Universidade de Aveiro

Gestão de Infraestruturas de Computação

Relatório

Operacionalização de um Produto

Autor :
84921 Rafael Direito

Professor :
João Paulo Barraca

12 de Julho de 2020

Conteúdo

Lista de Figuras

Lista de trechos de código

1 Introdução

Inserido no plano curricular da unidade corricular de Gestão de Infraestruturas de Computação, do curso de Mestrado em Engenharia Informática, da Universidade de Aveiro e lecionada pelo professor João Paulo Barraca, este relatório é proveniente da execução do projeto da época de recurso da disciplina.

Este projeto consiste na operacionalização, disponibilização e manutenção de um serviço de *Misago* - um fórum comunitário. Para tal, é necessário disponibilizar a solução através de um *Docker Swarm*, bem como implementar diversos mecanismos de replicação, armazenamento, balanceamento de carga, rede, virtualização, entre outros...

Para além disto, é também necessário efetuar a monitorização de todo o sistema e construir um conjunto de alarmes, definidos de acordo com o *SLA* estabelecido.

Por fim, é necessário, com base na ativação de alarmes, tomar medidas, como o aprovisionamento e desaprovisionamento de recurso.

Neste relatório podem ser observadas as estratégias aplicadas para a instalação do sistema, distribuição de carga, redundância, monitorização, alarmística e recuperação de falhas.

Todos os ficheiros de configuração e todo o código desenvolvido podem ser encontrados em :

`https://github.com/rafadireito/Misago-Deployment-GIC`

2 Produto a fornecer

Para a execução deste projeto, decidi colocar em produção a plataforma *Misago*. Esta define-se como sendo um fórum de debate de diversos temas, sendo que os utilizadores podem realizar diversas operações: iniciar uma *thread*, responder a *threads*, subscrever *threads* e favoritar as mesmas. Estas *threads* podem estar relacionadas com diferentes categorias criadas pelos administradores do fórum.

Esta plataforma foi lançada em 2018 e conta com apenas 1 *mantainer*, pelo que ainda não implementa *features* de interesse para a maioria dos utilizadores.

No que toca às tecnologias utilizadas, recorre ao Django (onde assenta toda a plataforma), ao PostgreSQL (como base de dados) e ao Redis (como apoio para o tratamento de tarefas assíncronas).

2.1 Cenário de utilização

De forma a definir uma base de utilização para o produto escolhido, é necessário estabelecer um cenário de utilização que forneça algum contexto à operacionalização e utilização do Fórum *Misago*.

Assim, tendo como base o elevado sucesso do StackOverflow[4] , defini que o produto escolhido seria aplicado como um fórum comunitário, onde os alunos do DETI possam tirar dúvidas relacionadas com programação. Desta forma, defini as seguintes características para o cenário escolhido:

- O fórum será utilizado por 1500 utilizadores;
- Os utilizadores irão executar as seguintes operações: ler *threads*, escrever *threads*, responder a *threads*, subscrever *threads*, consultar as novas *threads* do fórum, listar as *threads* que subscreveram e listar as *threads* que escreveram.
- Cada utilizador irá executar uma operação (leitura, escrita, listagem, ...) de 30 em 30 segundos.
- 15% dos utilizadores acedem ao fórum, autenticando-se. Isto é, fazem *log in* na plataforma.

Uma vez que a plataforma escolhida irá ser maioritariamente para o esclarecimento de dúvidas de programação, prevê-se que a maioria dos *posts* contenham apenas informação textual.

Relativamente aos acessos ao fórum, defini uma previsão para o comportamento do utilizador.

Sempre que um utilizador acede a um *URL* do domínio do nosso produto existe:

- 75% de probabilidade, de este acesso ser relativo à consulta de uma *thread*;
- 7% de probabilidade, do utilizador estar a responder a uma *thread*;
- 5% de probabilidade, do utilizador estar a escrever uma nova *thread*;
- 3% de probabilidade, do utilizador estar a subscrever uma *thread*;
- 3% de probabilidade, do utilizador listar as *threads* que subscreveu;
- 3% de probabilidade, do utilizador listar as *threads* do fórum que ainda não forma lidas;
- 3% de probabilidade, do utilizador listar as novas *threads* do fórum;
- 1% de probabilidade, do utilizador listar as *threads* que escreveu.

Este cenário permite modular o comportamento dos utilizadores, pelo que será a base para todos os testes de *benchmarking* executados, bem como para qualquer outra situação que envolva uma simulação do uso da plataforma.

2.2 Operação *out of the box* do produto

Tal como mencionado anteriormente, sendo o produto mantido apenas por 1 *developer*, este não apresentava quaisquer mecanismos de operacionalização para um cenário de produção. O produto consistia apenas numa aplicação *Django*, com um *Dockerfile* básico, que contruia uma imagem da aplicação, e um ficheiro *docker-compose* que lançava um *conatainer* de *Postgres* e outro de *Redis*. Relativamente à aplicação, esta era executada com *DEBUG=TRUE* e através do comando *python manage.py runserver*, pelo que, claramente, não poderia ser posta em produção desta forma. Para além dos riscos no que toca à segurança, os *request* eram processados por ordem de chegada, sem qualquer concorrência, pelo que o *throughput* do sistema era muito baixo. Obviamente que, dado este cenário, também não existia nenhum serviço para fornecer ficheiros estáticos.

Tendo em conta a situação encontrada, os primeiros passos foram imediatamente utilizar um *Gunicorn* para servir a aplicação, utilizar *NGINX* para servir os ficheiros estáticos e atuar como *reverse proxy* e criar uma melhor imagem *docker* da aplicação.

3 Estratégia adotada

Nesta secção descrevem-se as estratégias adotadas para operacionalizar a plataforma *Misago*. Estas incluem a definição de mecanismos de distribuição de carga, quer ao público, quer internamente, entre os componentes do sistema, mecanismos de redundância entre componentes e estratégias de monitorização,alarmística e de recuperação de falhas.

Nesta Secção é também apresentado o *Service Layer Agreement* (*SLA*), que define as métricas e regras de operação para o sistema em causa.

3.1 Considerações base

Relativamente à operacionalização do sistema escolhido, há que considerar uma restrição imposta pelo regente da disciplina - o sistema terá de ser operacionalizado recorrendo a um *swarm* de *docker*.

A utilização deste *swarm* de *docker*, permite-nos definir diferentes serviços para cada componente do sistema, sendo que este serviçoes estarão encapsulados dentro de um *docker container* e serão distribuídos pelos diferentes *nodes* do *swarm*, sendo que cada node tem acesso a 2 VCPUs e 4 GB de RAM. Esta abordagem permite uma maior simplicidade na configuração dos ambientes onde serão *deployed* os componentes do sistema, bem como relativamente à configuração dos próprios componentes. Outra vantagem desta abordagem advém do facto de o *docker swarm* já ter implementados mecanismos de *load balancing* internos, bem como mecanismos de replicação dos diferentes serviços criados. Desta forma, facilmente podemos aumentar, ou diminuir, o número de réplicas de um serviço, sendo que *docker swarm* irá fazer o balanceamento de carga entre estas. Caso queiramos um balanceamento de carga mais fino, este também pode ser atingido através de configurações manuais/programáticas nos diferentes serviços.

Tal como mencionado na Secção 2.2, tendo em conta que o sistema escolhido ainda está configurado para uma ambiente de *desenvolvimento*, foi necessário realizar diversas operações para que este pudesse ser posto em produção. Primeiro que tudo, foi necessário configurar um servidor *Gunicorn* - um servidor *python* que serve pedidos *HTTP* através de *WSGI* (*Web Server Gateway Interface*). Com esta alteração já efetuada, a aplicação django deixa de ser iniciada com *python manage.py runserver* e passamos a usar o *Gunicorn* para correr a aplicação: *gunicorn -b 0.0.0.0:80 devproject.wsgi:application*. O *Gunicorn* tem ainda a vantagem de nos permitir definir quantos *workers* queremos lançar. Contudo, o *Gunicorn* não serve ficheiros estáticos, pelo que precisamos de outro componente para realizar esta ação.

Assim, decidi utilizar NGINX para servir os ficheiros estáticos. A necessidade

de recorrer a este servidor surge pois o *Django*, quando em produção, divide os recursos estáticos dos dinâmicos, delegando os recursos estáticos para outro serviço, neste caso o NGINX que irei instanciar. Com este servidor, facilmente configuramos mecanismos de *caching*, pelo que irei definir este tipo de mecanismos para os recursos estáticos, permitindo aliviar carga do servidor aplicacional. Por fim, o NGINX servirá também de *reverse-proxy*, sendo que será através deste que os clientes irão comunicar com todo o sistema. A utilização deste *reverse-proxy* permite que possamos adiconar novos serviços, bem como remover serviços, sem que estas alterações sejam visíveis para os clientes do sistema.

3.2 Estratégias de *deployment*, distribuição de carga e redundância

Considerando que, relativamente à aplicação *Django* já foram configurados os mecanismos referidos na secção 3.1, falta ainda criar uma imagem *docker* com a aplicação do *Misago*. Para tal, foi necessário criar um *Dockerfile* (tendo como base o Dockerfile da versão de desenvolvimento, que se pode encontrar no GitHub do Misago) e definir todo o *entrypoint* desta imagem. Relativamente ao *Dockerfile* este recorre a uma imagem base de *python 3.7* e tem mecanismos para instalar todos os requisitos para correr a aplicação do *Misago*. No que toca ao *entrypoint*, que será invocado sempre que inciarmos um *container* ccom esta imagem, este contém o seguinte conjunto de instruções:

- Após conexão com a base de dados, irá criar o *schema* necessário para a operação do *Misago*. Caso a base de dados já esteja populada, este passo, não irá ter repercussões negativas;
- Lança os mecanismos para o tratamento assíncrono de tarefas;
- Lança o servidor de *Gunicorn* que irá servir os pedidos dos clientes.
- Lança um coletor de métricas *Telegraf*, que vai enviar as métricas para a *InfluxDB*

No que toca ao sevidor de *NGINX*, tal como mencionado anteriormente, este irá ser responsável por:

- Servir todos os conteúdos estáticos;
- Definir os mecanismos de *caching* associados aos ficheiros estáticos;
- Servir de *reverse-proxy*, sendo através deste que os clientes acedem ao sistema.

Relativamente à redundância destes componentes, ter-se-ão 3 réplicas do servidor de NGINX, definidas através do *docker swarm*, sendo que será este ambiente que será responsável por balancear os pedidos dos utilizadores entre as diferentes réplicas. Isto permite-nos uma abstração da localização real das réplicas, sendo que os clientes comunicam apenas com 1 *IP*. Este implementação ao nível do *docker swarm*, permite facilmente escalar ou desaprovisionar o sistema, uma vez que facilmente aumentamos ou diminuimos o número de réplicas de um componente, sem que isto seja visível para o cliente, ou resulte em *down-time* do sistema.

No que toca ao servidor aplicacional do *Misago*, também este terá 3 réplicas distintas. Estas 3 réplicas permitirão processar muitos mais dados, do que num cenário em que apenas existe um servidor. Estas comunicam com as intâncias de NGINX referidas anteriormente, sendo que o balanceamento de carga pode ser delegado para o *swarm*. De forma a permitir uma mais fácil utilização dos mecanismos de *scaling* do *docker swarm*, e uma vez que o sistema que estou a colocar é 100% *sateless*, pelo que não levanta problemas na área do balanceamento de carga, decidi optar por delegar a distribuição de carga para o *swarm*. Mais tarde, esta decisão permitirá um *scaling* do sistema muito mais simples.

Para além disto, de forma a assegurar a elevada disponibilidade de todos os componentes, estes serão lançados com a política de *restart* a *on-failure*. Este mecanismo, definido no ficheiro *yaml* que contém a descrição da *stack*, permite que, sempre que um *container* termine a sua execução com um código de erro, seja reiniciado de forma automática.

Do ponto de vista da persistência de dados, o criador do *Misago* indica que deve ser utilizada uma base de dados *PostgreSQL*. Contudo, uma vez que existe uma abstração dos dados ao nível do *Django*, que inicialmente irá criar todo os *schemas* da base de dados, poderia ter optado por outra opção. Contudo, após ter estudado detalhadamente *PostgreSQL* durante a execução do primeiro projeto desta disciplina e tendo já alguma prática na orquestração de serviços que utilizam esta base de dados, decidi seguir a sugestão do criador do Misago.

Relativamente à base de dados, surgem 2 aspectos críticos:

- Redundância de dados: de forma a não perdermos dados, caso exista um falha num disco. E de forma a não termos down-time no acesso à base de dados, uma vez que podemos aceder a outra instância da mesma;
- Balanceamento dos *requests*: uma vez que o *Misago* é um fórum comunitário, a grande maioria dos pedidos dos utilizadores irão ter de consultar a base de dados para obterem informação de uma *thread*, ou mesmo para escrevem uma nova publicação, pelo que é necessário fazer um correto balanceamento

de pedidos.

Relativamente ao balanceamento dos pedidos à base de dados, é impreterível que tenhamos diversas réplicas de onde possamos ler dados. Isto pois a maioria das ações tomadas pelos utilizadores serão de consulta de *threads*/publicações e não de escrita das mesmas. Esta restrição pode ser resolvida facilmente com um arquitetura *master-slave*. Assim, irei definir um *cluster* com 1 *master* e 2 *slaves*. Esta decisão leva a que tenhamos 1 instância de *PostgreSQL* a lidar com a escrita de novos dados e 3 instâncias (2 *slaves* e o próprio *master*) que podem lidar com a leitura de dados.

Do ponto de vista de replicação de dados, há que assegurar que os dados escritos no *master* são replicados nos *slaves*, pelo que decidi utilizar o mecanismo de *Streaming Replication*, que permite distribuir, de forma continua, os *records* do *WAL* do *master* pelos *slaves* do *cluster*.

