Universidade do Minho
Mestrado Integrado em Engenharia Informática
System Deployment and Benchmarking
Relatório da Fase 2 do Projeto Prático
Grupo 7

novembro 2020

Bruno Carvalho
A83851

Lázaro Pinheiro
A86788

Luís Cunha
A84244

Gonçalo Almeida
A84610

Gonçalo Ferreira
A84073

Conteúdo

Lista de Figuras

1 Introdução

O presente relatório desenvolve-se no âmbito da Unidade Curricular *System Deployment and Benchmarking*, tendo como objectivo expor o trabalho realizado em ambas as fases. Na primeira fase, o objetivo pretendido era a seleção, caracterização e análise de uma aplicação distribuída. Na segunda fase pretende-se, utilizando a análise feita na fase anterior à aplicação selecionada, realizar a automatização do processo de instalação, monitorização e avaliação da mesma.

Optou-se por seleccionar a aplicação *Mattermost*, que é uma plataforma de mensagens instantâneas segura *open-source*, direcionada ao trabalho de equipa, oferecendo a todos os utilizadores facilidade para conversar em *chats* (privados ou públicos). O ponto forte desta aplicação é a possibilidade de hospedagem e centralização da plataforma de comunicação, de forma privada, numa *NAS* local, permitindo o aumento do controlo, da disponibilidade, da confidencialidade e um alto potencial de armazenamento. Os utilizadores podem interagir com o sistema através das aplicações para computador, dispositivos móveis e aplicação *web*.

Como era requisito do enunciado, este relatório comporta a descrição da arquitectura e componentes da aplicação, padrões de distribuição usados e formas de comunicação, a descrição dos pontos de configuração e a identificação de operações críticas, que representam possíveis limitações ao desempenho. Ainda, aborda o processo de instalação, monitorização e avaliação da aplicação *Mattermost*.

Com a realização deste trabalho prático, o grupo objectiva uma melhor compreensão do processo de instalação, monitorização e avaliação de aplicações.

2 Arquitetura e Componentes da Aplicação

A *Mattermost*, na sua generalidade, consiste num servidor Go exposto como um servidor *Restful JSON* ligado a uma base de dados *SQL* e a um serviço de armazenamento de ficheiros, com clientes Javascript e Go. Um modelo desta arquitetura pode ser visto na Figura 2.

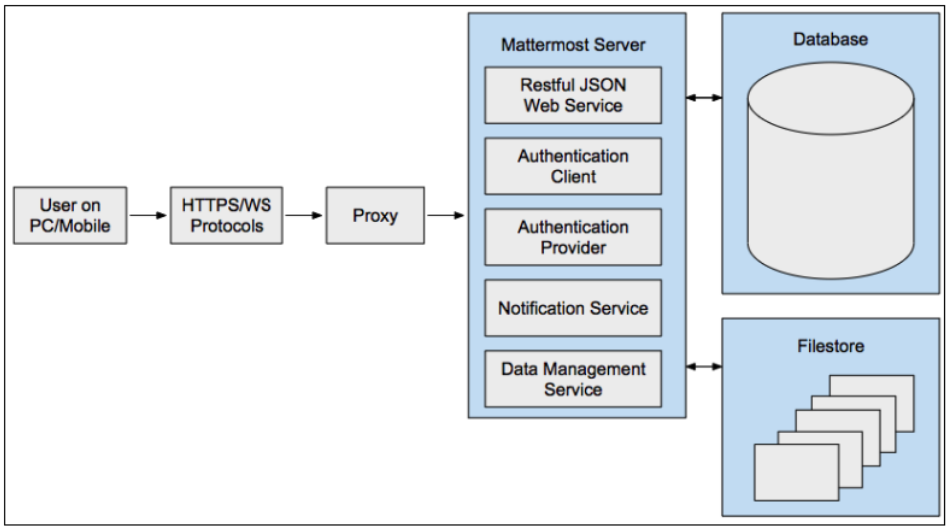

Figura 2: Arquitetura da Aplicação [1]

2.1 Servidor

O servidor é instalado através de um único ficheiro binário e pode ser configurado através do ficheiro *config/config.json*. Esta configuração pode ser feita diretamente no ficheiro ou através de uma interface *web* (*System Console*). Este pode ser acedido através de uma *RESTful API*.

Tal como está representado na Figura 2, o servidor contém vários componentes que são acedidos a partir da API. O *authentication client* fornece serviços de autenticação aos utilizadores através de *e-mail* e palavra-passe (a *Enterprise Edition* tem mais soluções). O *authentication provider* permite que a autenticação seja feita a partir de outros serviços, tais como o GitLab. O *notification service* está encarregue de gerir as notificações e, por fim, o *data management service* está ligado às bases de dados e *filestores* de forma a controlar o acesso aos dados.

Se a aplicação estiver configurada com a *Enterprise Edition*, pode usufruir de servidores em modo *cluster*. Esta abordagem permite por um lado, que a latência de acesso seja minimizada por se colocarem servidores fisicamente mais perto dos clientes, e por outro, que seja possível fazer balanceamento de carga entre servidores, assim como o *handoff* de tráfego entre servidores em cenários de falha. [3]

2.2 Base de Dados

A base de dados (*MySQL* ou *PostgreSQL*) é responsável pelo armazenamento de dados do sistema e pela execução de pesquisas de texto.

