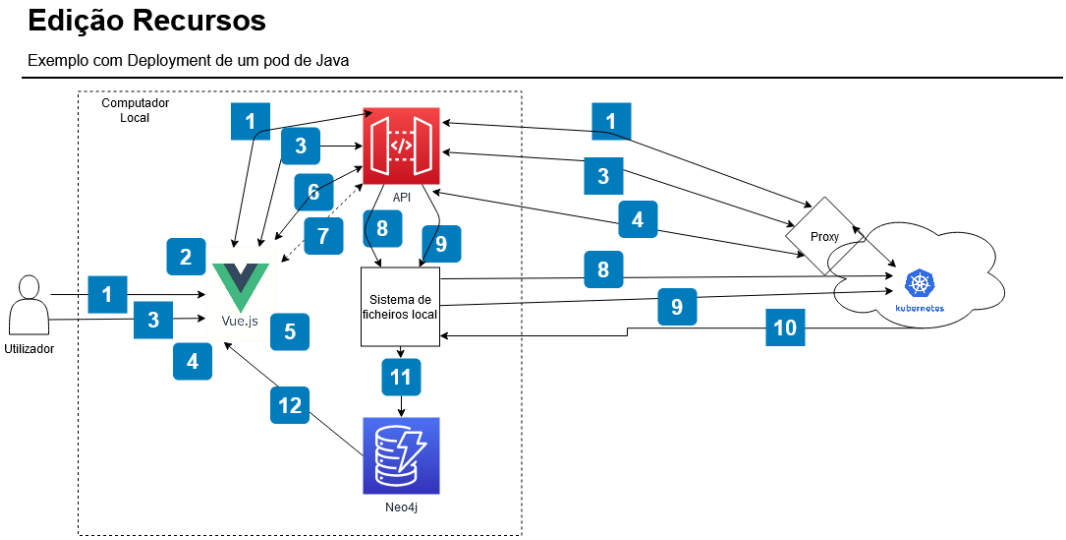


Figura 26: Exemplo da comunicação realizada

Vamos imaginar que o utilizador pretende mudar o número de réplicas de 1 para 2 de um *pod*. E antes sequer de poder começar a editar o número de réplicas tem de ser feito um *match* entre o nó que clicou na aplicação e o controlador correspondente no Kubernetes. Como descrito na secção 2.2.1, o SYSQUERY possui um atributo denominado **cont**. Olhemos para um exemplo:

$$kubepods/burstable/pod$$

$$\underbrace{54fc6959-b6cc-11ea-bd1e-42010a840007}_{\text{uid do } pod} / \underbrace{/c3ba440ee84424550e4c4193711165[...]}_{\text{Docker contentorId}}$$

A parte antes de *pod* pode variar, mas o que importa é o que está à frente deste. O número imediatamente a seguir corresponde ao uuid de um *pod* no Kubernetes, com este número podemos conseguir fazer a ligação entre o processo registado pelo SYSQUERY e a orquestração do Kubernetes. Contudo, a API quando feita a correspondência não envia as informações do *pod* mas do seu *owner*. Isto porque o *pod* é uma unidade básica do Kubernetes. Temos de encontrar o *owner* sempre do *pod*, que pode ser um *deployment*, *service*, etc e alterar a configuração dele. Ou seja, as alterações que fazemos não são só para um *pod* mas o ou vários *pods*, alterando o controlador deles. No caso de ser um RéplicaSet temos de encontrar o outro *owner* acima dele, que é uma implantação.

Assim, a solução (ver Figura 26) passa por, quando o utilizador clica num nó **[1]**, é enviado um pedido à Api para receber todos os *pods* e encontrar qual deles corresponde ao nó na rede. Depois disso extraímos o tipo de *owner* do *pod* e nome deste**[2]**.

```
    "metadata": {
                    "name": "rabbitmq-7764597b7b-bsghq",
                    "generateName": "rabbitmq-7764597b7b-",
                    "namespace": "sock-shop",
                    "selfLink": "/api/v1/namespaces/sock-shop/pods/rabbitmq-7764597b7b-bsghq",
                    "uid": "550c57e1-b6cc-11ea-bd1e-42010a840007",
                    "resourceVersion": "1155",
                    "creationTimestamp": "2020-06-25T10:12:10Z",
                    "labels": {
                        "name": "rabbitmq",
                        "pod-template-hash": "7764597b7b"
                    },
                    "ownerReferences": [
                        {
                            "apiVersion": "apps/v1",
                            "kind": "ReplicaSet",
                            "name": "rabbitmq-7764597b7b",
                            "uid": "54fc6959-b6cc-11ea-bd1e-42010a840007",
                            "controller": true,
                            "blockOwnerDeletion": true
                        }
                    ]
                },
```

Lista 3.12: Exemplo do JSON devolvido pela API do Kubernetes

Com isto podemos agora enviar um pedido à API [3] que vai procurar, neste caso, pela implantação com nome "rabbitmq-776459b7b". Depois, disto é guardada toda a informação necessária. É também verificado se no yaml existe limite e/ou request **[4]** (já que este não são obrigatórios existirem no yaml) e no caso de não existirem é atribuído o valor de 0.

```
[...]
    await this.$http.get('http://localhost:3000/pods').then(response =>
    (
      pods=response.data[0].items
   ))
    for(let i=0;i<pods.length;i++){

      if(pods[i].metadata.uid==temp){
        this.type=pods[i].metadata.ownerReferences[0].kind.toLowerCase()
        this.name= pods[i].metadata.ownerReferences[0].name
        break;
      }
   }
    if(this.type=="replicaset"){
```

```
15      var deploymentName;
16         await this.$http.get('http://localhost:3000/replicaset/'+this.name).then(response =>
17         (
18           this.name= response.data.metadata.ownerReferences[0].name
19         ))
20         this.type="deployment"
21
22     }
23     await this.$http.get('http://localhost:3000/'+this.type+'/'+this.name).then(response =>
24       {
25         this.finalName=response.data.metadata.name
26         this.finalInformation.replicas=this.verify(response.data.spec.replicas,0)
27         this.finalInformation.limitcpu=this.verify(objectPath.get( response.data, 'spec.template
               .spec.contentores.0.resources.limits.cpu' ),0)
28         this.finalInformation.limitmemory=this.verify(objectPath.get( response.data, 'spec.
               template.spec.contentores.0.resources.limits.memory' ),0)
29         this.finalInformation.requestcpu=this.verify(objectPath.get( response.data,'spec.
               template.spec.contentores.0.resources.requests.cpu'),0)
30         this.finalInformation.requestmemory=this.verify(objectPath.get( response.data,'spec.
               template.spec.contentores.0.resources.requests.memory'),0)
31       })
32   [...]
33  }
```

Lista 3.13: Código para ir buscar à API a informação do controlador de um *pod*

```
1  verify ( a, b ) { return a != undefined ? a : b }
```

Lista 3.14: Função que verifica se certo campo existe no yaml

Com a associação feita, o modal aparece ao utilizador **[5]**. Agora, quando o utilizador modifica o número de réplicas de uma para duas e grava, a aplicação um *post* para a API **[6]** para que modifique o yaml da implantação. A partir deste momento a aplicação manda um pedido 10 em 10 segundos **[7]**, para ir notificando o utilizador que quanto tempo falta para estar tudo concluído. Quando a barra chegar ao estado 4, uma mensagem aparece e a barra desaparece.

Já na API acontece muito mais durante este período. Quando a API recebe o pedido para modificar o número de réplicas, o estado é passado de 0 para 1 e imediatamente é editado e aplicado o yaml ao Kubernetes. O SYSQUERY é executado na máquina local, então é isso mesmo que é feito, localmente enviamos e iniciamos os agentes e o estado muda para 2 **[8]**. A orquestração é monitorizada durante 30 segundos e depois o estado passa para 3 e os agentes são parados **[9]**, sendo enviados os ficheiros que contém a informação sobre a orquestração para o computador local.

```
[...]
state.id="2"
state.msg="2- A come ar a monitorizacao"
exec('./startagents.sh', { cwd: './../deployment/' }, (error, stdout, stderr) => {

[...]

state.id="3"
state.msg="3-A parar monitorizacao"
exec('./stopagents.sh', { cwd: './../deployment/' }, (error, stdout, stderr) => {
[...]
```

Lista 3.15: Código para execução dos agentes e mudança de estados

Novamente, passado alguns segundos, quando os ficheiros estiverem todos criados, estes são usados para criar a nova rede no Neo4j **[11]**, o estado passa para 4, que por usa vez vai fazer com que a a rede da aplicação atualize **[12]**.

```
    exec('./loadresults.sh --clear --k8s services.txt agent.*.pickle > /dev/null 2>&1', { cwd: '
        ./../deployment/',stdio: ['pipe', 'pipe', 'ignore']}, (error, stdout, stderr) => {
    [...]
    state.id="4"
    state.msg="4-A lan ar dados para neo4j"
});
```

Lista 3.16: Código para criar nova rede de nós no Neo4j

Este é o processo executado para quase todas as ações feitas dentro da aplicação, à excepção de quando fazemos a monitorização manual (ver 3.4.2), onde depois de feito o primeiro pedido à API passa logo para o ponto **[8]**, onde são iniciados os agentes (claro que neste caso também não há *feedback* ao utilizador, de 10 em 10 segundos, já que é ele que tem de parar a monitorização, quando achar mais pertinente).