Para além disto, recorri ao *repmgr* - um *manager* de replicação - para gerir a comunicação entre os nós do *cluster*, bem como os mecanismos de replicação. Este mecanismo amplifica as capacidades nativas do *PostgreSQL* de *hot-standby*, permitindo a execucação de tarefas de recuperação de falhas, por exemplo. Para além disto, o *repmgr* é capaz de lidar com mecanismos de *Cascading Replication*. Com estes mecanismos, os *slaves* conseguem atualizar outros *salves*, diminuindo a carga exercida ao nível do *master* do cluster. Por exemplo, se o *master* atualizar o *slave 1*, então, o *slave 1* poderá atualizar o *slave 2*, evitando que a atualização deste seja feita pelo master.

Contudo, existe um outro problema. Se o *master* do *cluster* morrer, será eleito um novo *master*, contudo os serviços que acedem à base de dados não sabem a localização do serviço que assumiu a liderança do cluster. Para além disto, apesar de termos definido 3 instâncias de PostgreSQL de onde os servidores aplicacionais podem ler dados, não definimos qualquer mecanismo para que a aplicação do Misago consiga localizar estas instâncias.

Estes problemas levaram-me a utilizar o *Pgpool-II* - um *middleware* que atua entre os clientes da base de dados e o *cluster* onde estão armazenados os dados.

Este *middleware* oferece um elevado número que *features* bastante vantajosas, que permitem resolver os problemas apontados anteriormente:

- *Load balancing*: O *Pgpool-II* tem mecanismos de balanceamento de carga, que permitem distribuir as *queries* pelos vários nós do *cluster*. Quando existe um elevado número de utilizadores, esta *feature* é bastante vantajosa.
- *Queries* paralelas: Este mecanimos permite a divisão de *queries* muito exigen-

tes pelos vários nós do cluster.

- Limitação do número de conexões: nativamente, por *default*, o *PostgreSQL* tem um limite máximo de 100 conexões, em paralelo. A partir deste número de conexões, iremos obter um erro quando nos tentamos ligar à base de dados. Caso queiramos aumentar o número máximo de conexões teremos de lidar com perdas no *throughput* da base de dados. Com o *Pgpool-II*, podemos evitar que, assim que o número máximo de conexões seja atingido, seja retornado um erro, uma vez que este criam uma *queue* onde guarda as conexões extra. Apesar desta *feature*, neste momento, não ser preponderante na nossa solução, achamos que é relevante referir a sua existência.

Assim, decidi que a minha solução iria dispor de 3 réplicas do componente *Pgpool-II*.

Por fim, faltam as decisões relacionadas com os servidores *Redis*.

Uma vez que o Misago utiliza *Redis* para armazenar dados para tarefas assíncronas, é necessário, também, garantir a elevada disponilidade deste componente. Assim, decidi implementar um *cluster* de *Redis* com 2 *slaves* e 1 *master*. Neste *cluster* as escritar e leituras vão ser dirigidas ao *master*, sendo que os *slaves* irão receber continuamente os dados que forem escritos no *master*, de forma a assegurar a replicação dos mesmos. Desta forma, se por algum motivo, o *master* estiver *down*, podemos facilmente eleger um dos *slaves* a *master*, uma vez que estes tem as mesmas informações que o *master* anterior.

Contudo, o mecanismo de eleição de um novo *master* não é feito automaticamente pelos elementos do cluster. É necessário utilizar um componente adicional - o Redis Sentinel [1]. Este componente oferece as seguinte *features*:

- Monitorização: verifica se os *masters* e *slaves* estão a funcionar corretamente;
- *Failover* automático: se um *master* falhar, o *Sentinel* rapidamente elege um dos *slaves* como *master*;
- Fornecedor de configurações: os clientes conectam-se ao *Sentinel* para saber a localização do *master*, onde irão invocar operações de escrita, algo que não podem fazer ao nível dos *slaves*.

Este componente adicional permite que a disponibildiade dos serviços oferecidos pelo *Redis* seja bastante elevada. Para além disto, a sua configuração também é bastante simples, o que simplifica o *deployment* da solução.

Uma vez que o *Redis Sentinel* utiliza um protocolo de consenso para definir

o estado de cada serviço *Redis*, bem como eleger um novo *master*, é necessário ter várias réplicas deste serviço. Após alguma investigação, optei por lançar 3 instâncias de *Redis Sentinel* para monitorizarem e lidarem com as falhas do *cluster*. Para assegurar a disponibilidade destes componentes, estes são lançados no *swarm* de *docker* com uma política de *restart on-failure*.

Tal como mencionado anteriormente, o *Redis Sentinel* oferece mecanismos dee descoberta do *master*, através da invocação do comando *SENTINEL get-master-addr-by-name <nome do master>*. Contudo, estarem 3 servidores aplicacionais a perguntar constantemente qual a localização do *master* do *cluster* não é, de todo, a melhor abordagem. Após investigar soluções possíveis [2, 3], descobri que uma solução commumente utilizada é colocar uma *HAProxy - load balancer* de elevada *performance* - como *reverse-proxy* para o *cluster* de *Redis*.

O *HAProxy* foi configurado para, periodicamente, consultar os *Redis Sentinel* e atualizar as informações que tem acerca do *cluster*, permitindo que se adapte sempre que existir alguma modificação no *cluster*. Os clientes irão aceder ao *Redis* através deste *load balancer*, pelo que qualquer alteração no *cluster* será completamente transparente para estes.

De forma a garantir a redundância deste componente, irei lançar 2 réplicas do mesmo, geridas pelo *swarm*. Neste cenário, será efetivamente o *HAProxy* a balancear a carga entre os diferentes componentes de *Redis*, e não o *swarm*, como foi decidido no caso do balanceamento dos pedidos para o servidor aplicacional.

Na Figura 1 podemos observar o desenho da arquitetura da solução. Nesta, as setas a vermelho indicam a monitorização dos *Redis Sentinel* sobre os servidores *Redis* e as setas a amarelo representam a replicação de dados entre instâncias *PostgreSQL* e *Redis*.

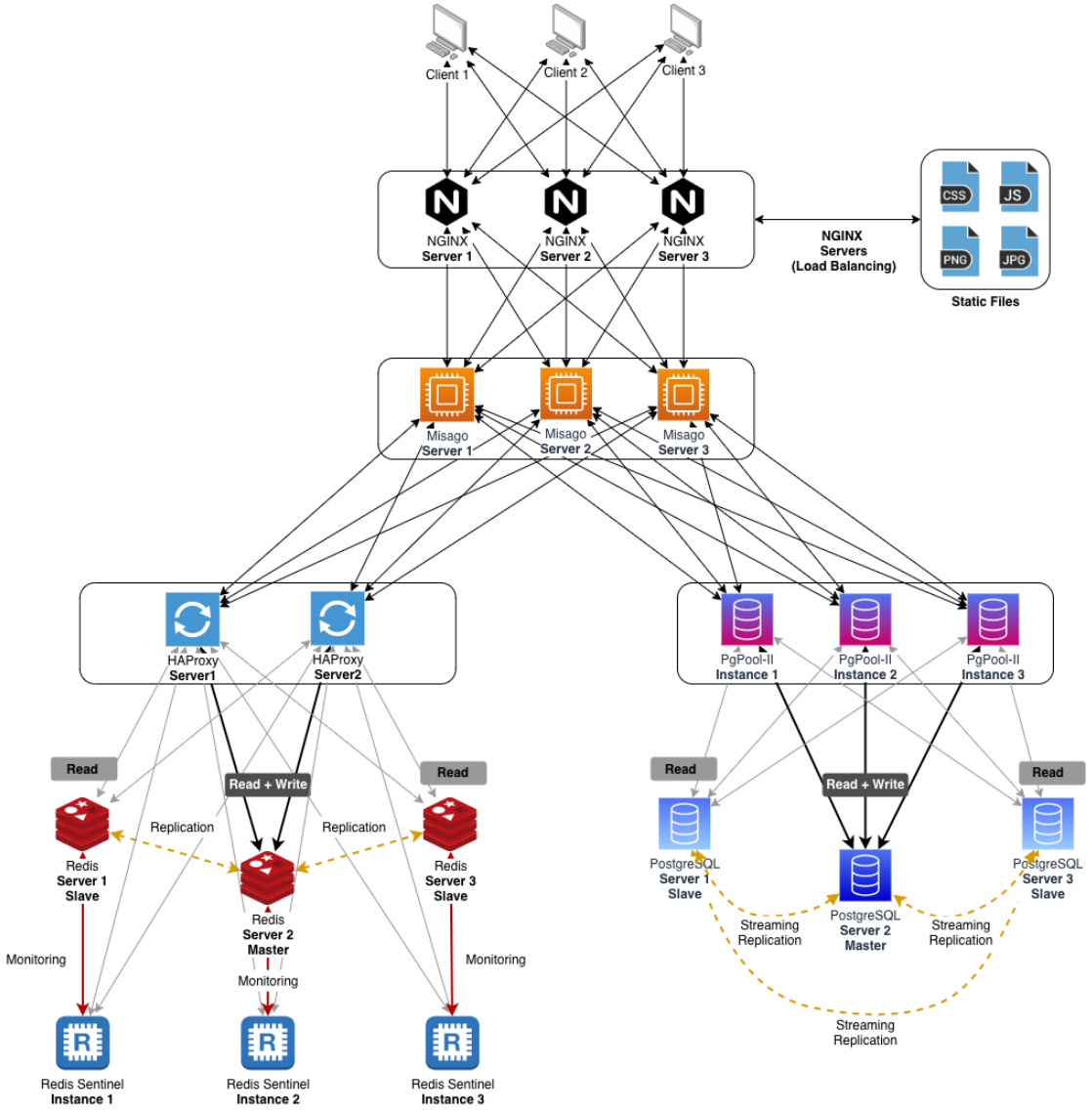

Figura 1: Arquitetura de *deployment*

3.3 *Service Layer Agreement* a fornecer

Uma vez que o âmbito deste projeto é a operacionalização de um produto, num cenário específico, surge a necessidade de estruturar um *SLA (Service Layer Agreement)* - um acordo entre a entidade que operacionaliza o produto e o cliente. Este acordo define um conjunto de regras, através das quais se vai reger a operacionalização do produto, bem como quais os mecanismos que devem ser aplicados caso a entidade prestadora de serviço não cumpra com o acordo estabelecido. Estes mecanismos não serão abordados no presente relatório pois não fazem inteiramente parte dos objetivos desta unidade curricular.

No que toca às regras do acordo, os *SLAs* são constituídos por um conjunto de *SLOs (Service Level Objects)* - as métricas que devem ser monitorizadas, associados a *SLIs (Service Level Indications)*, que se definem por ser o *thresholds* ou as funções que deverão ser aplicadas sobre os *SLOs*.

Segue-se o acordo definido:

1. Considerações base

- A entidade prestadora de serviço (EPS) não se responsabiliza por quaisquer falhas internas do Misago. Apenas detém responsabilidade sobre a operacionalização (instalação, *deployment* e monitorização) do produto;
- O autor do produto a ser disponibilizado afirma que foi efetuado muito pouco trabalho no que toca à eficiência do produto, pelo que este poderá não estar num estado indicado para ser operacionalizado com sucesso.
- A EPS não é responsável pela gestão do *hardware* onde está instalado o produto. Contudo, a EPS tem acesso a uma *pool* de elementos de *hardware*, fornecidos por um *third-party*, pelo que deve assegurar a operacionalização do produto, sempre que existir *hardware* disponível nesta *pool*;
- O serviço apenas será disponibilizado aos utilizadores que estiverem dentro da *VPN* indicada pelo cliente;
- A EPS assegura atendimento telefónico personalizado 24h por dia, durante todos os dias do ano;
- Qualquer pedido, por parte do cliente, que envolva a mudança da estrutura base de operacionalização, deve ser disccutido com a EPS, sendo que se chegaraá a um novo *SLA*, se tal for justificavel.

2. *SLOs* e respetivas *SLIs*

Os valores da Tabela 1 são relativos à operacionalização do Misago inserido no cenário de utilização definido na Secção 2.1 deste documento - 1500 utilizadores, que efetuam um *request* de 30 em 30 segundos.

Note-se que todos os *SLOs* expressos na Tabela 1 que não têm nenhuma *SLI* associada, são métricas que são monitorizadas, contudo, não existem *thresholds* definidos sobre as mesmas.

No seu todo, o serviço deve garantir que:

- O tempo de resposta a um GET, deverá ser inferior a 40 segundos;
- O tempo de resposta a um POST/PACH, deverá ser inferior a 50 segundos;
- O serviço tem de oferecer uma taxa de erro (requests não respondidos, ou respondidos com erro) inferior a 10 %;

3.4 Estratégias de monitorização, alarmística e recuperação de falhas

A definição de mecanismos de monitorização permite ter uma visão constante do estado do nosso sistema. Estes mecanismos permitem controlar a *performance* do sistema, encontrar falhas, reagir perante estas e muito mais...

No que toca à monitorização aplicada ao serviço disponibilizado, podemos dividi-la em monitorização de métricas e monitorização de *logs* textuais, sendo que estes 2 tipos diferentes de monitorização oferecem, também, vistas diferentes sobre o sistema.

Para monitorizar o produto que coloquei em produção, recorri a uma VM externa ao sistema onde estava disponibilizado o sistema. Isto permite que, caso a infra-estrutra de produção esteja *offline*, sejamos avisados de tal ocorrência, sendo que, se a *stack* de monitorização estivesse *deployed* na mesma infra-estrutura onde está o sistema a ser monitorizado, ficaríamos totalmente sem acesso aos mecanismos de monitorização e alarmística que definimos. Para além disto, esta separação física permite aliviar a carga da infra-estrutura de produção, uma vez que as *stacks* de monitorização executam algumas operações computacionalmente exigentes.

Estes mecanismos, juntamente com alarmes definidos sobre os *logs* e métricas obtidos, irão permitir que a entidade prestadora do serviço tenha conhecimento de comportamentos erráticos do sistema, o que lhe vai permitir um melhor controlo e cumprimento do *SLA* que definiu com o seu cliente.