Em ambientes empresariais (*Enterprise Edition)*, a base de dados pode possuir várias réplicas de leitura e consulta (*queries*). As réplicas de leitura podem, por exemplo, permitir que a base de dados *master* encaminhe operações para as réplicas em caso de falha, de forma a que esta não tenha um impacto significativo no funcionamento da aplicação. As réplicas de consulta podem ser configuradas de forma a que, por exemplo, cada uma apenas lide com um determinado tipo de *query*. [4]

2.3 Armazenamento de Ficheiros

O armazenamento de imagens e ficheiros pode ser configurado diretamente no servidor *Mattermost*, num servidor *NAS* (*Network-Attached Storage*) ou num serviço externo de armazenamento de ficheiros, como o *Amazon S3*, dependendo das necessidades dos utilizadores e da escala da implementação. [5]

2.4 Servidor *Proxy*

É aconselhado o uso de um servidor *proxy* entre o cliente e o servidor principal de forma a fornecer:

- Melhor segurança, pois pode gerir as comunicações SSL com o servidor.
- Melhor desempenho através de técnicas de balanceamento de carga entre múltiplos servidores.
- Monitorização de tráfego de dados, tais como transferências de ficheiros.

Na Figura 3 encontra-se um exemplo de uma implementação em modo de alta disponibilidade, disponível na *Enterprise Edition*.

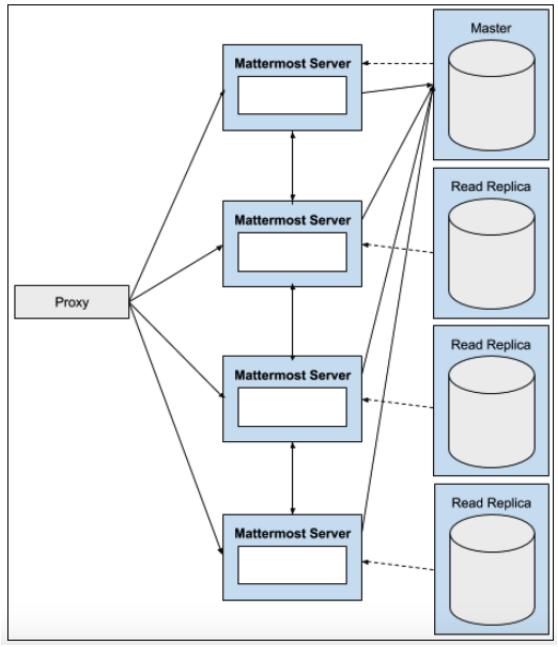

Figura 3: Exemplo de uma Organização de Alta Disponibilidade [2]

3 Distribuição

3.1 Padrões de Distribuição

Esta aplicação pode assumir vários padrões de distribuição, dependendo da forma como é configurada.

Como podemos observar pela Figura 2, caso se opte por não usar um servidor *proxy*, o cliente comunica diretamente com o servidor *Mattermost* através dos protocolos de comunicação. Neste caso, a aplicação segue uma arquitetura *client-server*.

Caso contrário, podemos utilizar uma arquitetura *proxy server*, em que os clientes comunicam com um servidor *proxy* e este comunica com os diversos servidores *Mattermost*. Neste caso, não há comunicação direta entre cliente e servidor.

Na *Enterprise Edition*, existe suporte a uma arquitetura de servidores *Mattermost* em *cluster*. Tal como demonstra a Figura 3, o servidor *proxy* pode distribuir a carga para servidores *Mattermost* que se encontrem com menos carga ou que se localizem mais perto do cliente.

Há também o suporte a uma arquitetura de réplicas de leitura do servidor da base de dados. Na Figura 3, podemos ver o servidor da base dados *master* e as várias réplicas. Desta forma a carga é distribuída e, espalhando as réplicas por diversos pontos geográficos, também conseguimos diminuir a latência das comunicações.

Em geral, a escalabilidade do sistema segue uma arquitetura *service oriented*, uma vez que escalonamos de forma separada os diversos tipos de entidades (*i.e.* armazenamento, bases de dados e servidores).

3.2 Formas de Comunicação

A aplicação é compatível tanto com HTTPS (*Secure HyperText Transfer Protocol*) como com WSS (*Secure WebSocket*). No entanto, existem diferenças na utilização dos dois protocolos.

Uma ligação HTTPS com o servidor fornece funcionalidades básicas e a capacidade de fazer *render* de páginas. No entanto, não suporta a interatividade em *real-time* disponibilizada pelo WSS. Caso não seja possível estabelecer uma ligação HTTPS, a aplicação não funcionará. Uma configuração HTTP pode ser utilizada para testes iniciais, mas não é aconselhada em produção.

Uma ligação WSS com o servidor fornece atualizações e notificações em *real-time*. Caso este tipo de ligação não esteja disponível, e o HTTPS esteja, o sistema continuará a funcionar sem as funcionalidades há pouco referidas (*e.g.*, as páginas só serão atualizadas com um *refresh*)[6].