3.5.2 *Administração da Api*

Quando um utilizador tenta fazer várias mudanças num dos controladores, estas mudanças têm de ter persistência no caso de o utilizador recarregar a página. Assim, para conseguir controlar e haver persistência das mudanças que se faz a um controlador, todos os *endpoints* para as respetivas mudanças são enviados para um *endpoint* apenas. Isto é, se quisermos mudar as réplicas de um *pod* de uma para duas, mas ao mesmo tempo também queremos aumentar o limite de memória que cada *pod* associado a esse controlador pode gastar, estas

mudanças são enviadas todas ao mesmo tempo para um novo endpoint denominado *queue*. Este endpoint trata depois de chamar todos os *endpoints* presentes na *queue* e, no final, faz uma monitorização de 30 segundos.

```
1     router.post('/queue',async(req,res,next)=>{
2
3     try{
4         var a = req.body.data
5         queue.number = req.body.data.length
6         queue.currentLeft=0
7         for(let key of a){
8             await axios.post(key).then(response =>
9             {
10                console.log(response)
11            } )
12            queue.currentLeft++;
13            await delay(20000); //delay de 20 segundos entre post para endpoints para evitar
                  conflitos no Kubernetes
14        }}catch(err){
15            console.log(err)
16        }
17    executeFeedback() // funcao que contem as funcoes para monitoriza ao
18 });
```

Lista 3.17: Código da *queue* de endpoints

Desta forma, é possível haver persistência quando o utilizador atualiza a página, sem se perder qualquer mudança que se queria fazer.

Enquanto que houver atividade na *queue*, o utilizador não poderá alterar nenhum outro controlador, de forma a que se evitem conflitos e corridas aos recursos desnecessárias, que podem levar a que um ou vários *pods* entrem em *crash*. Assim, também existe um delay de 20 segundos entre cada mudança num *pod*, pois o Kubernetes leva algum tempo a ter o *pod* pronto para outra mudança, isto é, depois de mudar as réplicas, por exemplo, o *pod* precisa de alguns segundos para ficar outra vez funcional e pronto a receber mais instruções.

Por outro lado, sempre que se inicie ou recarregue a aplicação é também necessário verificar se existia alguma atividade na *queue*. Então, sempre que exista uma iniciação ou recarregamento da aplicação é enviado um pedido à API para verificar se existe alguma atividade na *queue*, e, em caso afirmativo, as mudanças aos controladores ficam bloqueadas e é requisitado o estado atual dessas mudanças - em que estado está a mudança atual e em quantos *endpoints* faltam ser solicitados.

3.5.3 *Endpoints da API*

Foram referidos vários de *endpoints* até ao momento, mas não todos. Neste sub-capítulo vão ser descritos todos os *endpoints*, de forma a sintetizá-los.

GET /pods

Obtém listagem de todos pods existentes para o namespace definido nas configurações.

GET /deployments

Obtém listagem de todos implantações existentes para o namespace definido nas configurações.

GET /statefulset

Obtém listagem de todos statefulsets existentes para o namespace definido nas configurações.

GET /geral/state

Obtém o estado atual da(s) modificação(ões).

GET /geral/queue

Obtém o número de modificações total e as que já foram efectuadas.

POST /deployments/replicas/:deployment/:id

Aplica ao deployment o número de réplicas id.

POST /deployments/resources/limits/cpu/:deployment/:id

Aplica à implantação o limite de CPU id.

POST /deployments/resources/limits/memory/:deployment/:id

Aplica à implantação o limite de memória id.

POST /deployments/resources/requests/cpu/:deployment/:id

Aplica à implantação o request de CPU id.

POST /deployments/resources/requests/memory/:deployment/:id

Aplica à implantação o request de memória id.

POST /deployments/replicas/:deployment/:id

Aplica ao statefulset o número de réplicas id.

POST /statefulset/resources/limits/cpu/:statefulset/:id

Aplica ao statefulset o limite de CPU id.

POST /statefulset/resources/limits/memory/:statefulset/:id

Aplica ao statefulset o limite de memória id.

POST /statefulset/resources/requests/cpu/:statefulset/:id

Aplica ao statefulset o request de CPU id.

POST /statefulset/resources/requests/memory/:statefulset/:id

Aplica ao statefulset o request de memória id.

POST /geral/queue

Recebe um array de endpoints, para posteriormente os
executar de forma ordenada e sem comprometer a integridade do pod.

3.6 SUMÁRIO

O SYSQUERY deixou de conter os nós denominados sockets porque traziam dificuldades de interpretação da rede ao utilizador. Os nós passaram a estar diretamente ligadas por uma aresta, que contem o somatório de todos os *Bytes* trocados entre os processos.

No Neovis passou a haver a possibilidade de haver imagens associadas aos processos, para haver uma melhor e mais rápida identificação dos mesmos; possibilidade de o utilizador organizar a visualização por instância ou por processo, tendo assim uma maior diversidade para a análise de dados; foi adicionado um evento de DoubleClick aos nós na rede, para interagir com os processos - ver os gastos de CPU e RAM, ver o número de réplicas, limites e *requests* de CPU e RAM; a forma como o atributo comunidade é definido foi modificada, para atribuir uma cor dependendo do *host* que o nó está associado; adicionada legenda para as várias instâncias, para melhor visualização da rede;

A aplicação desenvolvida possui várias funções:

- Possibilidade de alterar vários recursos de um *pod* - réplicas, limite e *request* de CPU e RAM;
- Monitorização personalizada - o utilizador pode monitorizar por alguns segundos ou por vários minutos;
- Mostrar a informação recolhida em várias vistas - por processos, *pods*, *hosts*.

Para se conseguir alterar os recursos de um *pod* ou fazer outro tipo de comunicação com o Kubernetes, a aplicação comunica com uma API desenvolvida, em NodeJS, que por sua vez vai comunicar com a API do Kubernetes que vai transmitir as mudanças para o Kubernetes as executar. É possível lançar várias mudanças ao mesmo tempo sem comprometer a integridade da orquestração, pois existe uma *queue* que controla o acesso aos endpoints e pela ordem correta - todos os endpoints estão indicados na secção 3.5.3.

4

CASOS DE ESTUDO / RESULTADOS

Neste capítulo são discutidas as respostas, por parte do CloudOverWatch e do Weave Scope, a duas orquestrações de Kubernetes diferentes - uma mais simples para perceber como a informação é demonstrada aos utilizadores e outra mais complexa para perceber o impacto da complexidade na demonstração da informação. Será comparada a informação recolhida e como esta é apresentada ao utilizador. Também será avaliado se a informação recolhida pelas aplicações é a correta e o impacto da informação quantitativa recolhida pelo SYSQUERY, relativamente ao tráfego entre os diferentes processos, nos ajuda a perceber mais rapidamente se existe algum problema na nossa orquestração. O guia para setup de ambientes encontra-se disponível no apêndice C.

Haverá dois componentes para a avaliação/comparação entre as duas aplicações: uma avaliação booleana - se a aplicação em questão apresenta o requisito ou não - e, no caso de ambas as aplicações cumprirem o requisito e seja necessário perceber quem o faz requisito mais rapidamente, será feita uma avaliação *Keystroke Level Model* (KLM) - prevê quanto tempo demora a realizar uma tarefa. Também será comparada as diferentes vistas que as aplicações apresentam - por processos, *pods* ou *hosts* - ao nível da informação apresentada e da sua qualidade.

Além da comparação entre as ferramentas, também será interessante comparar a informação recolhida por uma arquitetura de monitorização orientada aos recursos. Para tal, será usada uma combinação entre o Prometheus - para recolha das métricas - e do Grafana - para a apresentação da informação.

Na análise dos resultados tem de se ter em conta que o SYSQUERY não recolhe informações sobre a **instância-1** - instância que contém os *pods* de controlo do Kubernetes - pois a informação trocada entre os variados *pods* e processos não é muito relevante para análise das orquestrações.

4.1 CONFIGURAÇÃO DOS AMBIENTES

Os ambientes são: um conjunto de base de dados Cassandra e o SockShop [1] - um *benchmark* de micro serviços. Primeiramente, vamos olhar para o conjunto de base de dados Cassandra com o objetivo de perceber melhor como as duas ferramentas mostram a informação, dado ser uma implementação mais simples. Depois, vamos ver como ambas se comportam num ambiente mais complexo, usando o *benchmark* Sock Shop, onde o foco será perceber que impacto tem a maior quantidade de dados na visualização dos dados por parte dos utilizadores, em cada umas das aplicações.

4.1.1 *Ambiente*

O ambiente de testes é a Google Cloud Platform, onde é usado um número considerável de máquinas para cada um dos testes, mas todas elas com a mesma configuração:

- Imagem do disco: xenial-k8s-sysquery - específica para correr o SYSQUERY
- Tipo de máquina: n1-standard-4
- vCPUS: 4, 15 GB de memória
- Disco: 375GB SSD

4.1.2 *Orquestração de Cassandras*

O objetivo nesta orquestração é ter em cada uma das instâncias um servidor Cassandra e um *worker* de Spark. A orquestração começa com a implantação de 4 máquinas e da iniciação do Kubernetes no ambiente da nuvem, posteriormente é feito a implantação de um *pod cassandra master* e depois são lançados outros três *pods* cada um com dois contentores. O yaml usado para esta implantação encontra-se no GitHub[2].

4.1.3 *Sock Shop*

O *benchmark* Sock-Shop simula um website e-commerce de venda de meias. Tem como objetivo principal a demonstração e teste de tecnologias nativas de microsserviço e nuvem, sendo assim ideal para testar as duas ferramentas.

