Universidade do Minho
Escola de Engenharia

Instalação, monitorização e avaliação experimental de uma aplicação distribuida

System Deployment & Benchmarking

Trabalho realizado por:

Daniel Regado, PG42577
Diogo Ferreira, PG42824
Fábio Gonçalves, PG42827
Filipe Freitas, PG42828
Vasco Ramos, PG42852

Abril de 2021

Índice

Lista de Figuras

Listagens

Capítulo 1

Introdução

1.1 Contextualização

No âmbito da UC de *System Deployment & Benchmarking* foi-nos proposta a realização de um projeto final que consiste no *deployment* de uma aplicação distribuída na plataforma *Google Cloud*, com o recurso a ferramentas como o *Ansible* e *Docker*. Este projeto deveria ser realizado em duas fazes: na primeira fase foi feita uma escolha da aplicação a instalar, uma escolha do método de *deployment*, e uma avaliação da viabilidade desse método; nesta segunda fase é suposto, recorrendo aos resultados da primeira, efetivamente desenvolver as ferramentas necessárias à instalação e ao *deployment* da aplicação. É também necessário, nesta fase, configurar ferramentas de monitorização da aplicação, e também fazer uma avaliação da sua *performance*.

1.2 Estrutura do relatório

Este relatório encontra-se dividido em 5 partes:

Na primeira parte é feita uma breve introdução a este trabalho, e é explicada a estrutura deste relatório.

Na segunda parte é feita uma breve revisão da arquitetura da aplicação escolhida, sendo explicadas também algumas pequenas alterações que foram efetuadas em relação ao que foi decidido na primeira fase.

Na terceira parte é feita uma análise *bottom-up* (no sentido em que começamos pela ferramenta *Ansible*, que é o que eventualmente vai correr as restantes ferramentas, e portanto, é a base de todo o processo de *deployment*) de todas as ferramentas que foram utilizadas para efetuar este *deployment*.

Na quarta parte é efetuada uma avaliação da *performance* da aplicação instalada, e é feita uma discussão dos resultados obtidos.

Na última parte é feita uma breve conclusão do trabalho realizado, onde são retiradas conclusões sobre o mesmo, e são apontados alguns aspetos considerados pertinentes sobre este trabalho.

Capítulo 2

Arquitetura da Aplicação

2.1 Arquitetura de Componentes

Tal como foi analisado no primeiro relatório, a arquitetura pode ser dividida em 3 componentes principais: *Frontend*, *Rest API* e Bases de Dados. O *Frontend* baseia-se em *React*, uma *framework* em *JavaScript*, a *Rest API* baseia-se em *Django*, uma *framework* em *Python*, e por fim, as bases de dados estão divididas em *PostgreSQL* e *Redis*, estando estas ligadas a uma *job queue*, em *Celery*, para executar tarefas assíncronas. A figura 2.1 esquematiza a organização do esquema, por componentes.

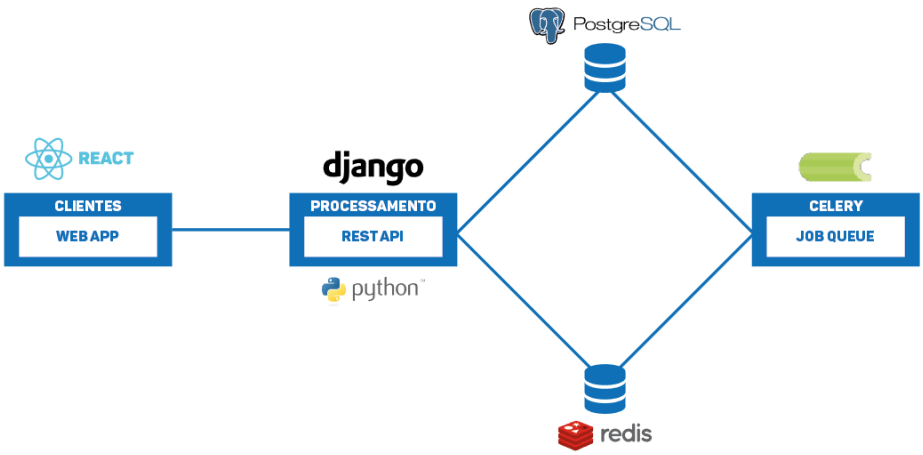

Figura 2.1: Arquitetura básica Misago

2.2 Arquitetura de Instalação/Distribuição

Como dito anteriormente, o ***Misago*** está distribuído pelos três principais componentes seguintes:

- Um servidor, onde irá correr a aplicação *Misago*, a qual é exclusivamente *Stateless*, isto é, não guarda quaisquer tipos de informação transiente ou persistente no servidor aplicacional, de acordo com [1];
- Um servidor de base de dados *PostgreSQL*, para guardar a informação persistente associada à lógica de negócio do sistema, tais como *threads* e publicações;
- Um servidor *Redis*, para *caching* e armazenar informação temporária relativa a tarefas assíncronas.

A escolha adequada dos tipos de distribuição a serem usados em cada componente é fundamental, não só para sermos capazes de satisfazer todos os pedidos realizados entre as várias camadas e componentes, mas também para garantirmos um serviço com alta disponibilidade e forte resiliência a possíveis falhas. A análise seguinte tem em conta o que foi teorizado no relatório anterior, mas também o que nos foi possível implementar no trabalho em si.

Relativamente ao **servidor aplicacional**, para este ter um maior desempenho e ser capaz de responder a altas cargas de pedidos, decidimos replicar a aplicação ***Misago*** por várias instâncias de servidores e coordená-los (principalmente ao nível de distribuição de carga) através de uma distribuição ***Proxy-Server***. Para mitigar as possíveis falhas do *proxy-server*, este poderá também ser replicado por várias instâncias e exportado para o cliente como um só.

Relativamente à base de dados transacional, ou seja, o servidor de base de dados *PostgreSQL*, tivemos em consideração as referências [2], [3], contudo, pela complexidade associada à operacionalização dos processos de replicação e distribuição de carga, apesar de concluirmos que seria vantajoso e necessário garantir replicação da base de de dados e também garantir alta disponibilidade e desempenho, não nos foi possível implementar os mecanismos discutidos no relatório anterior (estes mecanismos incluíam uma distribuição ***Master-Slave*** para distribuir carga entre várias instâncias de *PostgreSQL*, sincronizadas através de *Streaming Replication* e uma camada adicional de ***Proxy-Server*** para fazer o balanceamento e distribuição dos pedidos).

Para o servidor *Redis*, que tem a responsabilidade de *caching* e de armazenar dados associados a tarefas assíncronas é, também, necessário garantir a elevada disponibilidade deste componente. Então, à semelhança do que se fez com os servidores de base de dados *PostgreSQL*, decidimos aplicar uma distribuição bastante semelhante em que temos uma distribuição ***Master-Slave*** e por cima disso uma distribuição ***Proxy-Server*** para balanceamento e distribuição dos pedidos.

Por último, para a execução das tarefas assíncronas, que estão armazenadas no servidor *Redis*, iremos trabalhar com o *Celery*. Para tal, iremos aplicar uma distribuição de replicação para que toda a carga da execução das tarefas não esteja sobre uma única instância do *Celery*.

Assim sendo, a figura 2.2 representa graficamente as arquiteturas de **instalação** e **comunicação** que foi possível pôr em prática.