4 Pontos de Configuração

As configurações do servidor do *Mattermost* encontram-se definidas num ficheiro de configuração *mattermost/config/config.json*. Este ficheiro pode ser modificado através de um editor de texto ou da consola do sistema, desde que tenha permissões para tal, o que implica o *reload* deste.

O ficheiro anteriormente referido, é gerado através do código localizado em *mattermost-server/config/config_generator* que recorre ao modelo de configurações presente em *mattermost-server/model/model.go*

Para qualquer definição não definida em *config.json*, o servidor do Mattermost utiliza os valores default documentados em [7].

Algumas das principais configurações são:

- Web Server:
  - SiteURL: define o URL que os utilizadores utilizam para aceder ao Mattermost. É necessário indicar o número da porta caso esta não seja uma porta *standard* como 80 ou 433. Um exemplo de um valor desta configuração é "https://example.com/company/mattermost".
  - ListenAddress: define o endereço e porta para se conectar e ouvir. Ao atribuir o valor ":8056"vão ser conectadas todas as interfaces de rede. Ao atribuir o valor "127.0.0.1:8065"vai apenas ser conectada a interface de rede com o IP descrito.
  - Outras configurações: Forward80To443, ConnectionSecurity, TLSCertFile, TLSKeyFile, UseLetsEncrypt, LetsEncryptCertificateCacheFile, ReadTimeout, WriteTimeout, IdleTimeout, EnableAPIv3, WebserverMode, EnableInsecureOutgoingConnections, ManagedResourcePaths.
- Base de Dados:
  - DriverName: define o *driver* da base de dados e pode assumir os valores "mysql"e "postgres".
  - DataSource: define a *connection string* da base de dados principal. Quando o *DriverName* assume o valor "postgres", a forma da *string* é "postgres://mmuser:password@localhost:5432/mattermost_test?sslmode=disable&connect_timeout=10". Quando o *DriverName* assume o valor "mysql", a forma da *string* é "mysql://mmuser:password@localhost:5432/mattermost_test?sslmode=disable&connect_timeout=10".

  - Outras configurações: MaxIdleConns, MaxOpenConns, QueryTimeout, DisableDatabaseSearch, ConnMaxLifetimeMilliseconds, MinimumHashtagLength, AtRestEncryptKey, Trace.

- Armazenamento de Ficheiros

  - DriverName: define o *driver* do sistema de armazenamento de ficheiros e pode assumir os valores "local"(valor *default*) e "amazons3".

  - Directory: define a diretoria em que os ficheiros são escritos. Quando o *DriverName* assume o valor "local", o valor desta configuração é relativo à diretoria onde o Mattermost está instalado. Quando o *DriverName* assume o valor "amazons3", o valor *default* desta configuração é "./data/".

  - MaxFileSize: define o tamanho máximo dos ficheiros (em megabytes) guardados na *System Console UI*.

- Servidor Proxy

  - Enable: define se um *image proxy* está ativo para imagens externas de modo a preveni-las de se conectarem diretamente aos servidores remotos e pode assumir os valores "true"e "false".

  - ImageProxyType: define o tipo do *image proxy* utilizado e pode assumir os valores "local"(o próprio servidor Mattermost serve de *image proxy*) e "atmos/camo"(é utilizado um *atmos/camo image proxy* externo).

  - RemoteImageProxyURL: define o URL do *atmos/camo proxy*.

  - RemoteImageProxyOptions: define a URL *signing key* passada a um *atmos/camo proxy*.

5 Operações Críticas

5.1 Comunicação cliente-servidor

Através da Figura 2 podemos observar que a comunicação cliente-servidor, e vice-versa, é balanceada através de um *proxy*. Ora, visto que toda a comunicação passa por este componente, torna-se um ponto único de falha, ou seja, uma falha que compromete o funcionamento do sistema em geral.

Os serviços de autenticação que o servidor fornece (*authentication client* e *provider*) devem também estar sempre disponíveis, uma vez que tratam da autenticação dos utilizadores. Sem estes serviços ativos, não lhes é possível acederem ao resto do sistema.

De modo a viabilizar as características anteriormente mencionadas, é proposto um modelo de *Cluster* para o *Proxy*. Seguindo o modelo Servidor Proxy (consultar 2.4) é possível usar diversas tecnologias como *NGINX proxy* ou *Apache 2*.

Esta solução oferece diversos mecanismos como subdivisão de tarefas, e para solucionar o problema de falha, usa-se um método Activo-Passivo. Neste o agente Passivo, caso exista alguma falha com o servidor Activo, encarrega-se de substituir a sua função sem que haja algum tipo de quebra no serviço.

A autenticação aproveita este modelo de comunicação para balancear a sua carga por um conjunto de servidores Mattermost, como é possível observar na Figura 3, assim proporcionando uma alta disponibilidade do serviço de autenticação e não só.

5.2 Base de Dados e Armazenamento de Ficheiros

A disponibilidade dos dados é também um ponto crítico do sistema. Sem estes, o sistema torna-se praticamente inútil, uma vez que os utilizadores não têm acesso à informação que procuram.