A orquestração começa com o implantação de 4 máquinas e da iniciação do Kubernetes no ambiente da nuvem, posteriormente é criado um *namespace* denominado *sock-shop* e, por

1 Google SockShop - https://microservices-demo.github.io/
2 https://github.com/danielsilva2017/thesisDocumentation/blob/main/cassandra.yaml

fim, é iniciado toda a estrutura: 13 *pods* relacionados com a estrutura e suporte de uma loja - um *pod* de frontend, um *pod* da base de dados e depois são lançados outros dois *pods* que simulam a compra e venda dos items na loja, tendo no total 15 *pods* distribuídos ao longo de 3 das máquinas enquanto que a primeira máquina contêm os *pods* relacionados com o suporte ao Kubernetes. O *yaml* correspondente aos *pods* de suporte da loja e o yaml correspondente às simulações encontram-se no GitHub[3].

4.2 RESULTADOS

4.2.1 *Orquestração de Cassandras*

O objetivo para esta orquestração é perceber como ambas as aplicações reagem num ambiente pequeno, perceber que informação demonstram e como esta informação é apresentada - qualidade e nitidez da informação. A avaliação será sobre como a informação é apresentada, a dificuldade para obter a avaliação e o que as duas aplicações podem oferecer de diferente.

Pods

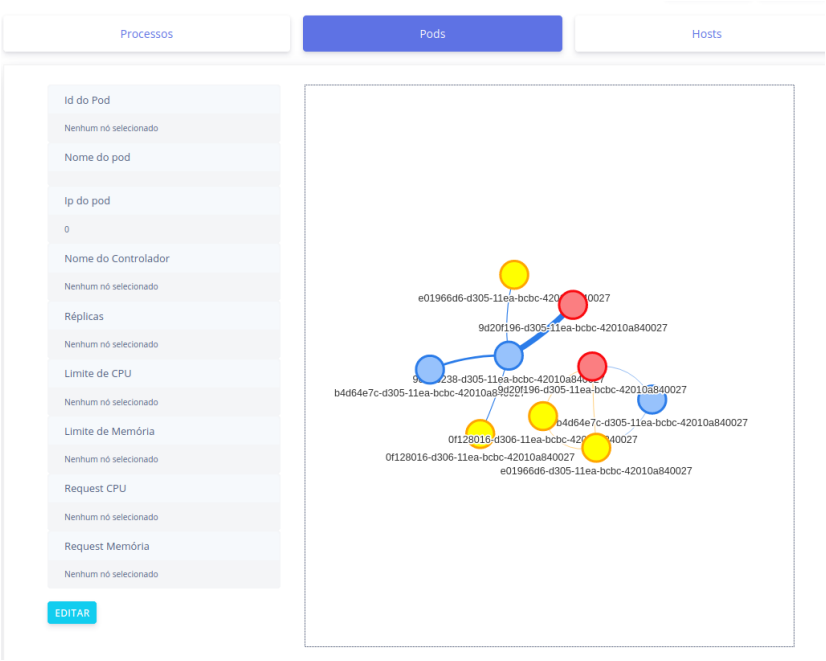


Figura 27: Vista geral sobre os *pods* - CloudOverWatch

3 https://github.com/danielsilva2017/thesisDocumentation/blob/main/sockshop.yaml
https://github.com/danielsilva2017/thesisDocumentation/blob/main/sockshop_loadtest.yaml

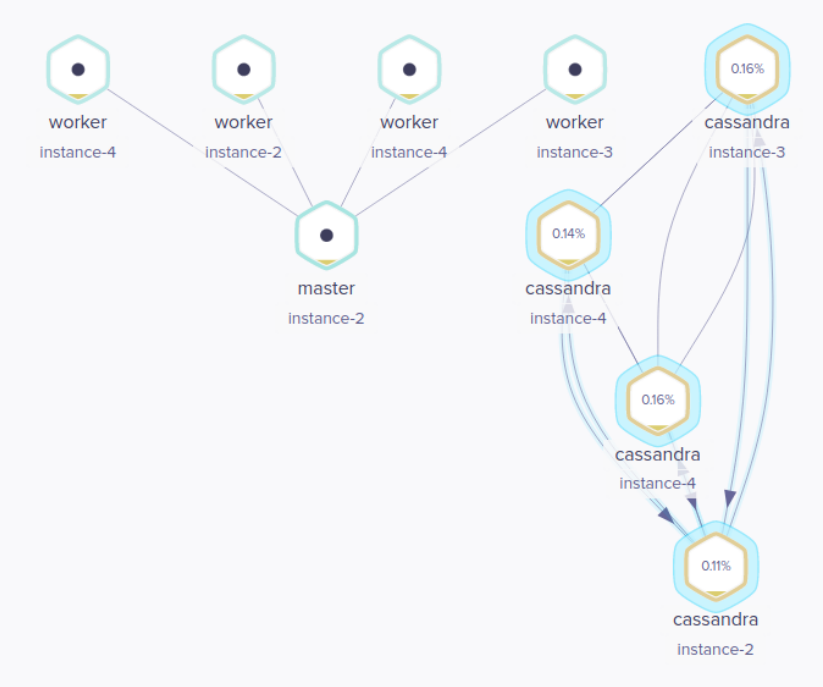


Figura 28: Vista geral sobre os *pods* - Weave Scope

Nas Figuras 28 e 27 é perceptível que ambas as aplicações mostram a mesma informação e em ambas é fácil perceber o que está a acontecer e que ligações existem entre os *pods*. Uma dificuldade acrescida por parte do CloudOverWatch, é que o SYSQUERY não consegue obter o nome do *pod* ou controlador, apenas consegue recolher o seu *id*. Sendo assim é necessário clicar no *pod* e esperar que a API devolva a informação completa, para saber o nome do *pod*.

Processos

Na Figura 30, é fácil reconhecer que é muito difícil retirar alguma informação útil na vista geral dos processos no Weave Scope. A informação apresentada é muito grande, o que torna impossível extrair qualquer indicação sobre o que está a acontecer na orquestração. Por outro lado, no CloudOverWatch, presente na Figura 29, conseguimos retirar alguma informação importante sem fazer qualquer outra ação além de escolher a vista por processos, devido a esta informação estar dispersa na rede de forma mais compacta.

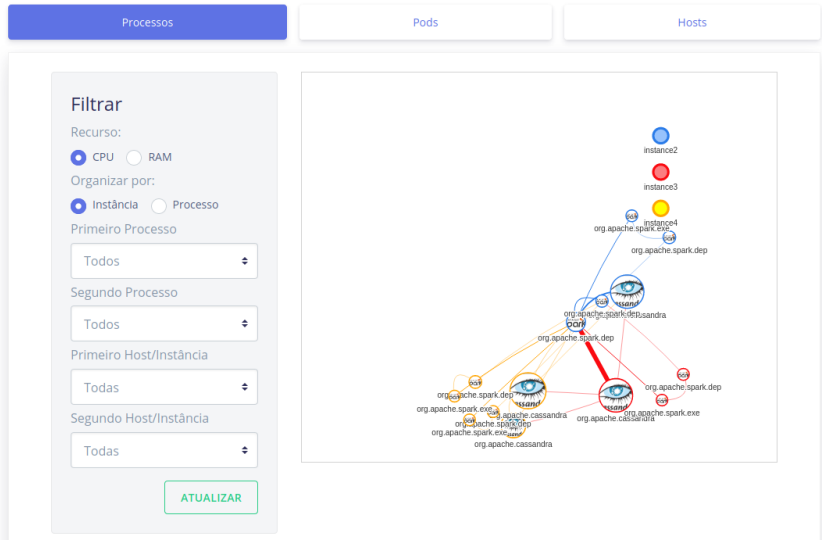
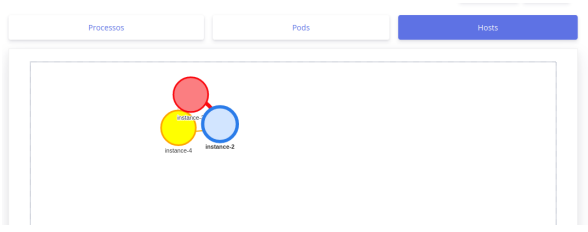


Figura 29: Vista geral sobre os processos - CloudOverWatch

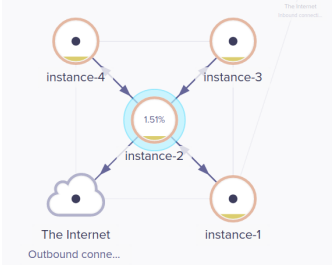
Figura 30: Vista geral sobre os processos - Weave Scope

Hosts

Pela observação das Figuras 31 e 32, à parte da informação recolhida da instância-1 por parte do Weave Scope, não existe qualquer diferença sobre a informação apresentada por parte das duas aplicações.