***SLOs* e respetivas *SLIs* por serviço**		
Serviço	***SLO***	***SLI***
NGINX	clientes por instâcia	< 1024
	requests por instâcia	< 100 req/s
	erros	< 5 erros/seg
	estado dos clientes (*writing*, *waiting*, *reading*)	-
Redis e HAProxy	erros	< 5 erros/seg s
	tempo de resposta	< 10 s
	número de servidores HA-Proxy	-
PostgreSQL	*rows* inseridas	< 200 *rows*/seg
	fetched rows	< 2000 *rows*/seg
	commits/rollbacks	-
	deadlocks/conflitos	-
Misago	tempo de resposta a um *POST*/*PATCH*	< 50 seg
	tempo de resposta a um *GET*	< 40 seg
	clientes por instâcia	< 1500
	erros	$< 7.5\%$
SNMP Docker Nodes	carga do CPU	$< 70\%$
	utilização da RAM	$< 70\%$
	utilização do disco	$< 80\%$
	upTime	-
	número de VCPUs	-
	memória total	-
	disco total	-
	pacotes recebidos	-
	pacotes enviados	-

Tabela 1: *SLOs* e *SLIs* por serviço

3.4.1 Estratégia de obtenção e monitorização de métricas

Relativamente à obtenção de métricas, após alguma investigação, e tendo como base algum trabalho efetuado no passado, decidi escolher a *TICK Stack* para recolher, armazenar e visualizar as métricas obtidas. Esta *stack* é composta pelos seguintes conponentes:

- Telegraf - um agente que reccolhe métricas e envia-as para uma *time series database*;
- InfluxDB - a *time series database* onde serão armazenadas as métricas recolhidas;
- Chronograf - uma interface de visualização de dados, que permite criar diferentes *dashboards* para as métricas que estão armazenadas na *InfluxDB*;
- Kapacitor - um *data processing engine* permite a criação de alarmísticas.

No que toca aos mecanismos de envio de métricas, será da responsabilidade de cada componente enviar as suas métricas para a *VM* de monitorização, sendo que, se esta não receber quaisquer métricas irá assumir que o componente está *offline*.

A monitorização das métricas de cada componente, será, na maioria do casos, bastante simples de efetuar, uma vez que o *Telegraf* tem diversos *plugins* que permitem a recolha das métricas que o utilizador deseja. Contudo, o mecanismo de recolha de métricas do *hardware* onde está assente cada componente é implementado de uma forma mais complexa.

Como os componentes não conseguem, por questões de virtualização, obter informações relativas ao *docker node* onde estão a 'correr', é necessário consultar o *docker* para obter a localização física de cada componente, e, posteriormente, coletar as métricas desse *docker node*. Estas métricas serão obtidas com recurso ao protocolo *SNMP*, que permite a obtenção de diversas m étricas relacionadas com o *hardwar* (*CPU*, *RAM*, informações acerca dos discos, etc).

3.4.2 Estratégia de obtenção e monitorização de *logs* textuais

No que toca à obtenção de *logs* textuais, estes devem ser armazenados num repositório central, de forma a que se possam obter importantes informações do sistema através destes. Devemos, ao nível deste repositório central, ter mecanismos para extrair informação dos *logs*, bem como execcutar diversas *queries* sobre estes.

Para monitorizar os *logs* textuais do sistema, decidi utilizar a *ELK Stack*. Esta é composta pelos seguintes componentes:

- Elasticsearch - um poderoso motor de busca, baseado em JSON, que é responsável por indexar os *logs* textuais;
- Logstash - uma *pipeline* que permite o *parsing* dos *logs*, resultando num maior conjunto de informações sobre cada componente, uma vez que depois é possível fazer diversas *queries* recorrendo aos campos que foram *parsed*;
- Kibana - uma interface de visualização de dados, que permite criar diferentes *dashboards* para a visualização da informação obtida dos *logs* textuais.

Relativamente ao envio dos *logs* textuais para esta *stack* de monitorização, este foi efetuado com recurso ao *Filebeat* e ao *Syslog*, que é um mecanismo já suportado pelo *docker swarm*, o que simplifica o envio dos *logs*.

No que toca à extração de conhecimento a partir de *logs* textuais, a utilização do *Filebeat* traz claras vantagens, face à utilização da *driver* de *Syslog*. Isto, pois o *Filebeat* é composto por um conjunto de módulos que exportam apenas a informação à qual se refere este módulo. Desta forma, já não é tão essencial que os *logs* textuais sejam *parsed* ao nível do *Logstash*.

Se utilizarmos o *Syslog*, é ideal que enviemos os *logs* para o *Logstash*, que façamos a extração de conhecimento e, só após isto os indexemos. Esta extração de conhecimento pode ser efetuada através da utilização do *Grok*, que gera dados estruturados e *queryable*, a partir de dados não estruturados. Para tal, recorre a um conjunto de *regexs*.

Uma vez que, não existe nenhum módulo de *Filebeat* que suporte a extração de conhecimento de *logs* de *Gunicorn* - o servidor que utilizei para servir o Misago - é necesssário configurar uma template de logging e definir uma regex sobre ela, que possa ser aplicada ao nível do *Logstash*.

Falta referir que, por omissão, a *driver* de *syslog* do *docker swarm* apenas captura os *logs* textuais do *stdout*, pelo que foi necessário efetuar alterações ao nível das imagens *docker*, alterando o destino dos *logs*.

Depois dos *logs* serem *parsed*, estes são enviados para o *Elasticsearch*, que os indexará.

3.4.3 Estratégia para a alarmística

De forma a que os adminsitradores do Misago estjam constantemente informados acerca de potenciais falhas/problemas, é essencial que se estabeleça um mecanismo de alarmística, que notifique as partes interessadas, sempre que ocorre um determinado evento. Estes mecanismos de alarmística serão uma preciosa ajuda para a manutenção de todas as cláusulas definidas ao nível do *SLA*.

Com vista à não violação do *SLA* estabelecido, foram definidos os alarmes presentes na Tabela 2.

Serviço	**Alarmes**
NGINX	*threshold*: nr. de clientes numa instância > 750 *threshold*: nr. de requests numa instância > 50 req/s *threshold*: nr. de erros > 2.5 req/s
Redis e HAProxy	*threshold*: nr. de erros > 2.5 req/s *threshold*: tempo de resposta > 5 s
PostgreSQL	*threshold*: nr. de *rows* inseridas > 1500 *rows*/seg *threshold*: nr. de *fetched rows* > 1500 *rows*/seg
Misago	*threshold*: tempo de resposta a um *POST*/*PATCH* > 30 seg *threshold*: tempo de resposta a um *GET* > 10 seg *threshold*: nr. de clientes por instâcia > 1000 *threshold*:% de erros > 4 %
SNMP Docker Nodes	textitthreshold:% de carga do CPU > 50% *threshold*: % de utilização da RAM > 55% *threshold*:% de utilização do disco > 70%

Tabela 2: Alarmes definidos, consoante o *SLA* estabelecido

Estes mecanismos serão estabelecidos ao nível, quer das métricas obtidas, quer dos *logs* textuais, pelo que será necessário utilizar diferente *software* para tratar cada um destes cenários.

Relativamente aos *logs* textuais, podemos recorrer aos mecanismos mode alarmística, quer do *Kibana*, quer do *Watcher* do *Elasticsearch*. Embora os mecanismos de alarmística do *Kibana* tenham sido lançados muito recentemente, na versão 7 - ainda estão em *Beta* - estes mecanismos oferecem um interface bastante siples para a criação de alarmes e a tomada de decisões consoante os mesmos. Facilmente conseguimos definir um *threshold* nos tempos de resposta do *NGINX* ou do *Misago*, sendo que conseguimos enviar notificações para diversas aplicações e mesmo enviar *POSTs* para um *webhook*.

Contudo, se pretendermos utilizar *queries* mais complexas, a melhor opção é utilizar os *advanced watchers* do *Elasticsearch*. Esta *feature* apenas está presente na versão *premium* do *Elasticsearch*, contudo, é possível experimentarmos a mesma durante 30 dias de forma gratuíta, o que permitiu que a utilizasse no decorrer deste projeto. Também esta opção permite enviar mensagens através de diversos *webhooks*.

No que toca às métricas recolhidas, podemos utilizar o *Kapacitor*, da *TICK*

Stack, como mecanismo de alarmística sobre estas. O *Kapacitor* dispõe de uma interface visual bastante apelativa e simples de utilizar, sendo que permite *queries* mais complexas, quando comparado com o *Kibana*. Ao nível do *Kapacitor* pode-se, também, definir mecanismos para o envio de mensagens para diversas aplicações e *webhooks*, bem como podemos executar comandos ao nível do *OS*, quando um alarme for ativado, o que facilita bantante a implementação de mecanismos de automação.

Caso seja necessária a criação de *queries* complexas ao nível do *Kapacitor* podemos recorrer a *TICKscripts*. Contudo, para garantir a monitorização do Misago, tal não foi necessário.

Para simular a plataforma de gestão utilizada pela equipa do Misago decidi criar um *slack workspace*. Será através do canal *alerts* que os administradores da plataforma serão notificados da ocorrência dos demais eventos.

3.4.4 Estratégia para a recuperação de falhas

Relativamente à ocorrência de falhas, há que definir mecanismos que sejam responsáveis por as detetar e recuperar das mesmas. No que toca à deteção de falhas, esta é assegurada pelos diferentes mecanismos de monitorização, abordados nas Secções 3.4.1 e 3.4.2, e pelos mecanismos de alarmística, abordados na Secção 3.4.3.

No contexto da operacionalização do *Misago*, define-se como falha qualquer violação do *SLA* estabelecido na Secção 3.3. Estas falhas podem ocorrer devido a diversos fatores. Por exemplo, podemos ter elevados tempos de resposta, ao nível da aplicação, devido ao excesso de utilizadores. Contudo, este aumento do tempo de resposta também pode estar relacionado com o facto de um dos *nodes* do *swarm*, onde estava a *deployed* um componente, ter ficado *offline*, entre outros motivos.

O *docker swarm*, por si só, já tem mecanismos para lidar com falhas ao nível dos *containers* que encapsulam cada componente da nossa *stack*, pelo que este é dos um dos mecanismos que utilizei para garantir que, caso um *container* tenha um erro, terminando a sua execução, este irá ser disponibilizado noutro *node*. Isto é garantido através do atributo *restart*, que defino no *docker-stak.yaml*. Assim, para todas os serviços da *stack* decidi utilizar o 'mecanismo' *restart:on-failure*, que define que, caso um *container docker* seja encerrado com erro, este será *redeployed* num outro *node*, de forma automática.

A existência de serviços redundantes é, também, uma mecanismo de recuperação de falhas. No sistema que implementei, todos os componentes têm, pelo menos, redundância N+1, sendo que o *HAProxy* é o único que não têm redundância N+2, como todos os outros componentes. Existem 3 instâncias do serviço *NGINX*,

3 instâncias do servidor da aplicação do *Misago*, 3 instâncias do serviço de *PgPool* e 2 instâncias de *HAProxy*.

No que toca ao armazenamento de dados, a redundância é visível na utilização de arquiteturas do tipo *master-slave*. No caso do *Postgres*, existe 1 nó *master* e 2 nós *slaves*, que regularmente trocam informação entre si, levando a que a informação esteja armazenada em 3 localizações diferentes. Neste cenário, se o nó *master* falhar, o *PgPool* será responsável por eleger um dos 2 *slaves* a líder do cluster, que continuará a operar, pelo que o utilizador não notará *downtime* do serviço. posteriormente, o agora *ex-master* voltará a ser *deployed* num novo nó do *swarm* e juntar-se-à novamente ao *cluster*. Este deploy automático deve-se ao mecanismo *restart:on-failure*, já abordado anteriormente.

Relativamente ao *Redis*, aplica-se o mesmo cenário - um *cluster* com 1 *master* e 2 *slaves*, que são constantemente observados por 3 instâncias de *Redis Sentinels*, que monitorizam o cluster e que, caso o *master* fique *offline*, elegem um novo líder através de um protocolo de consenso.

A existência de serviços 100% *stateless* ajuda, também, para a recuperação de falhas. Podemos configurar *swarm replicas* destes serviços, com poucas, ou nenhumas, configurações adicionais. Por exemplo, se o serviço estiver num pico de utilização, facilmente se utiliza a API do *docker* para aumentar o número de réplicas do servidor aplicacional, de 3 para 4, permitindo, assim, que se cumpra o *SLA* definido.

Por fim, tendo em conta o cenário de utilização deste sistema, facilmente podemos prever que a plataforma será atacada por diversos alunos que pretendam explorar as suas *skills* na área de segurança, pelo que também é importante configurar mecanismos que sejam capazes de bloquear o acesso destes utilizadores ao serviço. Por exemplo, se um utilizador tentar efetuar por diversas vezes *login*, sem utilizar credencias corretas, num curto espaço de tempo, este utilizador deverá ser bloqueado do sistema. Para tal é necessário que a infra-estrutura de *monitoring* informe a infra-estrutura de produção de tal acontecimento, o que é facilmente atingido com um pequeno *script* em *cherrypy*.

4 Instalação da Solução

Nesta cecção abordam-se os mecanismos de instalação de cada um dos componentes do sistema, considerando as questões e virtualização, armazenamento, rede, configurações e definição dos serviços.

O sistema foi instalado num *docker swarm* disponibilizado pelo docente da unidade ccurricular, pelo que todo o desenvolvimento dos diferentes componentes foi feito com base nesta restrição. Ainda assim,o sistema pode ser lançado através de *docker-compose*, *local*, sendo necessário a alteração de algumas propriedades nativas ao *docker swarm* (*secrets*, *configs*, etc)

4.1 Construção das imagens dos componentes

Sendo o sistema *deployed* num *docker swarm* foi necessário disponibilizar os diversos componentes através de imagens *docker*. As imagens utilizadas, à exceção da imagem do servidor aplicacional, que foi criada quase de raiz, tiveram como base imagens *docker* disponibilizadas por *third-parties*.