Com uma arquitetura simples deste serviço, apenas com um servidor base de dados, é difícil garantir a alta disponibilidade e redundância de dados como também origina ponto de falha único, tornando este inacessível. O exemplo remete também para a questão de débito, que dada um súbito aumento de clientes reflete uma baixa de performance do serviço, provocando *overflow* de pedidos.

Com objetivo de solucionar este problema, foi proposto a utilização de um conjunto de bases de dados, gerido por um *master* que sincroniza a informação com as restantes replicas de leitura. Este mecanismo oferece replicação de informação, escalabilidade e reduz a latência com o consumidor final (consultar 2.2).

À semelhança do serviço base de dados, o serviço de armazenamento de ficheiros deparasse com os mesmos problemas. Face a este obstáculo propõem-se utilizar um mecanismo de cópia de ficheiros entre servidores

master e *readable*, com recurso a utensílios de sistemas de replicação de dados, sendo possível atingir a solução desejada.

6 Ferramentas utilizadas

6.1 *Ansible*

A ferramenta de aprovisionamento escolhida para implementar esta aplicação é o *Ansible* [8]. Este permite, de forma automática e estruturada, preparar as máquinas onde a aplicação será instalada de forma a que satisfaçam todos os requisitos.

Neste caso, o *playbook* de *Ansible* começa por tratar da satisfação de todas as dependências da aplicação(*e.g., Docker*). De seguida, procede à instalação e configuração do *Elasticsearch* e do *Kibana* numa máquina e da instalação e configuração da *Mattermost* e dos componentes de monitorização (*Beats*) noutras máquinas. Tal é feito copiando os ficheiros *Docker* e de configuração para as respetivas máquinas e invocando o *docker-compose*, que se encarrega de as executar.

6.2 *Docker*

Tanto os componentes da *Mattermost* como da *Elasticstack* estão compartimentalizados em *containers Docker*, de forma a facilitar a sua instalação em várias máquinas distintas [9]. Estes estão separados em dois grupos: os relacionados com a *Mattermost* e a recolha de dados desta, e os relacionados com a agregação dos dados de monitorização e a sua visualização. Para tirar partido desta separação, e para facilitar a execução de vários *containers* ao mesmo tempo, usou-se o *docker-compose*. Assim, a execução do *compose* permite, de uma vez só, criar, configurar e executar um grupo de *containers*.

6.3 *Apache JMeter*

A aplicação *Apache JMeter* é projetada para testar o comportamento funcional de carga e medir o desempenho de servidores, podendo ser usada para realizar estes testes em recursos estáticos e dinâmicos. Pode também simular diferentes tipos de cargas num servidor ou grupo de servidores, de modo a testar a performance e desempenho geral.

7 Instalação

Recorrendo à ferramenta *Ansible*, foi possível a automatização da configuração e do provisionamento da aplicação *Mattermost* na *Google Cloud Platform*. Sendo esta plataforma um serviço de *cloud* pertencente ao grupo de abstração *Infrastructure-as-a-Service* ***(IaaS)***, esta segue o modelo presente na Figura 4, que demonstra quais as entidades responsáveis por gerir cada componente.

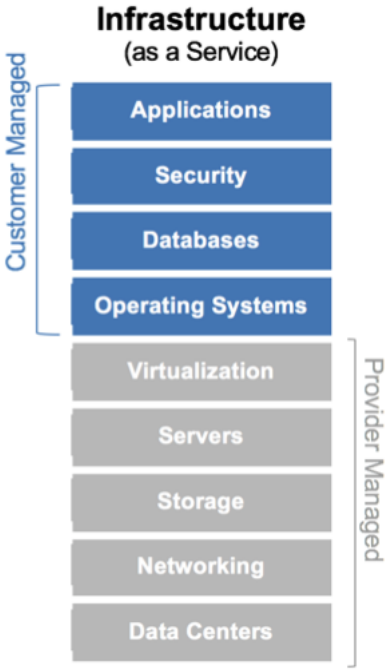

Figura 4: Diagrama modelo do grupo de abstração *Infrastructure-as-a-Service*

Desta forma, são fornecidos ao *developer* os recursos físicos virtualizados como a capacidade de processamento, o armazenamento, a rede, entre outros. Esta metodologia traz vantagens como o evitar de custos iniciais na gestão da infraestrutura e *hardware* e a ilusão de recursos praticamente infinitos. Estas vantagens conferem à implantação da aplicação escalabilidade e flexibilidade. Contudo, não permitem o controlo sobre o *hardware* e *software* de virtualização.

7.1 Arquitetura e Limitações

Como método de virtualização recorreu-se a *containers* para os serviços de servidor *web*, servidor aplicacional e base de dados, sendo que estes foram implementados com a utilização da ferramenta *Docker*. A arquitetura

da aplicação encontra-se representada na Figura 5. Este método permite, comparativamente a máquinas virtuais, uma maior facilidade nas atividades de teste, aprovisionamento e migração, bem como uma melhor utilização dos recursos conforme os custos, a possibilidade de instalação tanto em servidores físicos como virtuais e, por fim, um melhor desempenho.