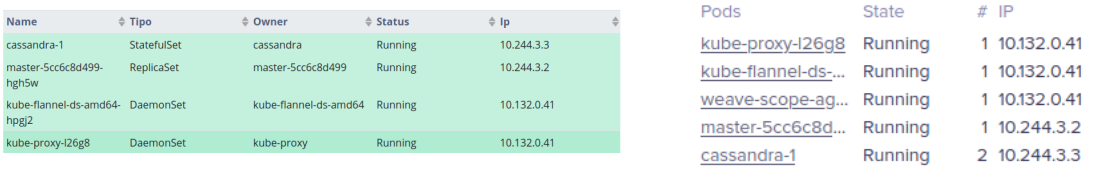


(a) CloudOverWatch



(b) Weave Scope

Figura 31: Vista geral sobre os *hosts*



Name	Tipo	Owner	Status	Ip
cassandra-1	StatefulSet	cassandra	Running	10.244.3.3
master-5cc6c8d499-hgh5w	ReplicaSet	master-5cc6c8d499	Running	10.244.3.2
kube-flannel-ds-amd64-hpgj2	DaemonSet	kube-flannel-ds-amd64	Running	10.132.0.41
kube-proxy-l26g8	DaemonSet	kube-proxy	Running	10.132.0.41

(a) CloudOverWatch

Pods	State	#	IP
kube-proxy-l26g8	Running	1	10.132.0.41
kube-flannel-ds-...	Running	1	10.132.0.41
weave-scope-ag...	Running	1	10.132.0.41
master-5cc6c8d...	Running	1	10.244.3.2
cassandra-1	Running	2	10.244.3.3

(b) Weave Scope

Figura 32: Informação detalhada dos *pods* a correr num *host*

4.2.2 *Sock Shop*

O objetivo com esta orquestração será agora perceber melhor como a aplicação reage a um aumento substancial de *pods* e que impacto isto tem sobre como a informação é apresentada, considerando não só a qualidade como também a nitidez da informação. Obviamente também é importante assegurar que a informação apresentada é a correta. Os *pods* que devemos encontrar serão os seguintes:

Namespace	Nome do Pod	Ip	Instância
loadtest	load-test-59d898fb7f-2fzmf	10.244.1.8	instance-4
loadtest	load-test-59d898fb7f-hsrqv	10.244.2.6	instance-3
sock-shop	carts-66bc68f95f-8jsxb	10.244.1.7	instance-4
sock-shop	carts-db-7d79c89fdb-hxxwk	10.244.3.3	instance-2
sock-shop	catalogue-6b5f68f7b6-4wvkd	10.244.3.7	instance-2
sock-shop	catalogue-db-55965799b9-mzj2j	10.244.2.3	instance-3
sock-shop	front-end-6694dcd58f-wkt6j	10.244.1.3	instance-4
sock-shop	orders-65bb8fcf9b-f624d	10.244.3.4	instance-2
sock-shop	orders-db-6cdb57dc6d-jb58l	10.244.2.4	instance-3
sock-shop	payment-6465b75bb6-hh2ps	10.244.1.4	instance-4
sock-shop	queue-master-6b5b5c7658-sdf8m	10.244.2.5	instance-3
sock-shop	rabbitmq-7764597b7b-85wkk	10.244.3.5	instance-2
sock-shop	shipping-7f86cf76f8-7nxvb	10.244.1.5	instance-4
sock-shop	user-8c7b575cc-whzf8	10.244.1.6	instance-4
sock-shop	user-db-86dcc8cdb5-2njpj	10.244.3.6	instance-2

Tabela 1: Tabela com os diferentes *pods* e *namespaces*

A primeira tarefa é comparar os valores recolhidos por ambas as aplicações e perceber se os valores recolhidos são corretos, que neste caso ambas recolheram corretamente - ver apêndice A.

Pods

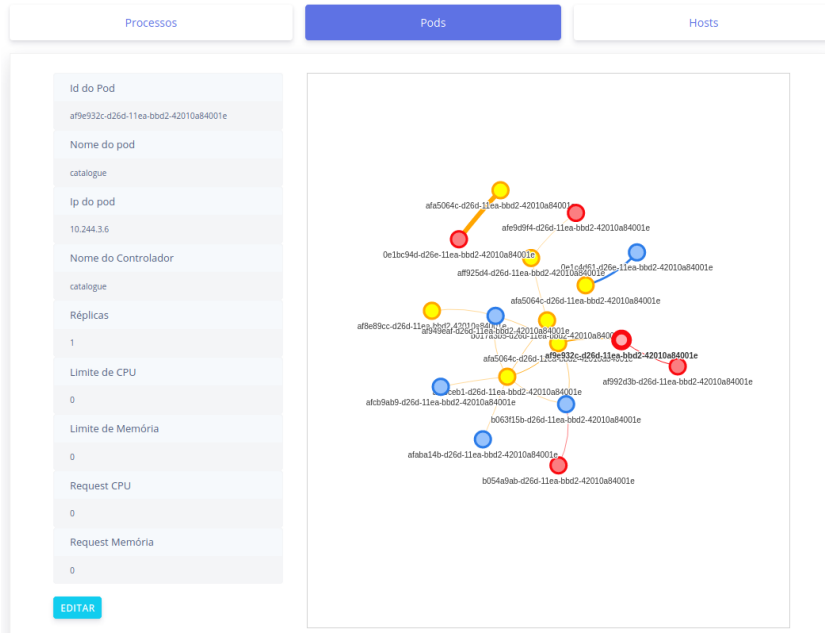


Figura 33: Vista geral sobre os *pods* - CloudOverWatch

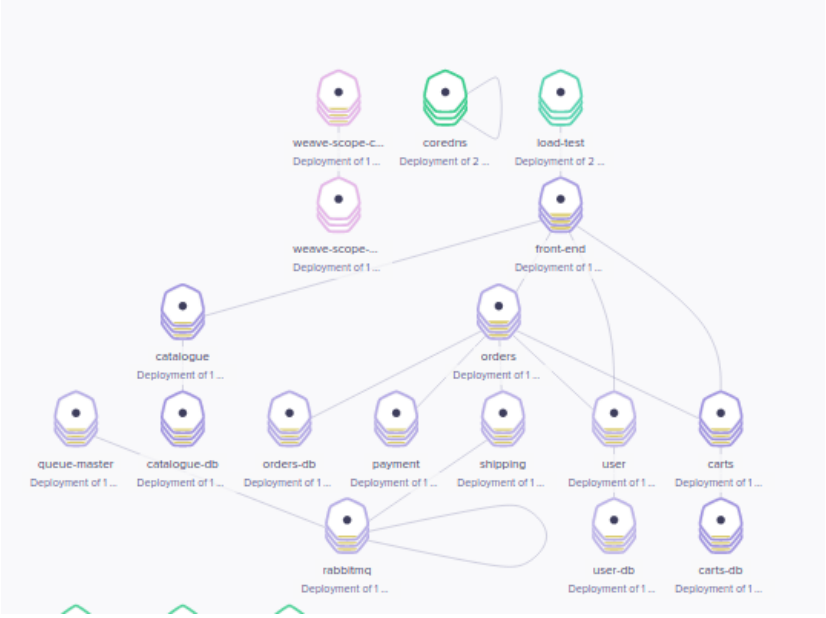


Figura 34: Vista geral sobre os *pods* - Weavescope

Ao nível de *pods* é possível observar, nas Figuras 33 e 34, que o Weave Scope continua a comportar-se melhor, sendo que a sua vista mais organizada continua a trabalhar a seu favor (caso que não acontece como já vimos e poderemos ver outra vez, nos processos). A nível de informação geral ambas as soluções apresentam a mesma informação optando o CloudOverWatch por mostrar os *pods* todos enquanto que o Weave Scope opta por mostrar os controladores - pode-se verificar no load-test que o Weave Scope diz que existem duas

réplicas enquanto que o CloudOverWatch mostra ambas as réplicas, sendo que no final a informação coexistente nos dois ambientes é a mesma, modificando apenas como é demonstrada, ficando ao utilizador final decidir qual prefere.

Processos

Ao nível dos processos, como é possível observar nas Figuras 35 e 34, é onde continua a haver uma grande diferença. É notório que o Weave Scope é ainda mais confuso e a vista geral tornou-se demasiado grande para ser possível tirar alguma conclusão, tornando o uso do zoom indispensável - aumentado assim o tempo de o utilizador utiliza para realizar determinada tarefa.

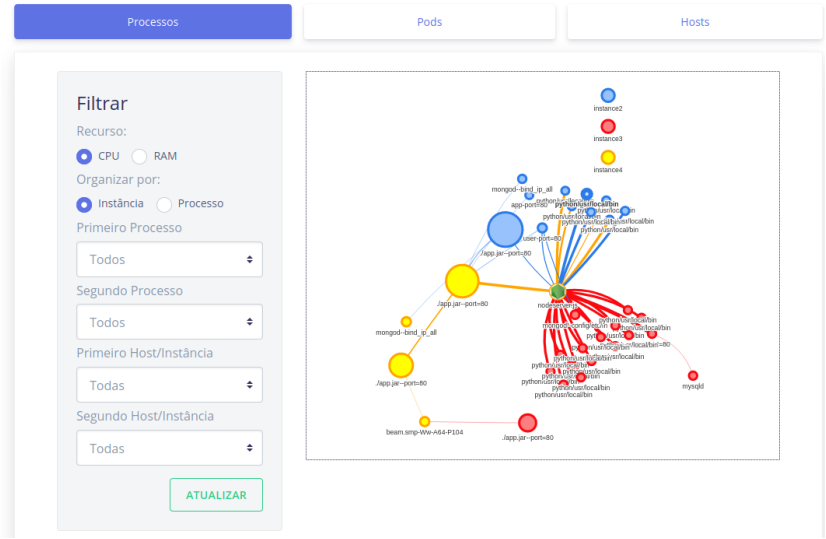


Figura 35: Vista geral sobre os processos - CloudOverWatch

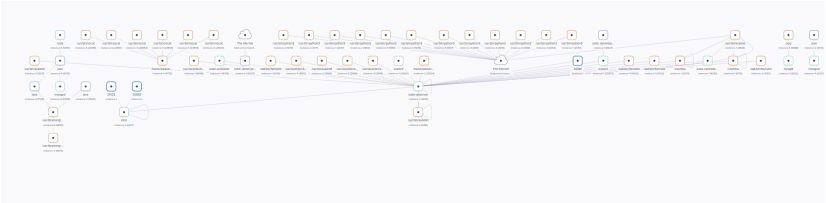
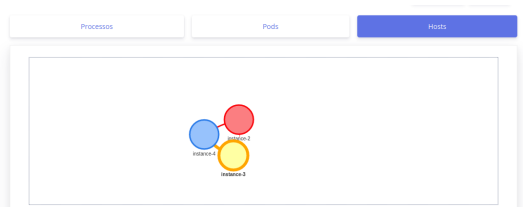