Assim sendo, foram utilizadas as seguintes imagens base:

- nginx:mainline-alpine - para os servidores *NGINX*;
- bitnami/postgresql-repmgr:11-debian-10 - para as bases de dados *PostgreSQL*, com *streaming replication*;
- bitnami/pgpool:4-debian-1 - para a *Pgpool* responsável por gerir o *cluster* de base de dados *PostgreSQL*;
- bitnami/redis:6.0-debian-10 - para a criação dos componentes *Redis*. Esta imagem foi usada, quer para os *Redis masters*, quer para os *Redis slaves*
- bitnami/redis-sentinel:6.0-debian-10: para os compnentes *Redis Sentinel*, que monitorizam o *cluster* de *Redis*
- haproxy:1.7 - para os *load balancers HAProxy*, relativos ao *cluster* de *Redis*;

Tendo em conta as estratégias de monitorização definidas na Secção 3.4, às imagens *docker* listadas acima, foi necessário adicionar os binários do *Telegraf* e do *Filebeat*, bem como fazer outras modificações ao nível da leitura de variáveis de ambiente, por exemplo.

4.2 Armazenamento

Relativamente aos mecanismos de armazenamento, recorri a *bind mounts* para o sistema de ficheiros *NFS*, que foi disponibilizado pelo docente da unidade curricular.

Neste *File System* estão alojados os dados que devem ser persistidos entre as diferentes *runs* de um *container*. Assim, persistem-se os seguintes dados:

- Ficheiros estáticos da aplicação do Misago. Estes ficheiros são gerados pela *app django* e devem estar acessíveis ao *NGINX*, que os vai servir;
- Dados e ficheiros de configuração da base de dados *PostgreSQL* pg-0;
- Dados e ficheiros de configuração da base de dados *PostgreSQL* pg-1;
- Dados e ficheiros de configuração da base de dados *PostgreSQL* pg-2;

Relativamente aos dados das diversas instâncias de *PostgreSQL*, estes são mantidos em diretórios independentes. Assim asseguramos a consistência de dados e a redundância dos mesmos, uma vez que, caso um dos diretórios seja corrompido, os dados estão, também, armazenados noutro diretório. Caso tivesse acesso a outro *File System*, alguns destes dados podiam ser movidos para lá, sendo necessário assegurar sempre que o *delay* relacionado com o acesso a estes dados não prejudicaria o serviço.

4.3 Ficheiros Estáticos

Uma vez que, ao correrrmos uma aplicação *django* com *DEBUG=False*, esta não é capaz de servir ficheiros estáticos, foi necessário configurar um mecanismo para assegurar esta funcionalidade.

Assim, tal como mencionado na Secção 3.2, decidi utilizar o *NGINX* para servir estes ficheiros estáticos.

Para tal, é necessário coletar os ficheiros estáticos da aplicção *django* - *python manage.py collectstatic* - e indicar ao *NGINX* qual a localização deles, e quais os pedidos aos quais devem ser devolvidos estes ficheiros. Isto tira, também, alguma carga do servidor aplicacional.

Foi, então, criada uma nova configuração para o *NGINX*, que permite servir ficheiros estáticos de uma forma eficiente. Para diminuir a largura de banda utilizada utiliza-se o *gzip* para comprimir os ficheiros antes de serem enviados para o cliente, embora este possa optar por receber os ficheiros não compactados. Configuraram-se, também, os *headers* dos ficheiros servidos, de forma a que estes ficassem em *cache*

durante 1 ano, de forma ao utilizador ter uma melhor (mais rápida) interação com o nosso serviço.

Segue-se a configuração utilizada para servir ficheiros estáticos:

```
location /static/ {
    gzip on;
    gzip_static on;
    gzip_disable "msie6";
    gzip_vary on;
    gzip_proxied any;
    gzip_comp_level 6;
    gzip_buffers 16 8k;
    gzip_proxied any;
    gzip_min_length 256;
    gzip_http_version 1.1;
    gzip_types application/javascript application/rss+xml
   application/vnd.ms-fontobject application/x-font application/x-
   font-opentype application/x-font-otf application/x-font-truetype
    application/x-font-ttf application/x-javascript application/
   xhtml+xml application/xml font/opentype font/otf font/ttf image/
   svg+xml image/x-icon text/css text/javascript text/plain text/
   xml;
    expires 365d;
    add_header Cache-Control "public, no-transform";
    alias /static/;
}
```

Trecho de código 1: Configurações utilizadas para servir ficheiros estáticos

4.4 *Docker Secrets*

De forma a armazenar com segurança as diferentes credenciais de acesso utilizadas no *deployment* da nossa *stack*, recorremos aos *docker secrets*. Estes não são mais que *blobs* de dados que são geridos pelo *docker swarm*, que se encarrega da sua transmissão encriptada até aos serviços que irão necessitar das credenciais que estes armazenam. Para além desta trasmissão encriptada, os *secrets* apenas estão disponíveis para um conjunto de serviços explicitamente definidos e apenas enquanto as tarefas destes serviços estão a ser executadas.

Os *secrets* são, portanto, um bom mecanismo para guardarmos, de forma segura, *usernames*, *passwords*, chaves de certificados e de *SSH*, sem que estes estejam expostos nas imagens de *docker*, ou na configuração de um serviço.

Uma vez que estes são disponibilizados aos *docker services* na forma de ficheiros, são necessários mecanismos para conseguirmos extrair o segredo, propriamente dito, destes ficheiros.

Embora muitas *docker images* já tenham este mecanismos embutidos, como é o caso de uma imagem de *PostgreSQL*, por exemplo; foi necessário configurar a imagem que contém a aplicação *django* do *Misago* de forma a adiconar esta funcionalidade. Assim, na configuração de um serviço, definem-se variáveis de ambiente com a localização do segredo pretendido e, posteriormente, no *ENTRYPOINT* da imagem, configura-se um mecanismo para carregar o segredo para uma nova variável de ambiente. Recorde-se que a instrução presente no *ENTRYPOINT* será chamada assim que se iniciar o *container*, sendo que não pode ser *overwriten* através da adição de parâmetros da *command-line* utilizados no comando *run* do *container*.

Foi, então, adiconado ao *entrypoint.sh* da aplicação *Misago* o seguinte código:

```
# Function to load secrets
file_env() {
  local var="$1"
  local fileVar="${var}_FILE"
  echo $fileVar
  fileLocation=$(printenv  $fileVar)
  if [ -f "$fileLocation" ]; then
    echo "$fileLocation exist"
    val=$(cat $fileLocation)
    export "$var"="$val"
  fi
}

# Usage
file_env 'POSTGRES_PASSWORD'
file_env 'SUPERUSER_PASSWORD'
```

Trecho de código 2: Função utilizada para carregar segredos para variáveis de ambiente

A função acima parte do pressuposto que a localização dos segredos vai ser ser passada através de varíaveis de ambiente na forma *<XXX>_FILE*. Tendo isto em conta, irá ler o conteúdo do ficheiro presente na localização descrita pela variável *<XXX>_FILE* e carregá-lo para a variável de ambiente *<XXX>*, que passará a conter o segredo que iremos necessitar para a configuração do *container*. No exemplo acima, utiliza-se este mecanismo para enviar a palavra-passe da base de dados do *Misago* para o *container* que irá correr o servidor aplicacional do mesmo, bem como para estabelecer a *password* do *superuser* da aplicação *django*.

4.5 *Docker Configs*

As *docker configs* permitem guarda informação não sensível, fora da imagem de um serviço. Este comportamtento oferece bastantes vantagens, uma vez que

permite o *decoupling* entre uma aplicação e a sua configuração.

Estas *configs* ficam armazenadas no *docker*, em base64 e, quando iniciamos um *container* que necessita de uma determinada *config*, esta é *decoded* e colocada no *target*, definido no ficheiro de *compose*.

Uma outra vantagem do uso de *docker cconfigs* advém do facto de, semopre que alterarmos uma configuração, não ser necessário voltar a construir uma imagem.

Devido a estas vantagens e à facilidade da utilização de *docker configs*, recorreu-se a este mecanismo, para importar a configuração do *NGINX* para dentro dos *containers* que executam este serviço.

Sempre que foi necessário alterar a configuração, apenas necessitei de fazer o *update* da *config* criada no *docker*.

4.6 Redes

Relativamente à comunicação entre os *containers* que contêm os nossos serviços, esta é efetuada através de redes *overlay*. Através destas redes, o *routing* de pacotes é efetuado de forma transparente, sendo que a configuração dos mecanismos de rede ao nível dos *containers* é bastante simples. Apenas é necessário indicar qual o nome do serviço a que nos queremos conectar, bem como a porta que está a servir a aplicação à qual nos queremos conectar, e o *docker*, através de *DNS*, estabelece a conexão que pretendemos.

Na Tabela 3 podem ser observadas quais as redes criadas, bem como o objetivo inerente à criação de cada uma destas.

Rede	**Tipo de Rede**	**Objetivo**
Redis Network	*overlay* interna	Comunicação entre os diferentes elementos de *Redis* e do *HAProxy*, bem como comunicação entre o servidor aplicacional do *Misago* e todos estes componentes
PostgreSQL Network	*overlay* interna	Comunicação entre as instâncias de *PostgreSQL* e as *Pgpools*, bem como comunicação entre o servidor aplicacional do *Misago* e todos estes componentes
Misago Network	*overlay* interna	Comunicação entre as instâncias de servidores *Misago* e as instâncias dos serviços *NGINX*
NGINX Network	*overlay* externa	Comunicação entre as instâncias de servidores *NGINX* e o mundo exterior

Tabela 3: Redes de Comunicação

Através da criação de apenas 1 rede externa, ao nível das instâncias *NGINX*, garante-se que os utilizadores exteriores apenas terão acesso a estes servidores, pelo que estes serão a porta de entrada de todos os *requests* ao sistema. Desta forma, um utilizador não consegue enviar um pedido diretamente para os servidores aplicacionais do *Misago*.

Na Figura 2 podemos observar as diferentes redes criadas.

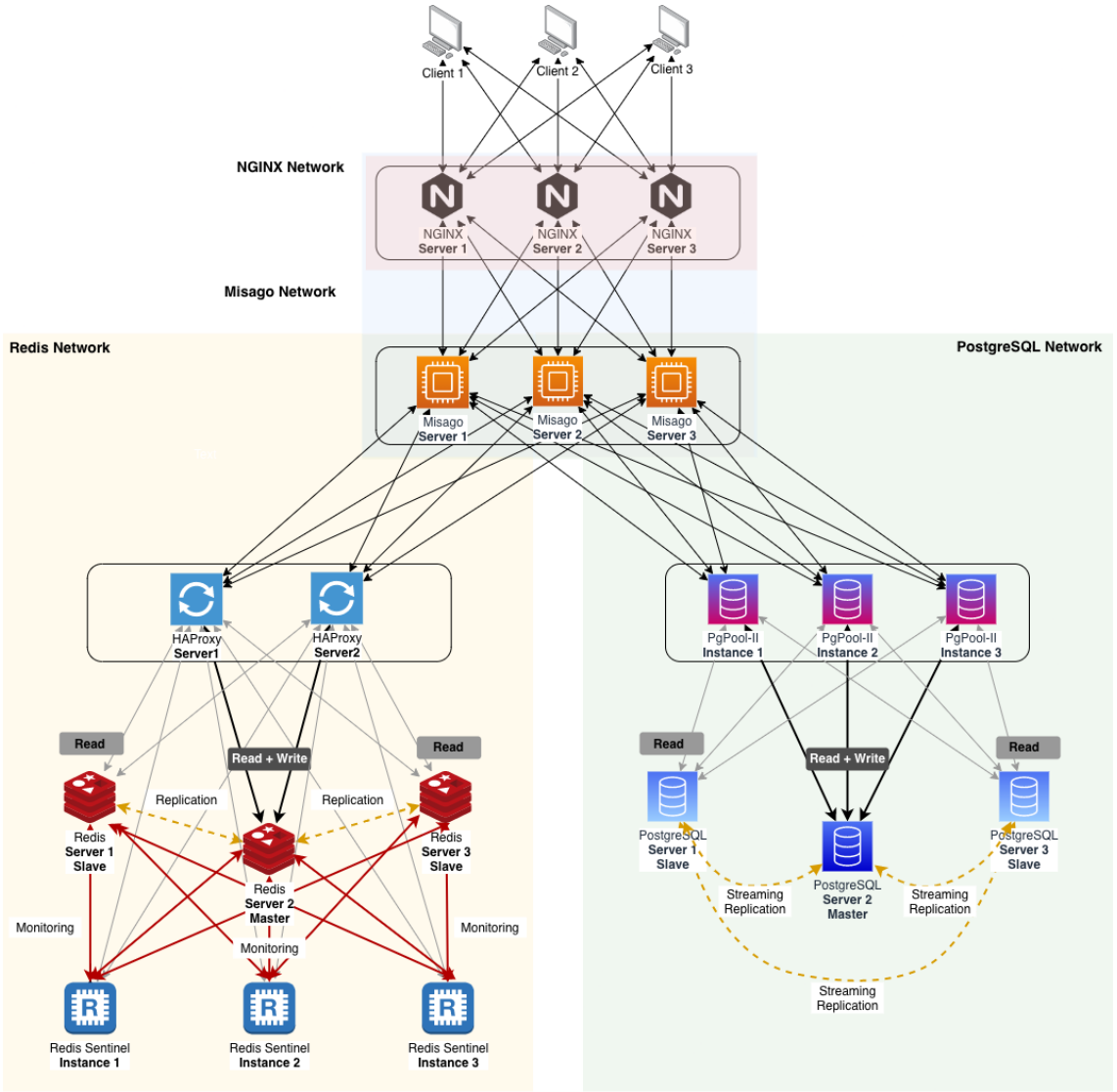

Figura 2: Redes de Comunicação

5 Operação

Tendo, na Secção 3 definido qual a estratégia para o *deployment* e operação do produto, resta descrever de que forma foram atingidas as estratégias delineadas.