É também utilizado um balanceador de carga com vista a distribuição das interações dos clientes com o sistema, atribuindo a cada pedido um determinado servidor, tendo este um IP estático.

O serviço dispõe de um conjunto de máquinas virtuais (*cluster*) onde cada uma destas possui, além dos componentes de monitorização, três *containers* distintos que possuem os serviços distribuídos a seguir descritos.

- **Balanceador de Rede Interna** - Distribuí a carga por todos os servidores aplicacionais(*Mattermost*) existentes na mesma máquina anfitriã, conseguido recorrendo à ferramenta *Nginx*, que constituí o *container web*. De salientar que cada um destes *containers* encontra-se ligado a apenas um sevidor aplicacional;

- **Servidor Aplicacional** - Representado pelo *container app*, consiste na lógica de negócio e funcionalidades do servidor *Mattermost* que possibilita o processamento dos pedidos dos clientes. Note-se que, cada um destes encontra-se ligado a todas as instâncias da base de dados;

- **Base de Dados** - Através do **SGBD** *PostgreSQL*, é fornecido o serviço de armazemento de dados necessários para o normal funcionamento da aplicação. Este serviço encontra-se divido em dois tipos de serviços distintos. Os quais são:

  - ***Master*** - Representado pelo *container pg_master*, este é responsável pelas operações de escrita/leitura e ainda pela sincronização dos dados com as várias réplicas.
  - ***Slave*** - Representado pelo *container pg_slave*, esta é uma replicação da base de dados *master*, possibilitando assim a relaxação das leituras, e também uma maior concorrência no acesso aos dados, uma vez que existe redundância dos mesmos, diminuindo o impacto das leituras no sistema.

  Todas os *containers* de bases de dados encontram-se ligados, por forma a sincronizar os dados.

De modo a possibilitar a comunicação entre os serviços distribuídos pelos diversos *containers* em máquinas diferentes, definiu-se uma rede do tipo *host*, onde cada *container* possuí a mesma interface de rede da máquina anfitriã.

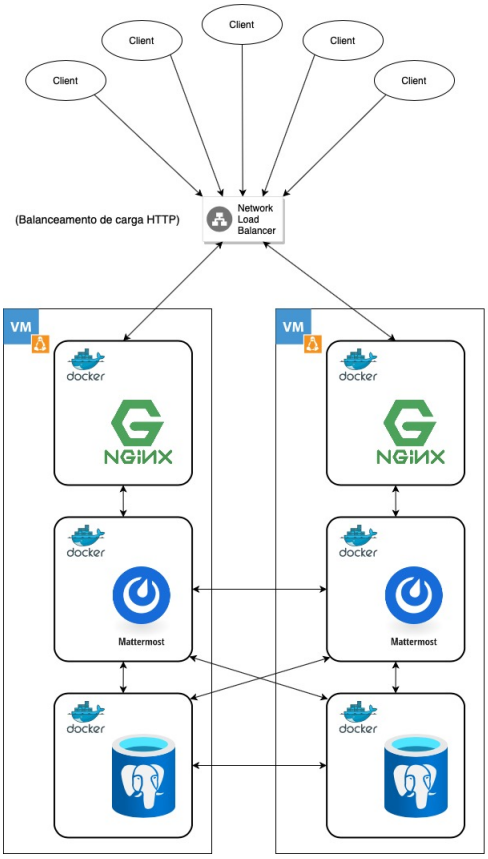

Figura 5: Diagrama simplificado da arquitetura implementada

A decisão do uso de *containers* é devidamente justificada pelo facto de serem utilizadas diferentes componentes para diferentes finalidades, facilitando a distribuição destas por várias máquinas. Possibilita ainda um melhor aproveitamento dos recursos, tornando esta implementação mais económica e de fácil atualização durante produção. A solução implementada (c.f. Figura 5) demonstra um encapsulamento nos diferentes componentes, que facilita a adição e configuração dos mesmos. Além disso, permite uma fácil integração de novos componentes, sem modificação dos já existentes, uma vez que se todos se encontram devidamente compartimentalizados.

7.2 Estrutura da solução de instalação

A solução de automatização da instalação desta aplicação tem a seguinte estrutura:

- **playbook.yml**: *Playbook* de *Ansible* responsável por executar toda a instalação.
- **hosts.inv**: Armazenamento dos endereços IP das várias máquinas onde a aplicação será instalada.
- **requirements.yml**: Alguns requisitos para a execução do *playbook*.
- *docker-pg-master*: Ficheiros respetivos á configuração do *container postgresql master*.
  - *master*: Ficheiros respetivos á configuração do *container master*.
    - **Dockerfile**
    - **setup-master.sh**
  - **docker-compose.yml**: Ficheiro *docker compose* para aprovisionamento da base de dados coordenadora.
  - **docker-swarm.yml**
- *docker-pg-slave*: Ficheiros de configuração do *postgresql slave*.
  - *slave*: Ficheiros de configuração do *container slave*.
    - **Dockerfile**
    - **docker-entrypoint.sh**
    - **gosu**
  - **.env**: Variáveis de ambiente respetivas á configuração.
  - **docker-compose.yml**: Ficheiro *docker compose* para aprovisionamento da base de dados réplica.
  - **docker-swarm.yml**
- *elasticstack*: Contém os ficheiros de configuração dos vários *beats*, do *Logstash* e do *Elasticsearch* e do *Kibana*.
- *mattermost*:
  - *Dockerfiles* e ficheiros auxiliares do servidor aplicacional, interface *web* e base de dados.
  - **.env**: Contém algumas variáveis de ambiente.
  - **travis.yml**: Configuração dos *containers*.
  - **docker-compose.yml**: Ficheiro *compose* responsável por criar, configurar e executar os *containers* da *Mattermost* e os *beats*.
- *roles*: Ficheiros *Ansible* com as várias tarefas necessárias à instalação e configuração da aplicação.