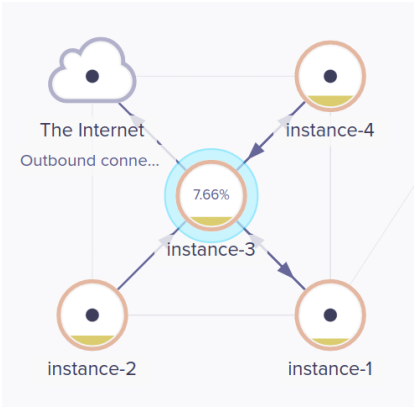
Figura 36: Vista geral sobre os processos - Weave Scope

Hosts

Como é possível observar, nas Figuras 38 e 39, que ambas as ferramentas apresentam, novamente, informações idênticas para o caso dos *hosts*. Em comparação com os resultados

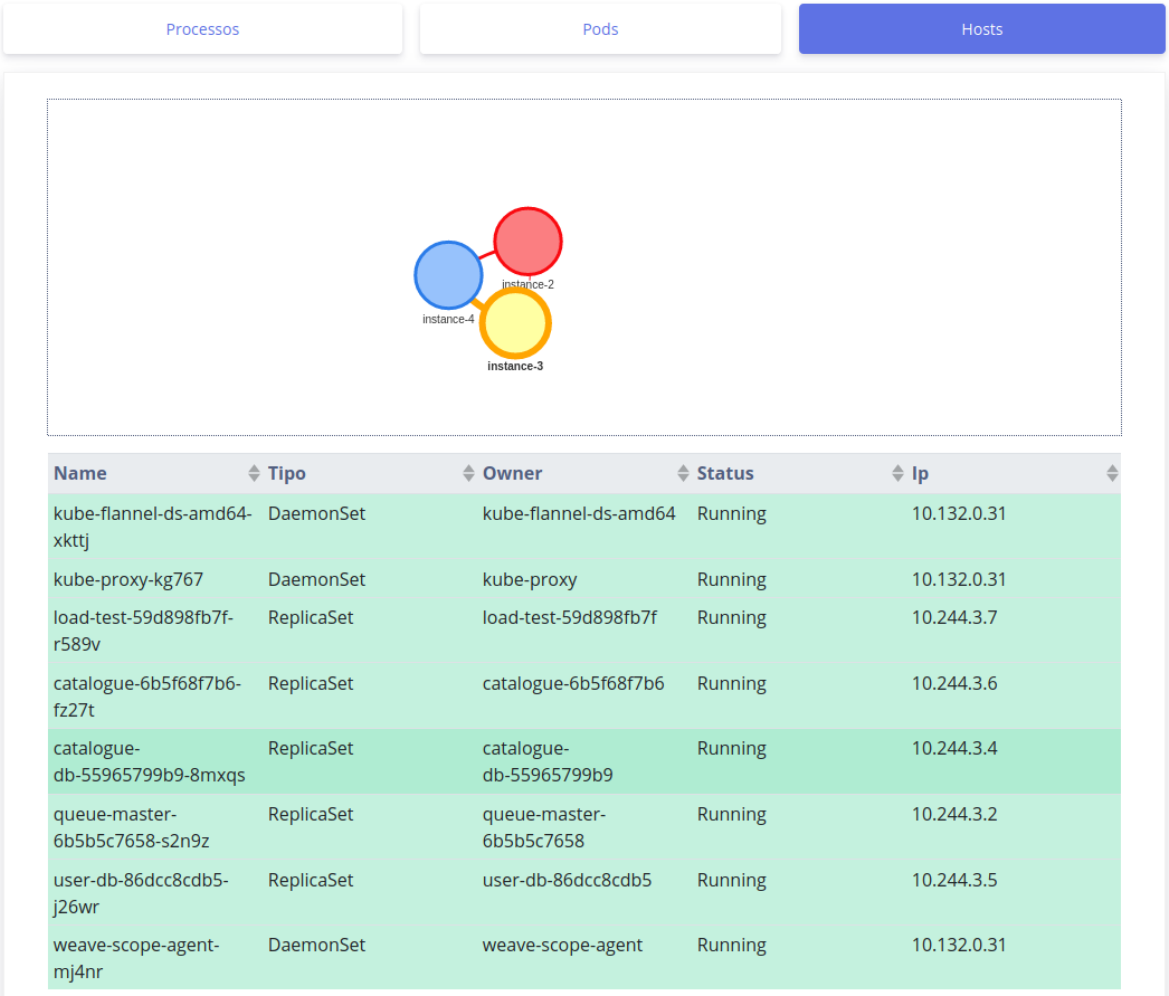


(a) CloudOverWatch



(b) Weave Scope

Figura 37: Vista geral sobre os *pods*



Name	Tipo	Owner	Status	Ip
kube-flannel-ds-amd64-xkttj	DaemonSet	kube-flannel-ds-amd64	Running	10.132.0.31
kube-proxy-kg767	DaemonSet	kube-proxy	Running	10.132.0.31
load-test-59d898fb7f-r589v	ReplicaSet	load-test-59d898fb7f	Running	10.244.3.7
catalogue-6b5f68f7b6-fz27t	ReplicaSet	catalogue-6b5f68f7b6	Running	10.244.3.6
catalogue-db-55965799b9-8mxqs	ReplicaSet	catalogue-db-55965799b9	Running	10.244.3.4
queue-master-6b5b5c7658-s2n9z	ReplicaSet	queue-master-6b5b5c7658	Running	10.244.3.2
user-db-86dcc8cdb5-j26wr	ReplicaSet	user-db-86dcc8cdb5	Running	10.244.3.5
weave-scope-agent-mj4nr	DaemonSet	weave-scope-agent	Running	10.132.0.31

Figura 38: Informação detalhada - CloudOverWatch

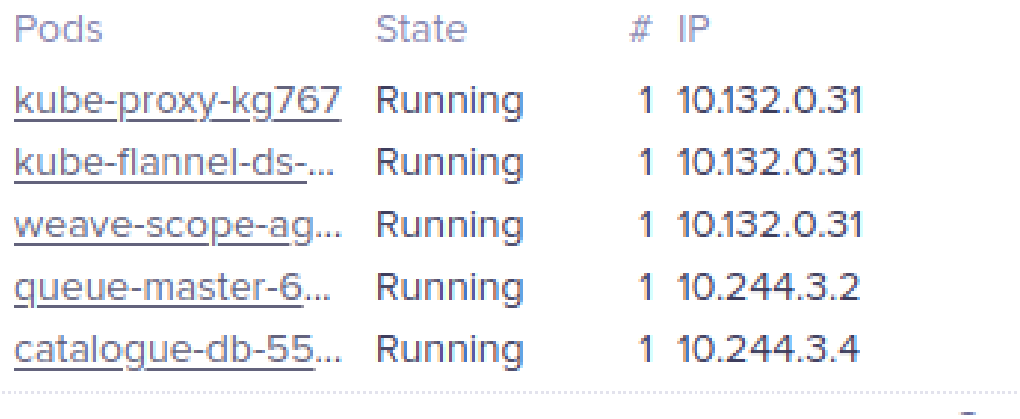


Figura 39: Informação detalhada - Weave Scope

apresentados no caso da orquestração mais simples, ao nível dos *hosts*, não há grande diferença em ambas as aplicações.

Teste adicional

O *benchmark* Sock-Shop consegue simular vários clientes na sua loja, que criam novos processos, em que o número de processos criados vai depender das instruções fornecidas. O teste consiste em aumentar o número de clientes para mais processos serem criados entre testes e tem como principais objetivos verificar como estas alterações são apresentados ao utilizador e qual a melhor aplicação neste caso específico. Esta simulação cria vários processos *python* que se ligam ao front-end (processo nodejs). Vai se começar com três clientes e, numa segunda fase, vai se aumentar para quatro clientes.