Assim, ao longo desta secção, descrevem-se detalhadamente quais os mecanismo utilizados para garantir cada uma das decisões e soluções apresentadas anteriormente.

5.1 Implementação dos mecanismos de monitorização dos recursos computacionais

Esta subsecção dividive-se em 2 momentos distintos. Inicialmente abordarei quais os mecanismo utilizados para a monitorização de métricas, sendo que, posteriormente, irei abordar quais os mecanismos utilizados para a monitorização de *logs* textuais.

5.1.1 Mecanismos de monitorização de métricas

Tal como mencionado na Secção 3.4, utilizei a *TICK stack* para a obtenção, armazenamento e visualização das métricas recolhidas.

Inicialmente foi necessário instalar todos os compenentes desta *stack* na *VM* de monitorização. Durante esta fase, foram encontrados alguns problemas com as versões dos componentes. A versão da InfluxDB que está presente nos repositórios utilizados pelo *Ubuntu*, por exemplo, tem um *bug* que afeta o reconhecimento de algumas das *tags* das métricas que são armazenadas esta, não conseguindo disponibilizar qual o *TAG-VALUE* relativo ao *host* de onde foi enviada uma métrica. Apesar destes percalços, considero que a instalação da *TICK Stack* é bastante simples, quando comparada a outros mecanismos de monitorização.

Tendo a *TICK Stack* instalada, foi necessário configurar cada imagem *docker* dos componentes do produto escolhido para que utilizassem o *Telegraf* para a obtenção e o envio das métricas. Assim, foi necessário alterar o *Dockerfile* de cada imagem para que fossem transferidos os binários e as configurações do *Telegraf* (ver Trecho de Código 3).

```
1 # Telegraf
2 COPY telegraf/telegraf  /usr/bin/
3 RUN chmod +x /usr/bin/telegraf
```

Trecho de código 3: Transferência dos binários do *telegraf* para uma imagem *docker*

Posteriormente, foi necessário alterar os *entrypoints* para 'correr' o *Telegraf*, de forma a este recolher e enviar métricas para a *InfluxDB*. Durante esta fase, foi implementado um mecanismo para a identificação dos vários componentes, uma vez que não sería viável utilizar apenas o nome do serviço como chave de um conjunto de métricas. Isto, pois, ao replicarmos serviços através da *API* do *docker*, este seriam todos reconhecidos como apenas um serviço, ao nível da *InfluxDB*.

Assim, decidi utilizar o IP de cada serviço, em conjunto com o tipo de serviço, como identificador único. Para tal, alterei os *entrypoints* para que, assim que um container fosse lançado, este guardasse numa variável de ambiente o seu identificador único (ver Trecho de Código 4). Aqui, a variável de ambiente *HOST_MONITORING_TAG* é definida ao nível do *docker-swarm.yaml*.

```
# save host id as an environment variable
export HOST_ID=$HOST_MONITORING_BASE_TAG$(awk 'END{print $1}' /etc/
    hosts)

# Run metric collector - telegraf
eval "/usr/bin/telegraf --config /telegraf.conf &"
```

Trecho de código 4: Identificação de cada componente do sistema

Posteriomente, o *Telegraf* associa o *host* de um conjunto de métricas, a este *HOST_ID*.

A configuração do *Telegraf* foi desenvolvida com o objetivo de ser o mais modular possível, pelo que recorre a variáveis de ambiente que indicam qual a localização da *InfluxDB* e qual o nome da base de dados ondem devem ser armazenadas as métricas (ver Trecho de Código 5).

```
services:
  misago:
    ...
    environment:
      ...
      # MONITORING - TELEGRAF
      - INFLUX_DB_LOCATION=http://10.5.0.109:8086
      - INFLUX_DB_NAME=telegraf
      - HOST_MONITORING_BASE_TAG=misago-app-
      # MONITORING - ELK
      - ELASTIC_SEARCH_HOSTS=["10.5.0.109:9200"]
```

Trecho de código 5: Variáveis de ambiente do *docker-stack.yaml* definidas para o envio de métricas

Durante a configuração do *Telegraf* encontrei alguns percalços nas imagens *bitnami*, que correm como *non-root*, pelo que foi necessário utilizar o *chown* para definir o *user non-root* como *owner* dos ficheiros do *Telegraf* e dar-lhe permissões para ser executado.

No que tocas às métricas recolhidas, o *Telegraf* possui um conjunto bastante vasto de *plugins*, que podem ser instalados para a recolha de métricas específicas de cada componente (*NGINX*, *Redis*, *PostgreSQL*,...).

Assim, para recolher as métricas indicadas na Tabela 1, foram instalados os seguintes *plugins*:

- nginx;
- redis;
- postgresql;
- haproxy;
- cpu;
- system;
- mem;
- disk;
- network.

Relativamente aos *plugins*: cpu, system, mem, disk e network, estes foram instalados em todos os componentes do sistema.

Falta, ainda, referir como foram recolhidas as métricas de *SNMP* relativas ao *hardware* onde está *deployed* cada serviço.

Para recolher estas métrica foi criado um *script python* de raiz. Apesar do *Telegraf* possuir um módulo para a leitura de métricas *SNMP*, este não oferece qualquer mecanismo para lidar com o dinamismo da infra-estrutura utilizada. Um vez que os *containers docker* que encapsulam os componentes do sistema podem ser *deployed* em qualquer nó do *swarm*, é necessário um mecanismo que permita adicionar e remover, de forma dinâmica, os novos swarm nodes onde estão os componentes.

Assim, foi criado um *script* em *python*, que executa uma chamda a um dos *swarm managers*, para saber em que nó está localizado cada componente do sistema. Com esta informação, temos de listar quais os novos nós a serem monitorizados e remover os nós que já não estão a hospedar os compnentes do Misago.

Tendo uma listagem atualizada dos nós a monitorizar, o *script python* irá realizar diversas chamadas do tipo *snmpget* de forma a obter todos os OIDs que se pretendem monitorizar.

As novas métricas são obtidas de 15 em 15 segundos, sendo que, de 3 em 3 minutos, é consultado um *swarm manager* para obter uma nova listagem da localização, em *hardware*, de cada componente do serviço.

Após a obtenção destas métricas, é efetuada uma inserção 'manual' das métricas obtidas numa *timeseries* da *InfluxDB*, através do módulo *python* que esta base de dados disponibiliza. Os dados são depois visualizados através de um *dashboard*, ao nível do *Chronograf*.

No Trecho de código 6 pode ser observada uma porção do *script* de obtenção das métricas *SNMP*.

```
# the metrics will be updated 12 times, 1 time each 15 seconds
for run in range(0,20):
# get all snmp data
for ip in computes_being_used_ips:
    valid = True

    # data that will be added to influx db
    data_to_add = {
        "measurement":"snmp",
        "tags": {},
        "time": datetime.now(),
        "fields": {}
    }

    # get all snmp values
    for oid in oids:
        try:
            output = subprocess.check_output(f"snmpget -v 2c -c
   gicgirs  {ip} {oid[0]}", shell=True).decode("utf-8")
            output_value = process_output_type(output.strip())
            data_to_add["fields"][oid[1]] = output_value
        except:
            print("Error on", ip)
            valid = False

        # if a rogue node is found
        if not valid:
            break

    # correct the tag
    if "hostname" in data_to_add["fields"]:
```

```
31          data_to_add["tags"]["hostname"] = data_to_add["fields"]["
    hostname"]
32          del data_to_add["fields"]["hostname"]
33          print("Write points:", data_to_add)
34          all_metrics_to_be_added.append(data_to_add)
35
36
37  # add all metrics to influx db
38  client.write_points([data_to_add])
39
40  #wait for next iteration
41  print("Sleeping...\n\n")
42  time.sleep(15)
```

Trecho de código 6: porção do *script* de obtenção de métricas *SNMP*

Tendo já recolhido todas as métricas desejadas, é necessário criar uma visualização das mesmas, de forma a que sejam facilmente interpretadas por um administrador. Para isto, recorri o *Chronograf*, que me permitiu facilmente criar *dashboards* para cada conjunto de métricas. Estes *dashboards* estão indexados pelo *ID* de cada componente, pelo que podemos facilmente ver a informação pormenorizada de todos os componentes.

Nas Figuras 9, 10, 11, 12 e 13 podem ser visualizados os *dashboards* criados através do *Chronograf*.

5.1.2 Mecanismos de monitorização de *logs* textuais

Assim como mencionado na Secção 3.4.2, recorri à *ELK Stack* para monitorizar e extrair conhecimento dos *logs* textuais dos vários componentes do sistema.

A instalação dos diferentes componentes da mesma foi bastante simples, sendo que o único problema encontrado se deveu a ter utilizado o *Filebeat 6*, em conjunto com o *Elasticsearch 7*. Para que o *deployment* desta *stack* seja mais facilitado, devem ser utilizadas versões iguais de todos os componentes.

Antes de monitorizar os *logs* textuais é necessário encontrar os ficheiros que os armazenam, dentro de cada sistema, bem como, por vezes definir os próprios mecanismos de *logging* de um componente. Isto verificou-se com o servidor aplicacional, que não produzia, nem armazenava quaisquer tipos de *logs*. Assim foi necessário alterar a aplicação para que esta produzisse *logs* textuais, o que foi alcançado/implementado ao nível do *Gunicorn*. Utilizando algumas *features* oferecidas por este servidor, defini um *template* para os *logs* da aplicação do Misago. Isto pois, de acordo com o *SLA*, precisamos de saber qual o tempo de resposta do servidor aplicacional, uma vez que existem limitações face a este. Assim sendo, o servidor

de *Gunicorn* foi 'lançado' com uma configuração de *logging* passada por argumento (ver Trecho de Código 7).

```
gunicorn ... --access-logformat 'Host:%(h)s %(l)s Username:%(u)s
    DateOfRequest:%(t)s Method:%(m)s Status:%(s)s URL:%(U)s
    ReponseLength:%(B)s Agent:%(a)s  ResponseHeader:%({server-timing
    }o)s' --access-logfile '-' --error-logfile '-'
```

Trecho de código 7: *Deployment* do *Gunicorn* com *custom logging*

Para além da criação de um *template* de *logging* foi, também, necessário, expôr estes *logs* para o *stdout*, de forma a que pudessem ser recolhidos pela *driver* de *Syslog* do *docker*, como mencionado na Secção 3.4.2.

Também ao nível do *Redis* e do *PostgreSQL* foi necessário redirecionar os *logs* para o *stdout*.

No que toca ao *Filebeat*, este foi utilizado para exportar os *logs* das intâncias de *NGINX*. Neste caso, não foi necessário disponibilizar os *logs* no *stdout*, uma vez que facilmente conseguimos definir na configuração do *Filebeat* qual a localização dos *logs* a serem exportados. Isto quando a localização é diferente daquela por omissão. Se os *logs* estiverem na sua localização por omissão (*/var/log/NGINX/*, a simples intalação do módulo de *NGINX*, ao nível do *Filebeat*, é suficiente.

Uma vez que, a extração de de conhecimento e o *parsing* executado pelo *Filebeat* são suficientes para os objetivos definidos para monitorização das intâncias *NGINX*, os *logs* foram diretamente enviados para o *Elasticsearch*, não passando pelo *Logstash*.

A instalação do *Filebeat* ao nível de cada imagem é bastante simples, uma vez que pode ser instalado com um simples *apt install*, e a sua configuração transferida para o *container* utilizando as *docker configs*. Contudo, uma vez que o *NGINX* tem como imagem base um Linux Apline, a instação torna-se mais complexa, pelo que decidi copiar os binários do *Filebeat* para dentro da imagem, aquando o *build* da mesma. Isto torna a imagem bastante mais pesada, uma vez que estes binários têm cerca de 90 MB.

Relativamente aos *logs* recolhidos através da *driver* de *Syslog* do *docker*, estes são enviados para o *Logstash*, onde é feito o seu *parsing*. Para tal, como já mencionado na 3.4.2, foi utilizado o *Grok*.

Podemos observar no Trecho de Código 8 o *parsing* dos *logs* do *Gunicorn*, através do *Grok*.

```
...
filter {
    grok {
        ...
    }
    ...
    grok {
        match => { "message" => "<%{NUMBER:random_number}>%{
    SYSLOGTIMESTAMP:syslog_timestamp} %{DATA:from}\.%{DATA:replica
    }\.%{DATA:container}.(?:\[%{POSINT:syslog_pid}\])?: Host:%{
    IPORHOST:[gunicorn][request][ip]} - Username:%{DATA:[gunicorn][
    request][user]} DateOfRequest:\[%{HTTPDATE:[gunicorn][request][
    date]}\] Method:%{DATA:[gunicorn][request][method]} Status:%{
    NUMBER:[gunicorn][request][status]} URL:%{DATA:[gunicorn][
    request][url]} ReponseLength:%{NUMBER:[gunicorn][response][
    length]} Agent:%{DATA:[gunicorn][request][agent]} (.*
    TimerPanel_total_time=%{NUMBER:[gunicorn][response][total_time
    ]};.*SQLPanel_sql_time=%{NUMBER:[sql][response][total_time]};.*
    SQL %{NUMBER:[sql][response][total_queries]}.*
    CachePanel_total_time=%{NUMBER:[cache][response][total_time]}.*
    Cache %{NUMBER:[cache][response][total_calls]}.*)?"}
    }
}
```

Trecho de código 8: *Parsing* dos *logs* do *Gunicorn*, através do *Grok*

É importante notar que, ao nível do *Logstash*, podemos ter diversos *parsers Grok*, de acordo com os diferentes *logs* textuais que vamos monitorizar.