7.3 Execução

Para proceder à instalação da aplicação, é primeiro necessário verificar os endereços IPs presentes em *hosts.inv*, *mattermost/.env* e *docker-pg-slave/.env*. De seguida, a execução de

```
ansible-playbook -i hosts.inv playbook.yml
```

trata da instalação de todas as componentes da aplicação nas máquinas indicadas.

Corrido o *playbook*, em `[app]:80` é possível aceder à *Mattermost*, e a monitorização pode ser feita através do *Kibana* em `[metrics]:5601`, em que `[app]` e `[metrics]` são os IPs indicados em *hosts.inv*.

8 Monitorização

A monitorização da aplicação é feita usando os seguintes componentes da Elasticstack:

- **Metricbeat**: Monitorização de métricas de desempenho da máquina *host* e dos diversos componentes da aplicação.
- **Packetbeat**: Monitorização do tráfego de rede.
- **Heartbeat**: Monitorização da disponibilidade dos diversos componentes da aplicação.
- **Filebeat**: Monitorização dos *logs* dos diversos componentes da aplicação.
- **Elasticsearch**: Agregação e pesquisa de dados.
- **Kibana**: Visualização dos dados.

Como é possível observar na Figura 6, os componentes responsáveis pela recolha dos dados são instalados juntamente com a aplicação e enviam os dados por HTTP para o *Elasticsearch*, que é acedido pelo *Kibana*. Estes dois últimos componentes encontram-se numa máquina distinta das que executam a *Mattermost*.

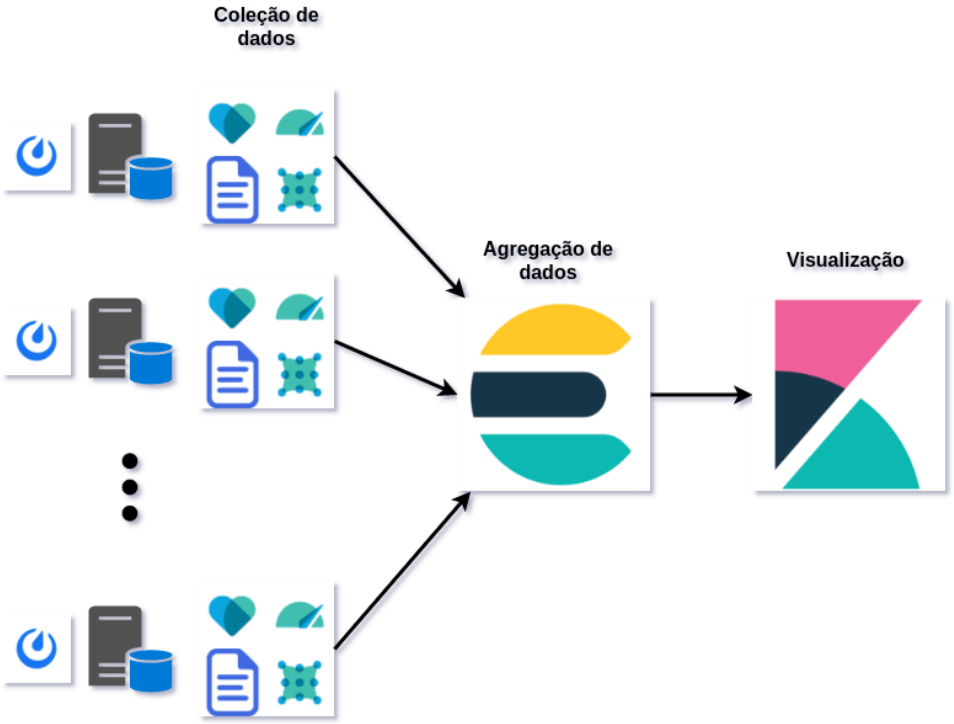

Figura 6: Arquitetura da monitorização

Esta monitorização permite avaliar de uma forma completa o estado da aplicação uma vez que permite visualizar, num único local, todos os aspetos relevantes que levam ao bom funcionamento desta. Através dos dados recolhidos pelo *Metricbeat*, podemos ver qual a carga atual das máquinas onde corre a aplicação, assim como qual a utilização de recursos dos componentes individuais da aplicação através da monitorização direta dos *containers* onde estes executam. O *Packetbeat* permite uma análise completa da rede, fornecendo dados sobre o tipo de tráfego que circula, a sua origem e destino, e quais as tendências deste. É também possível ter uma análise constante do estado dos serviços através do *Heartbeat*, que monitoriza constantemente se estes se encontram ativos ou não. Por fim, o *Filebeat* faz uma agregação de todos os registos de *log*, de forma a que os registos dos vários componentes possam ser acedidos facilmente, num único local.