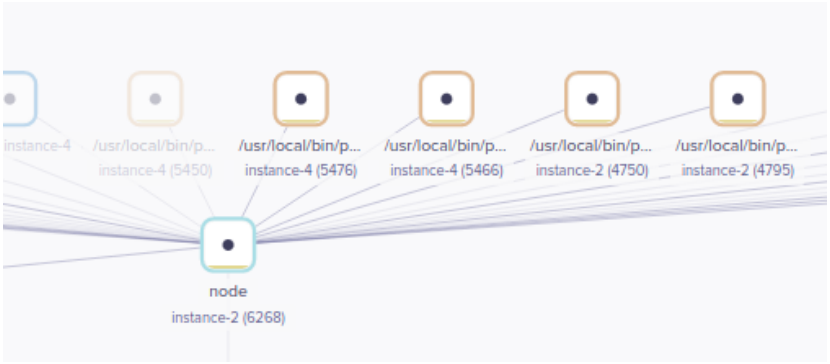


Figura 40: Weave Scope gráfico com 3 clientes

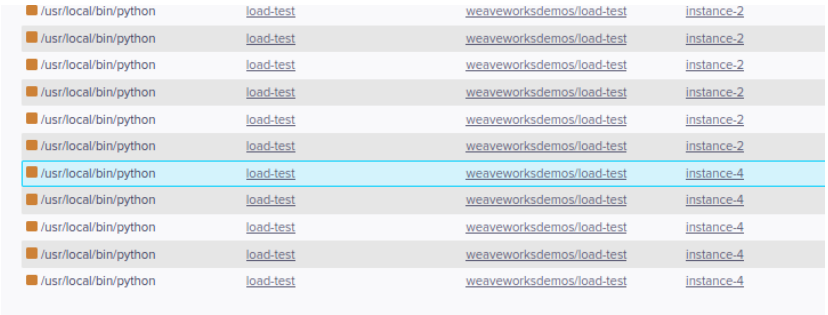


Figura 41: Weave Scope Tabela com 3 clientes

Na Figura 40, que foi tirada com o máximo de zoom possível, de forma a se conseguir observar com alguma nitidez e obter alguma informação, faz com que pareça que só existem cinco processos *python* ligados ao processo *node*. Mas como podemos ver na Figura 41 existem mais. No WeaveScope só podemos tirar a conclusão de que os processos são do mesmo tipo e que existem uma ligação ativa entre cada um deles com o processo *node*.

Ao contrário do WeaveScope, na Figura 42, é fácil de perceber que existem vários processos ligados ao *front-end*, inclusive conseguimos contá-los. É também nesta situação que o CloudOverWatch nos dá mais alguma informação do que conseguiríamos obter com outras soluções. Nas Figuras 42 e 43, podemos observar que o número de bytes é muito parecido - todas as ligações entre os processos *python* com o processo *node* andam à volta de 230 megabytes (MB) -, o que permite extrapolar que as operações a serem feitas além de serem do mesmo tipo, já que todas criam um processo *python*, poderão ser exatamente iguais. No panorama geral esta informação ajuda-nos a perceber que os utilizadores estão a realizar a mesma operação e, consequentemente, permite-nos entender o problema e atuar mais facilmente.

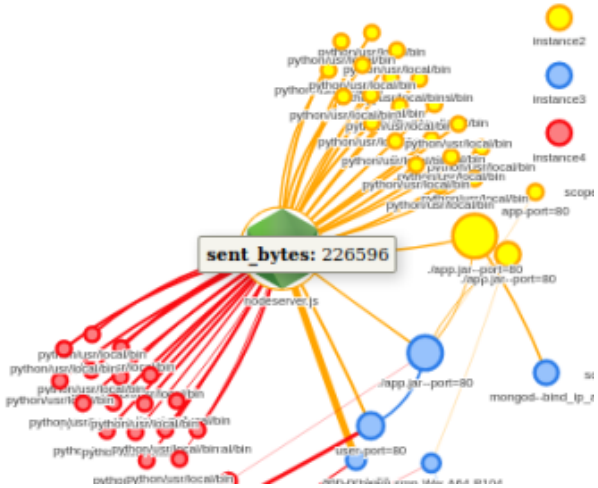


Figura 42: CloudOverWatch gráfico com 3 clientes

Figura 43: Outra ligação no CloudOverWatch

Agora, numa segunda fase, aumentou-se o número de clientes para 4 e, como podemos verificar na Figura 44, que é retirada do CloudOverWatch, é imediatamente perceptível que o número de clientes aumentou enquanto que no WeaveScope, devido ao facto de termos de fazer zoom para se retirar alguma informação, não se consegue retirar nenhuma conclusão. A forma mais fácil de perceber que algo está a acontecer na orquestração ao usar o WeaveScope é indo à tabela e perceber que se existe algum aumento significativo em algum dos *pods*.

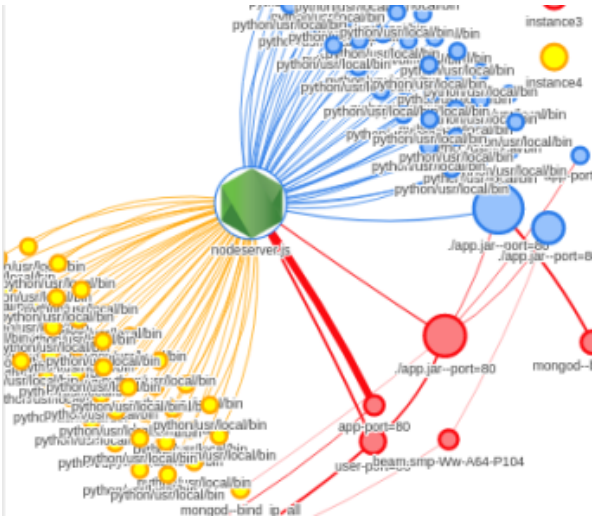


Figura 44: CloudOverWatch gráfico com 3 clientes

Comparação com primeira orquestração

O Weave Scope continua bastante pior ao nível dos Processos, a vista que oferecem fica demasiado comprida e, inicialmente, sem fazer qualquer tipo de zoom, não podemos tirar nenhuma conclusão acerca do estado da orquestração. Podendo assim ficar demonstrado que quando mais complexa for a orquestração maior será o tempo para fazer determinada tarefa no Weave Scope.

Por outro lado, no CloudOverWatch, a opção por uma vista em que os nós são dispostos por instâncias e usamos a forca de atracão entre nós, como mecanismos de disposição dos nos na rede, começa a ter impacto quando a complexidade da orquestração aumenta, já

que a dificuldade de obter informação nesta vista não piora. O facto de se perder alguma qualidade quando as orquestrações são mais pequenas, quando comparado com o Weave Scope, esta forma de disposição é mais agradável de visualizar quando a complexidade das orquestrações aumenta.

Ao nível de *pods* e *hosts* não houve nenhuma mudança significativa, podendo apenas extrapolar que o que aconteceu nos processos - apresentação da informação deteriorada por parte do Weave Scope - poderá acontecer ao nível dos *pods*, já que a disposição se baseia também em hierarquia.

Ficou também mais claro a grande vantagem de se saber a quantidade de dados trocados entre processos e a vantagem que trás a organização por instâncias na visualização ao nível dos processos.

4.2.3 *Avaliação a nível de requisitos*

A avaliação KLM segue os valores descritos no apêndice B. Para esta avaliação não foram usados os valores M (Operador Mental) visto que não traziam diferença aos valores comparativos finais, trazem apenas maior confusão na hora de analisar, a avaliação feita.

Tarefas	Operadores Weave Scope	Operadores CloudOverWatch
Descobrir PId e PPID do Controlador de um Processo	*Apontar para Processo(P)* *Click no Processo(B)* *Esperar load da págona(W1)* *Apontar para PID E PPID do Controlador(P)* *Final: PBW1P Tempo: 3.1s*	*Apontar para Pod(P)* *DoubleClick(BB)* *Esperar load da página(W)* *Apontar para PID e PPID do Controlador(P)* *Final: PBBWP Tempo: 3.4s*
Descobrir a que outros Processos um Processo está ligado	*Apontar para Processo(P)* *Click no Processo(B)* *Esperar load da págona(W1)* *Final: PBW1 Tempo: 2s*	*Apontar para Processo(P)* *DoubleClick(BB)* *Esperar load da página(W3)* *Final:PBBW3 Tempo: 2.3s*
Ver todos os Processos numa tabela	Apontar para botao "Table"(P) Click no botao (B) Esperar load da tabela (W2) Final: PBW2 Tempo: 1.8s	*Apontar para botao "Ver mais"(P)* *Click no botao (B)* *Esperar load da tabela (W4)* *Final: PBW4 Tempo: 1.40s*
Ver mais detalhes de um Pod	Apontar para Pod(P) Click no Pod(B) Esperar load da página(W1) Final: PBW1 Tempo: 2s	*Apontar para Pod(P)* *Click no Pod(B)* *Esperar load da página(W3)* *Final: PBW3 Tempo: 2.3s*
	Tempo final: 8.9s	Tempo final: 9.4s

Tabela 2: Tabela da avaliação KLM

No plano geral as ações no Weave Scope levam menor tempo a executar, mas tendo como maior factor a rapidez de carregamento das páginas e não o número de passos, já que o número de passos necessários é igual para todos as tarefas. Outros requisitos de comparação encontram-se na seguinte tabela.

Requisito	CloudOverWatch	Weave Scope
Descobrir quantidade de ligações entre dois pods/processos/hosts	✓	✓
Descobrir quantidade de dados trocados entre dois pods/processos/hosts	✓	
Possibilidade de editar o(s) controlador(es)	✓ (Editar diretamente certos atributos)	✓ (Editar os yamls)
Maior quantidade de informação		✓

Tabela 3: Tabela de requisitos

4.3 PROMETHEUS E GRAFANA

Ambas as aplicações analisadas anteriormente são orientadas à estrutura (ver a secção 2.2.2) e, além de o foco ser perceber como os variados *pods* estão distribuídos e conectados entre si,também permitem interagir com o Kubernetes diretamente. Já o uso do Prometheus com o Grafana - monitorização orientada aos recursos (na secção 2.2.1) - tem como foco dar ao utilizador o maior número de métricas possíveis sem se preocupar com interação com o Kubernetes, funcionam apenas como um depósito de informação.

Foi escolhido o benchmark do Sock-shop para este teste.