Depois do *parsing*, os *logs* são enviados para o *Elasticsearch*, onde são auto-indexados. Após isto, podemos visualizar os dados que retirámos dos *logs* textuais, através do *Kibana*. Uma vez que a maioria dos *default dashboards* do *Kibana* estão feitos para dados recebido através do *Filebeat*, o envio de *logs* através do *Syslog* gera alguma complexidade adicional, na medida em que pode ser necessário gerar *dashboards*, manualmente, para os nosso dados. No caso do *NGINX*, utilizei 2 *dashboards* padrão, sendo que para os restantes dados, contruí *dashboards* bastante simples, uma vez que a visualização de informação não é o foco desta unidade curricular.

Na Figura 3 podem ser observados os dados extraídos, através do *parsing* efetuado com base no Trecho de Código 8.

Os restantes *dashboards* criados podem ser observados nas Figuras 14, 15, 16 e 17.

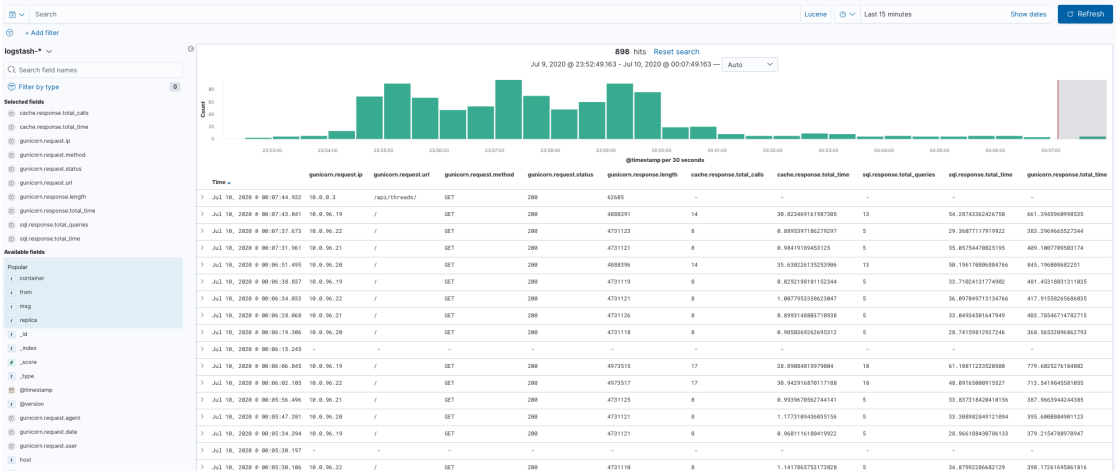

Figura 3: *Dashboard* relativo aos *logs* textuais do servidor aplicacional *Misago*

5.1.3 Implementação dos mecanismos de alarmística

No que toca aos mecanismos de alarmística, foi implementado um sub-conjunto dos alarmes mencionados na Tabela 2. Isto deve-se meramente ao facto do estabelecimento destes alarmes ser uma tarefa repetitiva (muitas vezes é apenas selecionar uma métrica, um *threshold* e definir para onde enviar uma notificação), de onde, devido à sua repetitividade, não resulta num maior conhecimento da pessoa que cria estes alarmes.

Assim foram efetivamente implementados os seguintes alarmes:

- Ativação de um alarme sempre que a percentagem de *RAM* utilizada de um *Docker Node* ultrapassa os 55 %;
- Ativação de um alarme sempre que a percentagem de utilização do *CPU* de um *Docker Node* ultrapassa os 50 %;
- Ativação de um alarme sempre que o número de clientes de uma instância *NGINX* ultrapassa os 750;
- Ativação de um alarme sempre que o tempo de resposta de um *GET* ultrapassa os 10 segundos.
- Ativação de um alarme sempre que um utilizador efetua 8 logins errados no espaço de 30 segundos.

Na Figura 4 podemos ver os alarmes definidos ao nível da *TICK Stack*.

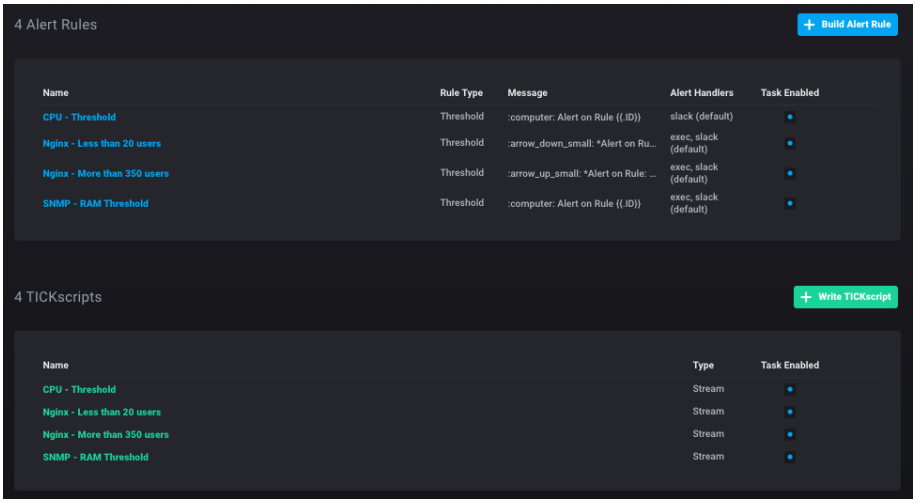

Figura 4: Alarmes definidos ao nível da *TICK Stack*

Mais tarde, para efeitos de aprovisionamento/desaprovisionamento de novos elementos, foram definidos novos alarmes, que são abordados na Secção 5.2.

Relativamente aos alarmes sob as métricas de *RAM* e *CPU*, obtidas por *SNMP*, estes são bastante semelhantes ao alarme definido para o número de clientes de uma instância de *NGINX*, sendo que apenas são utilizadas diferentes métricas e um valor diferente para o *threshold*, pelo que apenas irei abordar a implementação do mecanismo de alarmística para o excesso de clientes ao nível das instâncias de *NGINX*.

1. Implementação do alarme relativo ao número máximo de clientes numa instância de *NGINX*

Para a implementação deste alarme foi tida em conta a métrica *active users* do *NGINX*, sendo que se utilizou a mediana deste valor, quando agrupada em intervalos de 5 segundos. O alarme foi definido com recurso ao *Kapacitor*, da *TICK Stack*, onde apenas foi necessário definir o *threshold* de ativação e a métrica a ser monitorizada. A interface visual através da qual foi definido este alarme pode ser vista na Figura 5.

Caso um utilizador não pretenda definir nenhum *Alert Handler*, este será notificado através a interface visual *Alert History* da *TICK Stack*, onde é possível visualizar os alarmes atividados, o seu *timestamp* e qual o valor que os despoletou. Contudo, este não foi o cenário ao qual recorri.

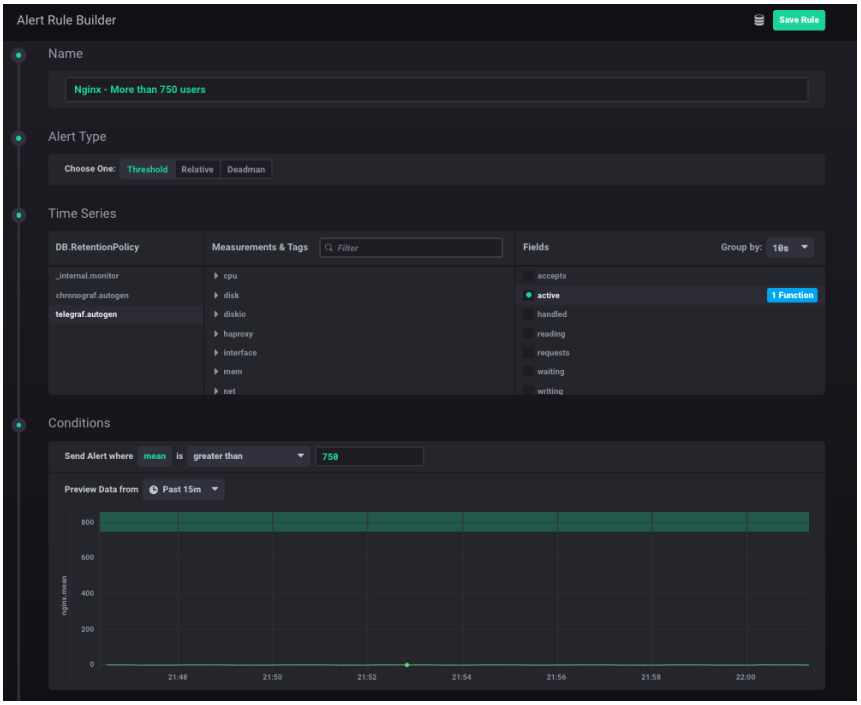

Figura 5: Interface visual do *Kapacitor*

Como mencionado na Secção 3.4.3, foi criado um *workspace slack* para onde são enviados todos os alertas criados pela estrutura de monitorização/alarmística. A criação deste *workspace*, foi bastante simples.

Para que seja possível receber uma notificação num canal do *slack*, foi necessário criar uma *slack app* e definir um *webhook*, que permite comunicação com a mesma. Posteriormente, na *TICK Stack* foi criado um *alarm handler* que utiliza o *webhook* definido anteriormente para enviar uma mensagem para o canal *alerts* do *slack*. Esta mensagem contém informações relativas à instância que ultrapassou o excesso de utilizadores, bem como ao valores de utilizadores que despoletou o alarme. Esta pode ser observada na Figura 6. Note-se que, para efeitos de teste, foram utilizadoss valores diferentes daqueles definidos na Tabela 2.

Para além do envio da mensagem de alarme para o *slack*, foi ainda definido outro *alarm handler*. Sempre que o alarme é ativado, é executado um *script* em *python* com o objetivo de aprovisionar e desaprovisionar novos elemento. Este *mecanismo* é abordado na Secção 5.2, sendo que a sua execução é possível graças ao *alarm handler* '*execute*', da *TICK Stack* (ver Figura 7).

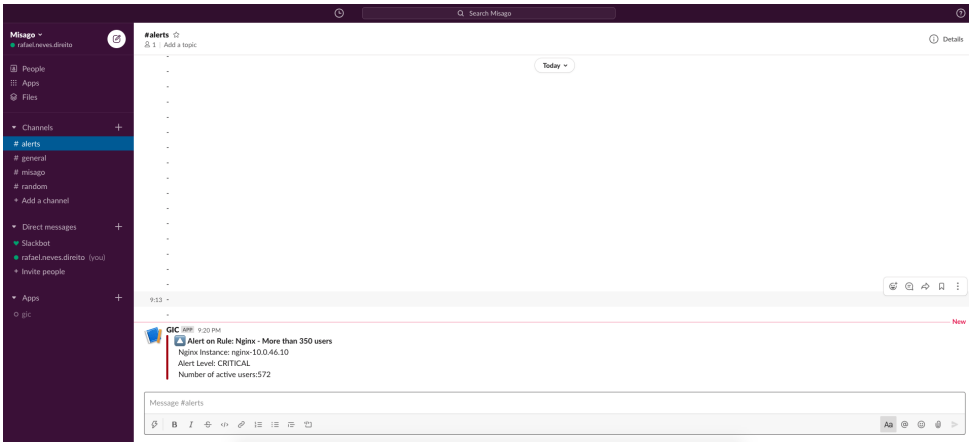

Figura 6: Envio de um alarme para o *slack*

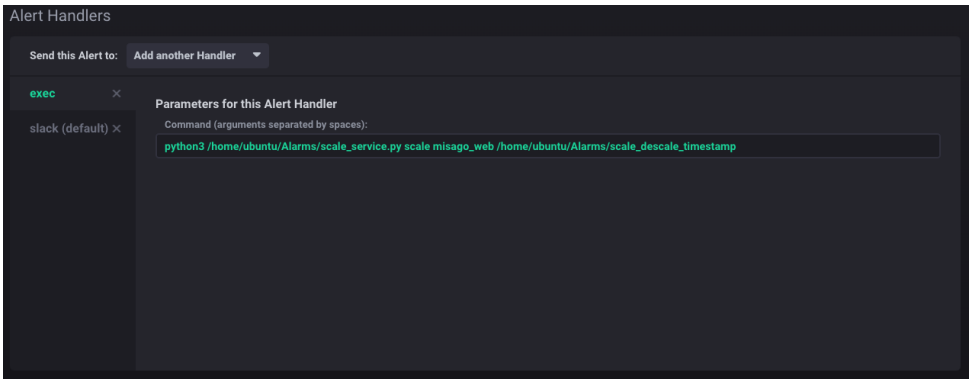

Figura 7: *Alarm Handler* da *Tick Stack*

2. Implementação do alarme relativo ao tempo de resposta do *Misago*

Para a alarmística relativa ao tempo de resposta do servidor aplicacional do *Misago*, foram utilizados os *logs* textuais do *Gunicorn*, recolhidos através do *Syslog*. Estes *logs* são indexados de forma automática pela *Elasticsearch*, pelo que foi necessário alterar alguns campos, de forma a permitir realizar as *queries* necessárias para a deteção dos eventos que estão na base deste alarme.

Após estas alterações, utilizaram-se os *Kibana Alerts* - mecanismos de alarmística lançado na versão 7 do Kibana, e que ainda se encontram em fase *Beta* - para a definição da métricas a monitorizar, bem como os *thresholds* que as limitam. Assim sendo, defini que, se durante os últimos 3 minutos, a média do tempo de resposta ultrapassar continuamente os 10 segundos, deverá ser ativado um alarme. Esta de-

finição pode ser observada na Figura 8. Note-se que, para efeitos de teste, foram utilizados valores diferentes daqueles definidos na Tabela 2.