O *Elasticsearch* é responsável por agregar todos estes dados de forma facilmente pesquisável, que depois são acedidos pelo *Kibana*. Este último fornece várias formas de visualização intuitiva para que seja possível uma deteção rápida de anomalias e ter uma visão geral do desempenho da aplicação.

De notar que o *Logstash* é instalado juntamente com o resto destes componentes, apesar de não estar em utilização. Caso estivesse, estaria no lugar do *Elasticsearch* na Figura 6, encaminhando depois os dados para o *Elasticsearch*, seguindo para o *Kibana*. O papel do *Logstash* consistiria em processar os vários ficheiros em *pipelines* antes de serem enviados para o *Elasticsearch* e visualizados no *Kibana*. No entanto, não foi possível criar um pipeline que fosse capaz, de forma confiável, de processar os dados, o que fazia com que houvesse alguns campos que o *Elasticsearch* não interpretava corretamente. Por isso, este não faz parte da configuração atual, mas é enviado juntamente com o resto da *Elasticstack* para uma eventual futura reparação. A alteração para uma configuração com o *Logstash* apenas consistiria na troca dos campos de *output* de cada um dos ficheiros de configuração dos *beats* para a porta onde o *Logstash* estaria à escuta, em vez da porta do *Elasticsearch*.

9 Avaliação

Para a avaliação foi usado JMeter e foram testadas duas versões da aplicação: uma versão *standalone* e uma versão *3-way cluster*. Todos os servidores encontravam-se situados na Bélgica e possuiam 2 CPUs e 4GB de memória. Deste modo, a localização e as especificações dos servidores influenciaram de igual forma os resultados obtidos nos diferentes testes. Foram testados também, dois pedidos: envio de mensagens e envio de ficheiros. Cada teste foi executado utilizando 1 thread e com a duração de 60 segundos.

Os resultados obtidos foram os seguintes:

Standalone - envio de mensagens

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	749	62	8.28%	79.54	36	183	81.00	90.00	97.50	121.50	12.49	9.54	3.87

Standalone - envio de ficheiros

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	106	0	0.00%	568.65	405	3795	511.00	725.00	832.60	3614.54	1.75	2.97	2701.33

Cluster - envio de mensagens

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	792	0	0.00%	75.16	65	176	73.00	82.00	88.00	122.21	13.21	10.56	4.08

Cluster - envio de ficheiros

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	97	0	0.00%	621.57	407	3150	512.00	823.00	919.10	3150.00	1.61	2.70	2471.68

Figura 7: Comparação das estatísticas

Sample	#Samples	#Errors	Error	#Errors
Total	749	62	429/Too Many Requests	62

Figura 8: Erros da versão standalone no envio de mensagens

Como podemos observar pela Figura 7, em relação ao envio de mensagens, a versão *cluster* apresentou um maior débito, maior número de pedidos respondidos e menor tempo médio de resposta. Podemos ver também, que a versão *standalone* falhou em 8.28% dos pedidos, e como mostra a Figura 8, o erro é 429 Too Many Requests, significando que o servidor não teve capacidade de responder a todos os pedidos que lhe chegaram.

Em relação ao envio de ficheiros, a versão *standalone* conseguiou mostrar um maior débito, maior número de pedidos respondidos e menor tempo médio de resposta.

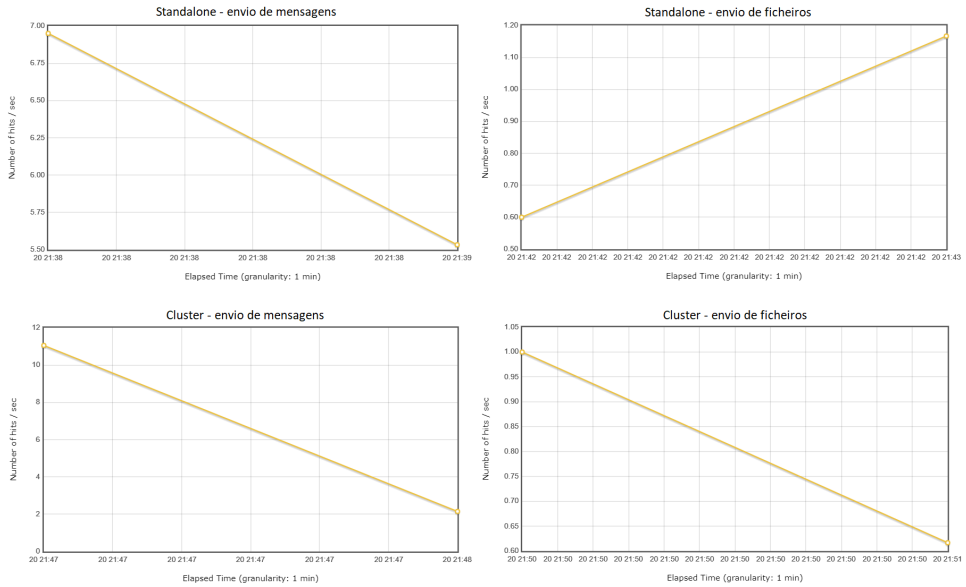

Figura 9: Comparação dos débitos

Como podemos ver pela Figura 9, todos os débitos foram diminuindo com o passar do tempo, menos na versão *standalone* no envio de ficheiros.