4.3.1 *Setup do ambiente*

Primeiramente é preciso instalar o Prometheus. Uma instância do mesmo foi adiciona a uma das VMs, a correr na porta *9090*, seguindo [5]. Depois inicia-se uma instância Grafana 7.1.5 na porta 3000 [6].

```
1 sudo apt-get install -y adduser libfontconfig1
2 wget https://dl.grafana.com/oss/release/grafana_7.1.5_amd64.deb
3 sudo dpkg -i grafana_7.1.5_amd64.deb
4 sudo systemctl daemon-reload
5 sudo systemctl start grafana-server
6 sudo systemctl status grafana-server
```

Lista 4.1: Script para correr o Grafana

Para ambas ferramentas é feita uma *port-forwarding*, para se conseguir ligar pelo browser na máquina local a cada uma das respectivas ferramentas.

4.3.2 *Resultados*

O *Grafana* deteta todos os processos e *pods* como ambas as ferramentas anteriormente apresentadas. Para os diferentes *pods* vai recolhendo diferentes métricas, usando o Prometheus e, vai mostrando os dados ao longo do tempo.

Na Figura 45, podemos confirmar que o Grafana deteta corretamente os vários *pods* como as outras ferramentas e também é possível verificar o uso de CPU por parte de cada um dos *pods* ao longo do tempo, e na Figura 46 podemos ver o consumo de CPU para um determinado controlador.

Figura 45: Vista no Grafana do CPU gasto por controlador

Figura 46: Vista no Grafana do CPU gasto de um controlador

Além do CPU também é interessante ver o gasto de memória ao longo do tempo, que se encontra na Figura 47. É possível também descobrir qual das instâncias gastou mais memória, neste caso a **instância-1**, e a nível de controladores foi o *queue-master*.

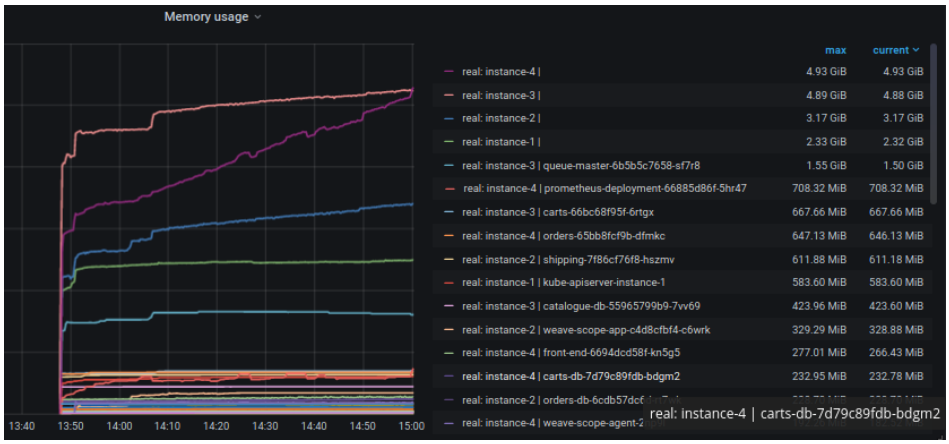


Figura 47: Vista no Grafana da Memória gasta por controlador

O Grafana também disponibiliza a quantidade de tráfego de rede de cada controlador, mas não é possível perceber quem troca dados com quem. Esta informação está presentada na Figura 48.

Figura 48: Tráfego de rede - Grafana

4.3.3 *Comparação*

Como esperado a nível de métricas ao longo tempo esta abordagem tem melhores resultados e consegue-se tirar melhores conclusões relativamente a variadas métricas, principalmente porque, o Weave Scope atualiza o uso de CPU e memória, mas não guarda histórico e o CloudOverWatch dá o uso de CPU e memória ao longo de certo tempo - durante o tempo que os agentes de monitorização estão ativos - e não vai atualizando o valor.

Apesar de não de mostrar as ligações entre os diferentes controladores, o Grafana mostra a quantidade de tráfego de cada processo. No entanto, não é possível perceber como esse tráfego está a ser feito - entre que processos ou quantas conexões são feitas, o que faz com que se considere esta informação incompleta.

4.4 SUMÁRIO

Os resultados apresentados mostram que ambas as aplicações mostram informação bastante similar, cumprindo os requisitos para uma boa visualização de uma orquestração na nuvem. Tendencialmente, a informação no Weave Scope requer menos cliques e mostra alguma informação adicional. De notar, que o foco da aplicação desenvolvida foi mostrar a quantidade de informação trocada entre processos, *pods* e *hosts*, dados que o Weave Scope não apresenta. O Weave Scope apenas mostra a quantidade de ligações feitas entre processo,*pods* e *hosts*, o que não representa muita informação, dado que, por exemplo, 100 ligações entre dois processos não significa que exista ali um *bottleneck*, por outro lado uma só ligação pode criar um *bottleneck*, se esta trocar muita quantidade de dados, por isso é muito importante quantificar a quantidade de dados trocados e não só o número de ligações.

No primeiro caso de estudo o Weave Scope comportou-se melhor que o CloudOverWatch porém quando se aumentou a complexidade da orquestração, a resposta do CloudOverWatch manteve-se enquanto que o WeaveScope ficou mais confuso e mais difícil de perceber o que estava a acontecer na orquestração.

Por último, verificou-se a utilidade de ferramentas como o Grafana, quando se quer uma visualização ao longo do tempo, das diferentes métricas recolhidas sobre a orquestração mas que a nível de estrutura não se consegue retirar nenhuma conclusão, ficando assim a monitorização incompleta.

5

CONCLUSÃO

O estudo da arte revelou uma lacuna ao nível da monitorização do Kubernetes. O sistema de monitorização que está integrado no Kubernetes é obsoleto. Ao longo do segundo capítulo, falou-se de duas abordagens diferentes para resolver este problema : uma orientada aos recursos e uma orientada à estrutura. Relativamente à primeira, são expostas as grandes vantagens relativamente ao sistema de monitorização interna do Kubernetes mas também dito que já não é suficiente dado ser cada vez mais importante analisar a estrutura e a troca de informação entre os diferentes componentes que a compõem. É, então, introduzida a alternativa, uma monitorização orientada à estrutura, dando como exemplo o WeaveScope, que além de ainda ter preocupação em mostrar as métricas clássicas de monitorização, também mostra como os diferentes componentes estão interligados e quantas vezes trocam informação entre os outros, não mostrando a quantidade de dados trocados.

Com esta informação em mente, foi proposto como dissertação o desenvolvimento de uma aplicação orientada à estrutura com o acrescido de poder também conseguir visualizar a quantidade de tráfego trocado entre *pods*, utilizando o SYSQUERY, e que pudesse interagir diretamente com o Kubernetes.

Assim, o desenvolvimento da aplicação CloudOverWatch foi desde a familiarização de todas as diferentes ferramentas até à implementação de complexas interações entre os múltiplos componentes, em que o foco principal da aplicação não foi a optimização, mas sim comprovar que inúmeros objetivos diferentes eram possíveis de implementar.

Ao longo do terceiro capítulo, podemos evidenciar esses mesmos objetivos a serem todos atingidos, com destaque para a possibilidade de alteração vários recursos de controladores de *pods*;possibilidade de visualização da informação em grupos diferentes - processos, *pods*, *hosts*; construção e ordenação - por processos ou instâncias - controlada da rede. São apresentadas as diferentes mudanças a já existentes ferramentas ou bibliotecas, é discutida a implementação do lado do cliente e como este comunica com o Kubernetes para efectuar mudanças na orquestração.

Posteriormente, foram feitos vários testes para comparar a aplicação desenvolvida com o WeaveScope e também com uma abordagem orientada aos recursos. Como espectável, percebeu-se que a monitorização orientada aos recursos apresentada com mais detalhe

informações relativas à RAM e CPU mas que não mostra qualquer informação de como os diferentes componentes estão interligados e que impacto isso tem. Já quando se comparou o CloudOverWatch com o WeaveScope, percebeu-se que o WeaveScope requer em média menos cliques para fazer a mesma ação e mostra mais alguma informação. Contudo percebeu-se que o modo de disposição dos nós na rede por parte do CloudOverWatch é quanto melhor que o do WeaveScope quanto mais complexa for a orquestração.

Por último, para trabalho futuro, poderia ser modificado o tipo de recursos que o utilizador pode interagir ao nível do Kubernetes porque a aplicação dispõe de uma secção para interagir e modificar os recursos de uma orquestração no Kubernetes, mas apenas foram escolhidos alguns atributos - e.g. limite memória, limite CPU, número de réplicas - mas muitos mais poderiam ter sido adicionados, apenas não o foi feito porque o objetivo, no âmbito desta dissertação, era comprovar que era possível editar alguns recursos. Outro ponto a melhorar é que quanto mais se queria adicionar uma nova funcionalidade, ao nível de visualização os dados, mais se sentiu restrições por parte dos Vis.JS. Como alternativa, poderia-se criar uma biblioteca de visualização e aproveitar a versão modificada do NeoVis para fazer a ligação entre essa biblioteca e a aplicação CloudOverWatch. Ambos estes pontos estão fora do foco desta dissertação, mas valerá a pena explorá-los para criar uma verdadeira alternativa competitiva ao WeaveScope e a outros sistemas de monitorização.

Apesar de ainda não ser uma alternativa competitiva ao WeaveScope, a base desenvolvida representa um novo passo no mundo da monitorização de sistemas baseados em contentores, com a facilidade com que se consegue visualizar a troca de dados e outras métricas e através destas perceber o que está de errado ou não com a implementação a ser monitorizada. Facilmente com o trabalho desenvolvido no âmbito desta dissertação e com trabalho desenvolvido no SYSQUERY, poderá se criar uma verdadeira alternativa aos sistemas dominantes no mercado.