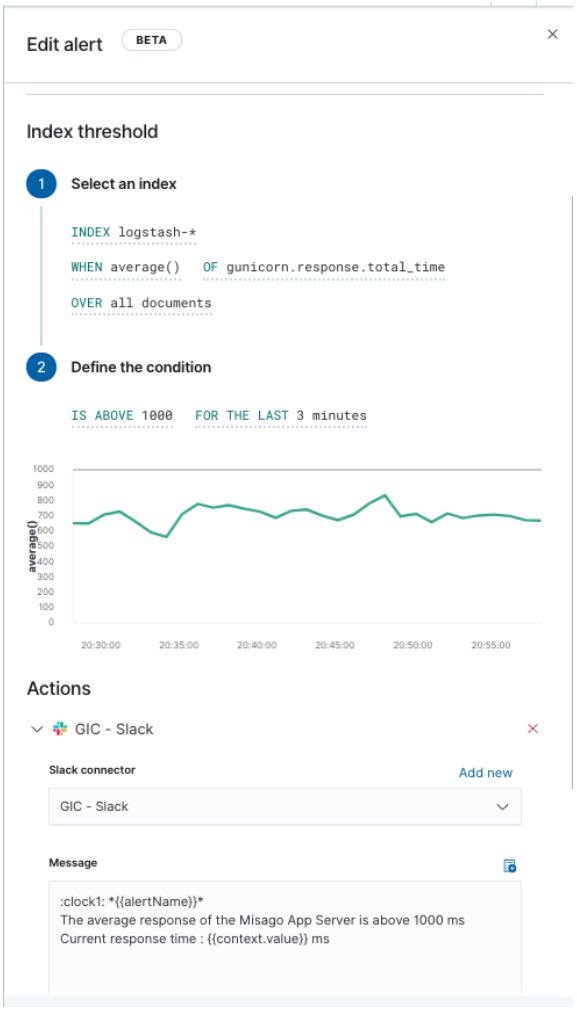

Figura 8: Definição do alarme relativo ao tempo de resposta do *Misago*

Tal como definido anteriormente, a ativação de um alarme gera uma mensagem que será enviada para o *slack* do *Misago*, através de um *webhook*.

Relativamente às ações tomadas face à ativação deste alarme, estas serão de aprovisionamento de novos recursos. Ao contrário do *Kapacitor* os *Kibana Alerts Actions* não permitem a execução direta de *scripts*, pelo que foi necessário disponibilizar o código para aprovisionamento de recursos através de uma aplicação *cherrypy*. O *Kibana* envia um *POST* para esta aplicação, através do seu URL.

A ativação deste alarme pode ser observada através de um vídeo disponível

em `https://youtu.be/FpyD-DIsg50`.

3. Implementação do alarme relativo a um elevado número de *logins* incorretos, num curto período de tempo

Tal como mencionado anteriormente, o cenário de utilização do *Misago* poderá ser passível a diversos ataques de vários alunos. Assim, é necessário definirmos alarmística que permita que os administradores do sistema sejam informados sempre que existirem comportamentos erráticos da parte dos utilizadores.

Procedi, então, à implementção de um alarme, sempre que um mesmo *IP* se tentar autenticar erradamente 8 vezes, no espaço de 30 segundos.

Para definir este alarme, recorri ao *Watcher* do *Elasticsearch*, sendo que foi mesmo necessário escrever manualmente *queries* para o *Elasticsearch*, de forma a obter a informação pretendida. Nestas *queries* observamos o número de pedidos às API de autenticação, provenientes de um mesmo *IP*, que resultaram em resposta HTTP do tipo 400. Estes resultados são depois filtrados, de forma a considerarmos apenas os utilizadores que fizeram mais de 8 pedidos em 30 segundos.

Quando é ativado este alarme, é enviada uma mensagem para o *slack* de administração do *Misago* e enviado, também, um pedido HTTP para uma aplicação *cherrypy* que está *deployed* nas instâncias de *NGINX*. Esta *call* informa as instâncias *NGINX* do *IP* do atacante, sendo que esste será impedido de aceder ao sistema.

No Trecho de Código 9 podemos observar uma porção da implementação da *querie* de obtenção dos *IPs* dos atacantes, sendo que a ativação deste alarme pode ser visualizada através de um vídeo presente em `https://youtu.be/1IvFWvsKQOc` .

```
...
"query": {
"bool": {
  "filter": [
    { "range": {
        "@timestamp": { "from": "now-30s",  "to": "now"}
      }
    },
    {"match": { "http.request.method": "POST"}},
    {"match": {"http.response.status_code": 400}},
    {"match": {"url.original": "/api/auth/"}
    }
  ]
}
},
"aggs": {
```

```
"failed_ip": { "terms": { "field": "source.ip", "min_doc_count": 5}},
"count": {"cardinality": {"field": "source.ip" }}
}
...
"condition": {"compare": { "ctx.payload.aggregations.count.value": {"gte": 1}}}
```

Trecho de código 9: Implementação da *querie* de obtenção dos *IPs* dos atacantes

5.2 Implementação dos mecanismos de automação para aprovisionamento e desaprovisionamento de novos elementos

Tendo como base os mecanismos de alarmística definidos na Secção 3.4.3, foram tomadas diversas medidas para automatizar a resposta aos alarmes ativados.

De acordo com o *SLA*, definido na Tabela 1, existem determinadas métricas de *performance* que têm de ser mantidas pelo sistema, sendo, talvez, as mais importantes, aquelas relacionadas com o tempo de resposta do mesmo. Assim é necessário criar mecanismos que permitam manter baixos tempos de resposta.

Uma vez que o sistema assenta num *docker swarm* podemos facilmente utilizar os mecanismos de *scaling* oferecidos pelo este. Tal como mencionado anteriormente, quer o *NGINX*, quer o servidor aplicacional do *Misago*, são completamente *stateless*, pelo que faccilmente podemos aumentar o seu número de réplicas e deixar que o *docker swarm* trate do balanceamento de carga, sem necessidade de efetuar qualquer configuração adicional.

Esta foi a abordagem utilizada quando se verifica um número muito elevado de clientes, ao nível de uma instância de *NGINX* e quando os tempos de resposta do *Misago* ultrapassam os 10 segundos.

No último caso, quando este alarme é ativado, ao nível de *Kibana*, é enviado um pedido *HTTP* para uma aplicação *cherrypy*. Esta aplicação será responsável por aumentar o número de réplicas do servidor aplicacional do *Misago*. Esta aplicação recebe o nome do serviço que será escalado e aumenta o seu número de réplicas em 1 unidade.

Contudo, nos momentos em que o sistema não está sob uma forte utilização, é pretendido que exista uma poupança de recursos, de forma a permitir uma operação mais económica do mesmo. Para tal, também é necessário definirem-se alarmes que sejam ativados sempre que o sistema esteja em pouco esforço. Este tipo de alarmes foi implementado, por exemplo, ao nível do *NGINX*, onde, sempre que o número de

clientes de uma instância é inferior a 150, podemos efetuar o desaprovisionamento das instâncias deste serviço.

Nos alarmes implementados através do *Kapacitor* da *TICK Stack*, estes invocam diretamente o *script* de aprovisionamento e desaprovisionamento de novos elementos. Para tal, é necessário enviar qual o serviço que pretendemos alterar, qual a operação que prentendemos efetuar (aprovisionamento ou desaprovisionamento) e a localização do ficheiro que guarda o *timestamp* da última alteração daquele serviço. Este *timestamp* é importante para manter a consistência do sistema, sendo que só pode ser efetuado um provisionamento/desaprovisionamento a cada 5 minutos. Caso fossem enviados 3 pedidos de provisionamento no espaço de 30 segundos, por exemplo, estaríamos a escalar desnecessariamente o serviço. Este *timestamp* leva a que só seja equacionada outra ação de provisionamento/desaprovisionamento quando o sistema se encontrar estável.

Relativamente ao número de réplicas de cada serviço, estas têm de estar entre 1 e N (definido pelo utilizador), sendo que o *script* criado se assegura deste facto.

Este *script* foi criado em *python* e executa chamadas à *API docker* dos *swarm managers*, sendo que pode ser observada um porção do mesmo no Trecho de Código 10.

```
...

# if we are trying to descale a service wiht only 1 replica - leave
if replicas < 2 and operation == "descale" :
    print("Wont descale a service with 1 replica!")
    exit(0)

may_scale_or_descale = True
# check when was the last scale/descale down
if os.path.isfile(operation_timestamp_file_path):
    operation_timestamp_file = open(operation_timestamp_file_path)
    last_op_date_str = operation_timestamp_file.readline().strip()
    operation_timestamp_file.close()

    last_op_date = datetime.strptime(last_op_date_str, "%d-%b-%Y %H
    :%M:%S")
    # get current time
    now = datetime.now()
    # get seconds from the last operation
    diff = now - last_op_date
    duration_in_s = diff.total_seconds()
    if duration_in_s < 5 * 60:
        may_scale_or_descale = False

# scale or descale
```

```
25 if may_scale_or_descale:
26     # compute new number of replicas
27     updated_replicas = replicas - 1 if operation == "descale" else
       replicas + 1
28
29     # scale or descale service
30     print(f"Operation {operation}, on {docker_service} ...")
31     to_execute = f"export DOCKER_HOST=\"{docker_manager}\" &&
       docker service scale {docker_service}={updated_replicas}"
32     subprocess.check_output(to_execute, shell=True)
33
34     print(f"Operation {operation} on {docker_service} has been
       completed. This service now has {updated_replicas} replicas!")
35
36     # save timestamp to file
37     now_str = datetime.now().strftime("%d-%b-%Y %H:%M:%S")
38     operation_timestamp_file = open(operation_timestamp_file_path,
       "w")
39     operation_timestamp_file.write(now_str)
40     operation_timestamp_file.close()
41 else:
42     print("The last operation was committed too recently!")
```

Trecho de código 10: *Script* de automação para o (des)aprovisionamento de novos elementos

Uma vez que estas operações de aprovisionamento/desaprovisionamento de novos elementos são difíceis de demonstrar através de imagens, foi gravado um vídeo onde se pode observar o mecanimos implementado. Este pode ser encontrado em `https://youtu.be/co_5Rx7L61k`.

Note-se que, mal sempre que o sistema era iniciado, com 3 réplicas de *NGINX*, o mecanismo de desaprovisionamento de recursos atuava imediatamente, diminuido o número de réplicas deste serviço.

Falta referir o *script* criado para o bloquear o acesso de determinados *IPs* ao sistema. Este foi escrito em *python* e recorre ao *cherrypy* para disponibilizar métodos que permitam que a *VM* de monitorização envie o *IP* a bloquear/desbloquear através de uma chamada *HTTP*.

Para bloquear os utilizadores, iria recorrer-se às *iptables*, contudo, a alteração das *iptables* teria de decorrer ao nível dos *docker nodes* que hospedavam cada servidor *NGINX*. No contexto desta disciplina, onde existem diversos alunos a usar o *docker swarm*, o mecanimos apresentado acima poderia por em causa o trabalho dos restantes utilizadores do *swarm*, pelo que este, apesar de disponível, não efetua qualquer alteração na *iptables*, servindo apenas como demonstração.

No Trecho de Código 11, podemos encontrar uma porção da aplicação de bloqueio de *IPs*.

```
class BlockIps:
    # http://localhost:8081/blockIp?ip=1824.124.123.1
    @cherrypy.expose
    def blockIp(self, ip=None):
        if ip:
            block_command = f"iptables -A INPUT -s {ip} -j DROP"
            print(block_command)
            status = os.system(block_command)
            return "success" if status == 0 else "error"
        return 'No Ip was passed' if ip is None else 'No Ip was
    passed'

    # http://localhost:8081/allowIp?ip=1824.124.123.1
    @cherrypy.expose
    def allowIp(self, ip=None):
        if ip:
            allow_command = f"iptables -D INPUT -s {ip} -j DROP"
            print(allow_command)
            status = os.system(allow_command)
            return "success" if status == 0 else "error"
        return 'No Ip was passed' if ip is None else 'No Ip was
    passed'
```

Trecho de código 11: *Script* para bloquear um *IP*

6 Testes e *Benchmarking*

Para testar o sistema operacionalizado, foi utilizado o *Locust* - um *framework* de testes, criado em *Python*. Este *framework* lança N users, que executam diversos pedidos *HTTP*, consoante o seu *lifecyle*. Este pode ser facilmente definido por um programador, através de um conjunto de *tasks*. O programador pode definir, também, qual a probabilidade de execução associada a cada *task*.

Foram, então, criadas diversas *tasks*:

- Consultar uma *thread*;
- Responder a uma *thread*;
- Escrever uma nova *thread*;
- Subscrever uma *thread*;
- Listar as *threads* subscritas pelo utilizadore;
- Listar as *threads* do fórum que ainda não forma lidas;
- Listar as novas *threads* do fórum;
- Listar as *threads* que escreveu.

Estas *task* obedecem às restrições de probabilidades definidas na Secção 2.1, pelo que é muito mais provável que um utilizador consulte uma *thread* existente, do que escreva uma nova *thread*.

Para definir quais os métodos HTTP que deveriam ser chamados em cada *task*, foi necessário executar as diversas *calls* manualmente e listar os métodos individuais que eram chamados.

Após esta listagem, foram removidas todas as chamadas que envolviam ficheiros estáticos. Isto porque o *Locust* não providencia qualquer mecanismo de *caching*, pelo que iria necessitar de pedir os ficheiros estáticos diversas vezes, o que não representa um cenário real, onde um utilizador tem estes ficheiros estáticos já em *cache* e consegue aceder-lhes em menos de 5 ms.

Tendo sido criado o *lifecyle* de cada utilizador, foram lançados 1500 utilizadores, de acordo com o cenário de utilização apresentado na Secção 2.1, sendo que foram gerados 10 novos utilizadores por segundo, até se atingir o número de utilizadores pretendido. Os utilizadores, durante o seu *lifecycle*, executam 1 tarefa a cada 30 segundos.

Pela análise dos dados fornecidos pelo *Locust* pode concluir-se que as operações de *POST* e *PATCH* relacionadas com a escrita de novas *threads*, resposta a *threads* e a subscrição destas, ao contrário do esperado, não tiveram um tempo de resposta muito elevado. Isto pode estar relacionado com a elevada *performance* das bases de dados *PostgreSQL*.