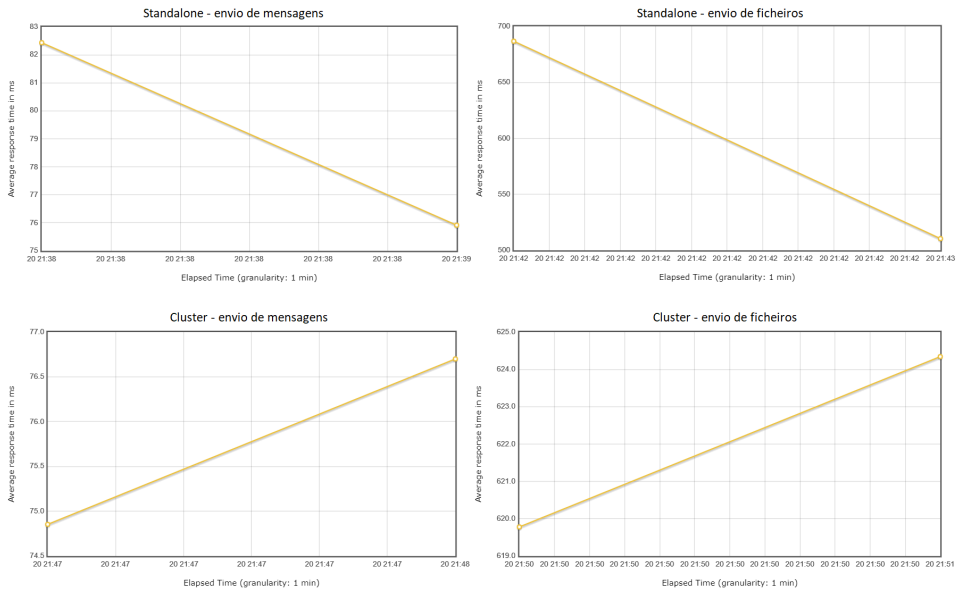

Figura 10: Comparação dos tempos de resposta ao longo do tempo

Como podemos ver pela Figura 10, os tempos de resposta na versão *standalone* foram melhorando ao longo do tempo, ao contrário do que observou na versão *cluster*.

10 Conclusão

Culminada a elaboração da primeira fase do trabalho prático, importa referir que a execução do mesmo permitiu aos elementos do grupo compreender a arquitetura e o modo de funcionamento da aplicação *Mattermost* que o grupo selecionou como alvo da sua análise.

A execução desta fase do trabalho prático permitiu uma melhor consolidação dos construtos teóricos implementados pelo *Mattermost*.

Ao nível de dificuldades sentidas, importa referir alguma complexidade sentida na adaptação à arquitetura da documentação fornecida pelos *developers*.

Esta fase tornou-se fulcral, dado que permitiu analisar e caraterizar a aplicação selecionada, tendo em vista, num futuro próximo, a fase de automatização do processo de *deployment* e de *benchmarking*.

No ímpeto geral o desenvolvimento desta fase do trabalho decorreu como planeado, alcançando os objetivos delineados pelo enunciado.

No que toca à segunda fase do trabalho, que consistiu no desenvolvimento de uma solução capaz de instalar automaticamente a *Mattermost*, assim como os componentes relativos à sua monitorização e avaliação, considera-se que esta foi bem sucedida. Através do *Ansible*, foi possível automatizar o aprovisionamento das máquinas onde os diversos componentes seriam instalados, e o uso de *Docker containers* permitiu que estes executassem de forma isolada e que fossem facilmente replicados noutras máquinas.

Dito isto, existem questões que podem ser alvo de melhorias. As *passwords* de acesso à base de dados para a sua configuração, por exemplo, são utilizadas em texto limpo. Além disso, não foram utilizadas todas as capacidades do *Ansible*, nomeadamente os *handlers* ou *templates*. Por fim, não foi possível tirar partido do mecanismo de replicação de bases de dados interno ao *Mattermost*.

Referências

[1] https://docs.mattermost.com/overview/architecture.html#communication-protocols

[2] https://docs.mattermost.com/overview/architecture.html#high-availability-and-scalability

[3] https://docs.mattermost.com/deployment/deployment.html#mattermost-server

[4] https://docs.mattermost.com/deployment/deployment.html#data-stores

[5] https://docs.mattermost.com/deployment/deployment.html#file-store

[6] https://docs.mattermost.com/deployment/deployment.html#communication-protocols

[7] https://docs.mattermost.com/administration/config-settings.html

[8] https://www.ansible.com/

[9] https://www.docker.com