BIBLIOGRAFIA

[1] About images, containers and storage drivers. https://docs.docker.com/v17.09/engine/userguide/storagedriver/imagesandcontainers/#images-and-layers. [Online; 12-December-2019].

[2] Approaches to Configuration Management: Chef, Ansible, and Kubernetes. https://kublr.com/blog/configuration-management-chef-ansible-and-kubernetes/. [Online; 12-September-2020].

[3] Containers vs VMS. https://www.redhat.com/en/topics/containers/containers-vs-vms. [Online; 10-September-2020].

[4] Containers vs VMS image. https://www.docker.com/blog/containers-replacing-virtual-machines/. [Online; 10-September-2020].

[5] How to Setup Prometheus Monitoring On Kubernetes Cluster. https://devopscube.com/setup-prometheus-monitoring-on-kubernetes/. [Online; 12-September-2020].

[6] Install on Debian or Ubuntu. https://grafana.com/docs/grafana/latest/installation/debian/2-start-the-server. [Online; 17-September-2020].

[7] Introducing WeaveScope. https://www.weave.works/docs/scope/latest/introducing/. [Online; 27-July-2020].

[8] Kubernetes. https://kubernetes.io/docs/concepts/overview/components/. [Online; 2-December-2019].

[9] Kubernetes. https://kubernetes.io/pt/. [Online; 27-November-2019].

[10] Prometheus vs. Heapster vs. Kubernetes Metrics APIs. https://blog.containership.io/kubernetes-metrics-collection-options/. [Online; acessed 8-December-2019].

[11] What is a container. https://www.docker.com/resources/what-container. [Online; 29-September-2019].

[12] Why should you use microservices and containers? https://developer.ibm.com/articles/why-should-we-use-microservices-and-containers/. [Online; accessed 7-December-2019].

[13] Black-box inter-application traffic monitoring for adaptive container placement. 2019.

[14] Pooyan Jamshidi Armin Balalaie, Abbas Heydarnoori. Microservices Architecture Enables DevOps. 2016.

[15] En-Hau Yeh Phone Lin Chia-Chen Chang, Shun-Ren Yang and Jeu-Yih Jeng. A Kubernetes-Based Monitoring Platform for Dynamic Cloud Resource Provisioning. 2017.

[16] Rajiv Ranjan Chang Liu Lydia Chen Massimo Villari Maria Fazio, Antonio Celesti. Open Issues in Scheduling Microservices in the Cloud. 2016.

[17] Ram Rajamony Juan Rubio Wes Felter, Alexandre Ferreira. An Updated Performance Comparision of Virtual Machines and Linux Containers. 2015.

A

APÊNDICE A - TABELA DE COMPARAÇÃO

		Kubectl		CloudOverWatch		WeaveScope	
Namespace	Nome do Pod	Ip	Host	Ip	Host	Ip	Host
loadtest	load-test-59d898fb7f-9j2sc	10.244.2.6	2	10.244.2.6	2	10.244.2.6	2
loadtest	load-test-59d898fb7f-r589v	10.244.3.7	3	10.244.3.7	3	10.244.3.7	3
sock-shop	carts-66bc68f95f-h8wzj	10.244.2.2	2	10.244.2.2	2	10.244.2.2	2
sock-shop	carts-db-7d79c89fdb-54g24	10.244.1.2	4	10.244.1.2	4	10.244.1.2	4
sock-shop	catalogue-6b5f68f7b6-fz27t	10.244.3.6	3	10.244.3.6	3	10.244.3.6	3
sock-shop	catalogue-db-55965799b9-8mxqs	10.244.3.4	3	10.244.3.4	3	10.244.3.4	3
sock-shop	front-end-6694dcd58f-dsp76	10.244.1.3	4	10.244.1.3	4	10.244.1.3	4
sock-shop	orders-65bb8fcf9b-5rwrx	10.244.1.4	4	10.244.1.4	4	10.244.1.4	4
sock-shop	orders-db-6cdb57dc6d-p7klj	10.244.2.4	2	10.244.2.4	2	10.244.2.4	2
sock-shop	payment-6465b75bb6-fgx2k	10.244.2.3	2	10.244.2.3	2	10.244.2.3	2
sock-shop	queue-master-6b5b5c7658-s2n9z	10.244.3.2	3	10.244.3.2	3	10.244.3.2	3
sock-shop	rabbitmq-7764597b7b-dnwmq	10.244.1.5	4	10.244.1.5	4	10.244.1.5	4
sock-shop	shipping-7f86cf76f8-wl8w2	10.244.1.6	4	10.244.1.6	4	10.244.1.6	4
sock-shop	user-8c7b575cc-j8hvq	10.244.2.5	2	10.244.2.5	2	10.244.2.5	2
sock-shop	user-db-86dcc8cdb5-j26wr	10.244.3.5	3	10.244.3.5	3	10.244.3.5	3

B

APÊNDICE B - TABELA KLM

Operador	Valor
Keystroke	0.2s
Point	1.1s
Button	0.1s
Home	0.4s
Mental	1.2s
W1aiting	0.8s
W2aiting	0.6s
W3aiting	1s
W4aiting	0.2s

Existem vários Waiting porque cada se refere a um *loading time* de uma página diferente.

Tabela 5: Valores para avaliação KLM

C

APÊNDICE C - GUIA RÁPIDO DE UTILIZAÇÃO

Neste capítulo será apresentado como iniciar a aplicação, primeiramente no âmbito geral e depois com um caso de uso.

Para começar a correr a aplicação são precisos três componentes:

- SYSQUERY - Composto por três pastas - deployment, sysquery e bench
- yamlChanger : API para modificar Kubernetes
- app-monitor : aplicação principal

Ao todo são cinco pastas e para evitar conflitos é preferível estarem todas no mesmo path.

C.1 GERAL

Podemos começar por iniciar a aplicação, dentro da pasta app-monitor.

```
1   $  npm run serve
```

E, noutra consola, dentro da pasta dos yamlChangers também se inicia o servidor que permite comunicar a aplicação com o Kubernetes.

```
1   $  node server.js
```

Tem de ser criar uma instância de Neo4j.

```
1   $  docker run --publish=7474 --publish=7687:7687 neo4j
```

Noutra consola, temos de ter o *path* para a pasta *deployment*. Caso não seja a primeira vez a executar todos estes comandos, tem de se eliminar todos a informação sobre a monitorização anterior.

```
rm agent.*.pickle
```

De seguida iniciamos as máquinas e o Kubernetes.

```
$ ./startinstances.sh;
$ ./startk8s.sh;
```

Depois fazemos *deploy* de todos os yamls que precisamos para por a orquestração a correr.

```
$ ./deploy.sh {path_to_file}
```

De seguida começamos a monitorizar a aplicação.

```
$ ./startagents.sh
```

E quando quisermos podemos terminar.

```
$ ./stopagents.sh
```

E, por fim, enviamos os dados para o Neo4j.

```
$ ./loadresults.sh --clear --k8s services.txt agent.*.pickle
```

Noutra consola, ligamos à primeira instância da orquestração para fazer proxy do kubectl , para se conseguir aceder à API do Kubernetes. Não obstante, liga-se também a porta local

8001 à porta 8001 da máquina - porta onde corre o proxy - para o yamlChanger funcionar corretamente.

```
$ gcloud compute ssh --ssh-flag="-L 8001:localhost:8001  instance-1
$ kubectl proxy &
```

C.2 CASO DE USO

Um dos benchmarks usados foi o Sock-shop[1]. Se o adicionarmos dentro da pasta de deployment, os comandos para por a correr com o CloudOverWatch - por cronológica - são os seguintes:

Terminal 1

```
user@user-computerName:~/tese$ cd ~/tese/app-monitor
user@user-computerName:~/tese/app-monitor$ npm run serve
```

Terminal 2

```
user@user-computerName:~/tese$ cd ~/tese/yamlChanger
user@user-computerName:~/tese$ node server.js
```

Terminal 3

Para iniciar um servidor de Neo4j. É preciso mudar a password. Para efeitos desta dissertação é 123456.

```
user@user-computerName:~/tese$ docker run --publish=7474 --publish=7687:7687 neo4j
```

Terminal 4

```
user@user-computerName:~/tese$ cd ~/deployment
user@user-computerName:~/tese/deployment$ ./startinstances.sh
```

1 Github: https://github.com/microservices-demo/microservices-demo

```
3   user@user-computerName:~/tese/deployment$ ./startk8s.sh
```

Terminal 5

```
1   user@user-computerName:~/tese$ gcloud compute ssh --ssh-flag="-L 8001:localhost:8001
        instance-1
2
3   //ssh para maquina
4
5   user@instance-1:~$ kubectl create namespace sock-shop
6   user@instance-1:~$ kubectl proxy &
```

Terminal 4

```
1   user@user-computerName:~/tese/deployment$ ./deploy.sh ./microservices-demo/deploy/kubernetes
        /complete-demo.yaml
2   user@user-computerName:~/tese/deployment$ ./deploy.sh ./microservices-demo/deploy/kubernetes
        /manifests/loadtest-dep.yaml
3    user@user-computerName:~/tese/deployment$ ./startagents.sh
4    user@user-computerName:~/tese/deployment$ ./stopagents.sh
5    user@user-computerName:~/tese/deployment$ ./loadresults.sh --clear --k8s services.txt agent
         .*.pickle
```

NB: place here information about funding, FCT project, etc in which the work is framed. Leave empty otherwise.