Na Tabela 4 podemos observar os resultados obtidos a partir do teste de carga.

Ação a executar	**Mediana do tempo de resposta**	**Tempo mínimo de resposta**	**Tempo máximo de resposta**	**% de falhas**
Login	0.743 s	0.339 s	4.44 s	0.42%
Escrever Thread	0.508 s	0.292 s	58.569 s	0.7%
Ler Thread	0.073 s	0.035 s	0.851 s	0.45%
Subscrever Thread	0.417 s	0.316 s	12.019 s	0.41%
Responder a Thread	0.521 s	0.392 s	46.591 s	3.3%
Verificar Minhas Threads	0.670 s	0.377 s	34.577 s	1.12%
Verificar Novas Threads	1.110 s	0.630 s	71.635 s	1.01%
Verificar Threads Não Lidas	0.565 s	0.338 s	69.147 s	1.58%
Verificar Threads Subscritas	0.473 s	0.291 s	68.27 s	1.07%

Tabela 4: Tempos de resposta

No que toca ao *SLA* este apenas foi violado no que toca à utilização de *RAM* ao nível dos *docker nodes* e da percentagem do espaço de disco livre. Contudo, estes aspetos não são originados pelos componentes do sistema, em si, estando relacionados com as características da infra-estrutura subjacente.

Relativamente ao número de instâncias dos servidores aplicaionais do *Misago*, apesar de o teste ter começado com 1 instância, rapidamente foi escalado para 2

instâncias, sendo que se manteve com esse número de servidores até ao final do teste, que durou cerca de 15 minutos.

O componente *NGINX* foi, também, inicializado com apenas 1 réplica, contudo, foi escalado para 2 réplicas e, posteriormente para 3, sendo que no final do teste, e durante a mioria da sua duração, existiram 3 servidores *NGINX*. Isto pode parecer um número muito elevado de servidores para apenas 1500 utilizadores. Contudo, uma vez que cada *docker node* só tem 2 *VCPUs*, pelo que, só são lançados 2 *NGINX workers* - seguindo as recomendações da documentação de *NGINX* [5], que conseguem suportar, cada um, até 512 conexões, ficamos com um número máximo de 1024 conexões suportadas por servidor de *NGINX*.

No decorrer destes testes, infelizmente, não foi possível recorrer ao *Pgpool*, uma vez que existiam diversos problemas ao nível da rede do *docker swarm*, que acabavam por não permitir o tráfego de informações do *Pgpool* para o *cluster* de bases de dados *PostgreSQL*. Assim, devido ao *Healthcheack* do *Pgpool* este era reiniciado constantemente. Ao realizar os mesmos testes em *local*, tal acontecimento não se verificou, pelo que, tendo em conta o historial associado à rede do *docker swarm* utilizado neste unidade curricular, estas falhas, provavelmente, sejam originadas por falhas de rede, na infra-estrutura de *deployment*.

7 Conclusão

A operacionalização total de um produto é algo bastante complexo, sendo que envolve uma estratégia bem estruturada e que contemple aspectos como o balanceamento de carga, redundância de elementos, recuperação de falhas, monitorização, alarmística, entre outros.

Com este projeto foi possível planear e implementar todas estas fase, o que forneceu uma vista geral dos processos relativos à operacionalização de um produto e à sua elevada complexidade.

A utilização de uma grande *stack* de tecnologias foi, também, um aspeto bastante positivo proveniente da execução deste projeto, sendo que certamente utilizarei os conhecimentos adquiridos no decorrer de outros projetos, quer profissionais/universitários, quer pessoais.

Como trabalho futuro, surge a realização de mais testes de carga sobre a plataforma, algo a que não foi dada prioridade durante este projeto. Também podem ser estudadas alternativas ao *Pgpool*, de forma a que se consiga manter a estabilidade do sistema em ambientes com condições de rede que não sejam ótimas.

Referências

[1] Redis Sentinel Documentation,
https://redis.io/topics/-sentinel

[2] StackOverflow Update: 560M Pageviews A Month, 25 Servers, And It's All About Performance,
http://highscalability.com/blog/2014/7/21/stackoverflow-update-560m-pageviews-a-month-25-servers-and-i.html

[3] Performance - Stack Exchange,
https://stackexchange.com/performance

[4] StackOverflow,
https://stackoverflow.com/

[5] Documentação do NGINX,
https://nginx.org/en/docs/

8 Anexos

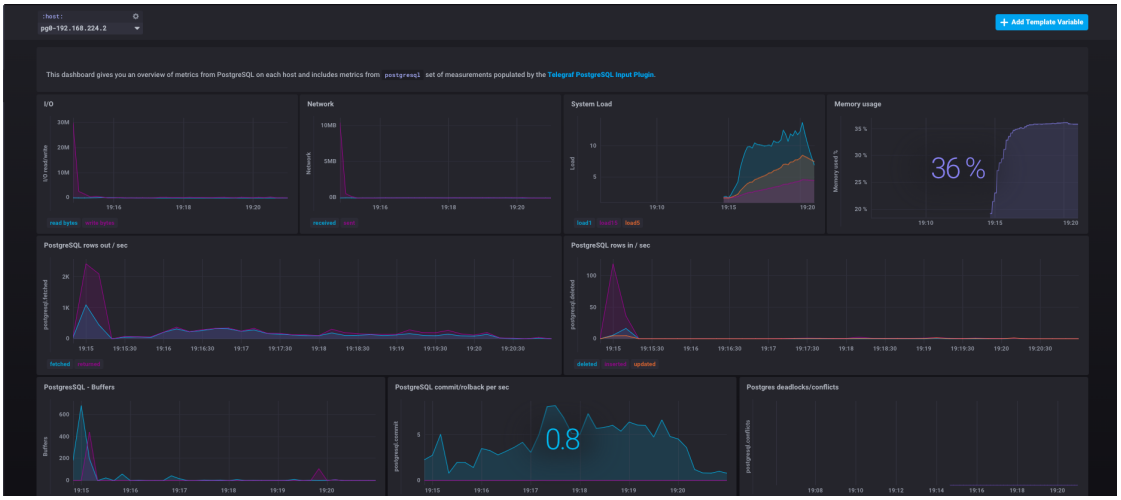

Figura 9: *Dashboard* do *Chronograf* para uma instância de *PostgreSQL*

Figura 10: *Dashboard* do *Chronograf* para uma instância de *Redis*

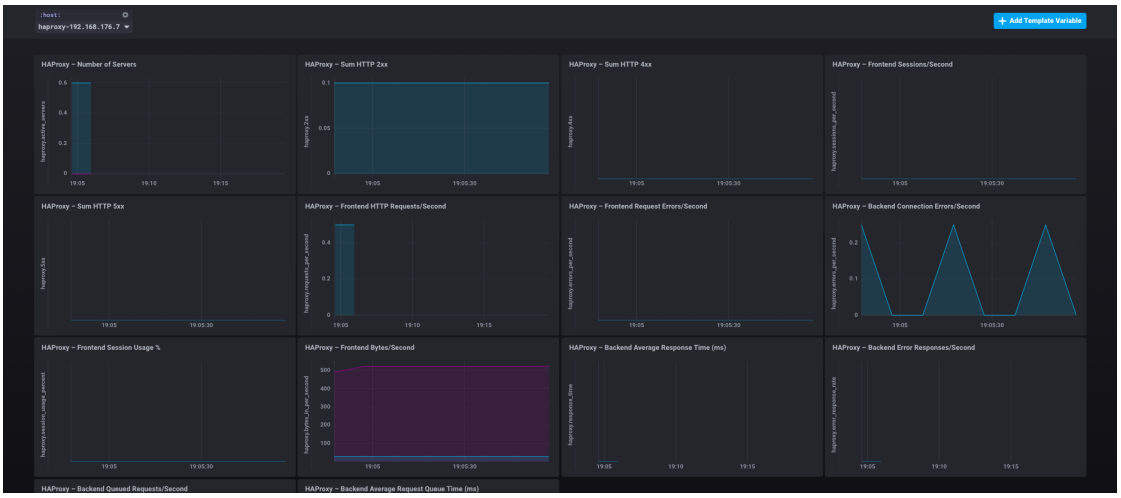

Figura 11: *Dashboard* do *Chronograf* para uma instância de *HAProxy*

Figura 12: *Dashboard* do *Chronograf* para uma instância de *NGINX*

Figura 13: *Dashboard* do *Chronograf* para um componente do *sistema*

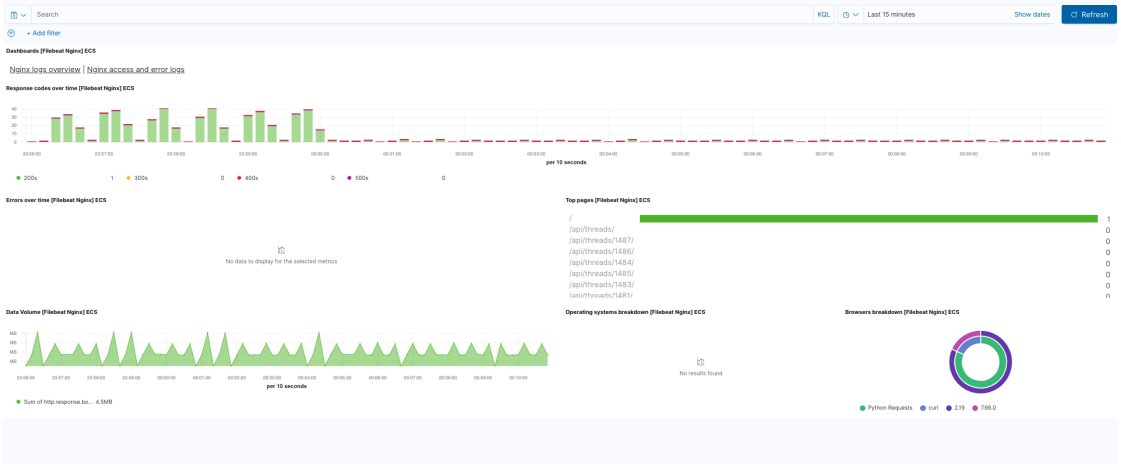

Figura 14: *Dashboard* relativo aos *logs* textuais do *NGINX* (*overview*)

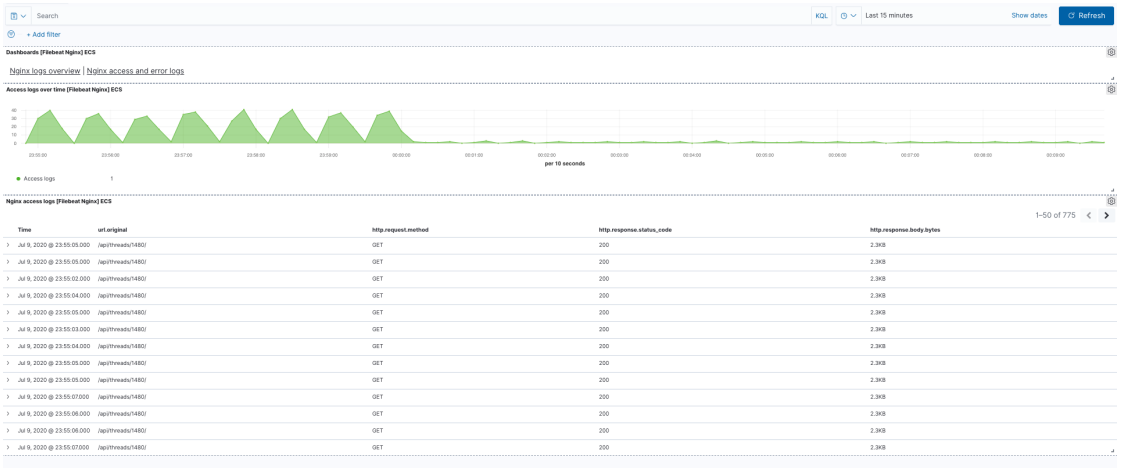

Figura 15: *Dashboard* relativo aos *logs* textuais do *NGINX* (dados de acesso)

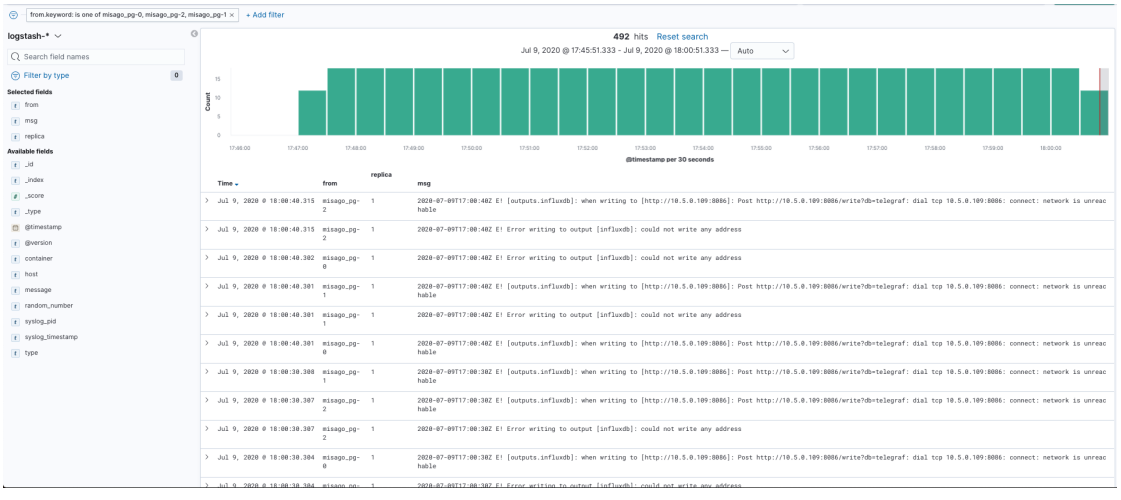

Figura 16: *Dashboard* relativo aos *logs* textuais das instâncias *PostgreSQL*

Figura 17: *Dashboard* relativo aos *logs* textuais das instâncias *Redis*