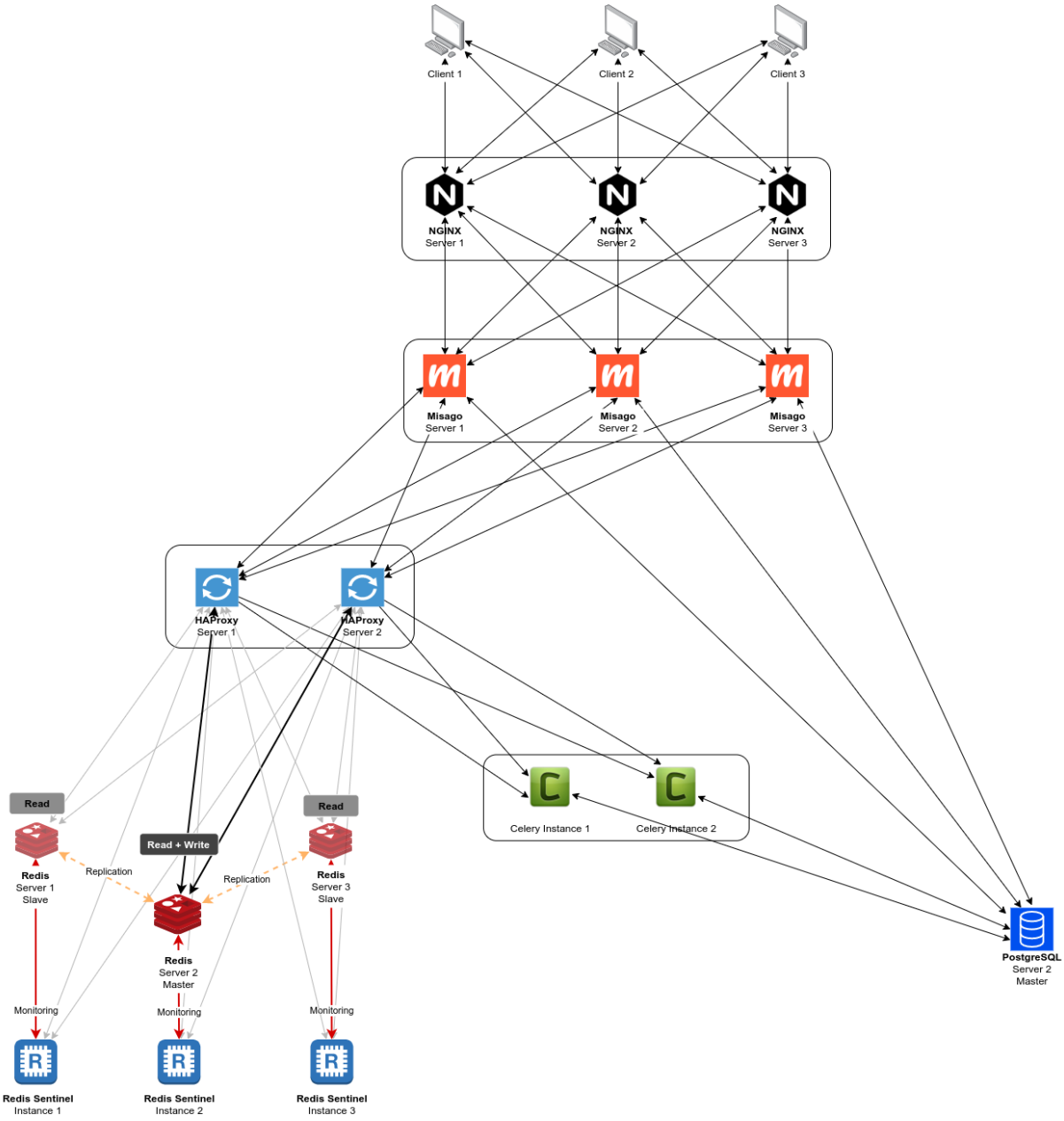

Figura 2.2: Possível arquitetura de comunicação/ *deployment*

Capítulo 3

Ferramentas Utilizadas

3.1 Ansible

O *Ansible* é uma ferramenta que permite efetuar o aprovisionamento remoto de máquinas de um modo escalável, previsível, e totalmente automático. É, portanto, uma ferramenta extremamente útil para efetuar a instalação dos componentes necessários para correr esta aplicação, e portanto, foi esta ferramenta que utilizámos para instalar algumas das ferramentas necessárias (como a *Docker Engine*), e também para fazer a criação e manutenção das máquinas virtuais na *Google Cloud*.

Assim sendo, foram criados dois *Playbooks*, cada um com o seu propósito:

- ***Playbook* 1: *playbook.yml* -** Permite efetuar todas as tarefas de criação e manutenção de máquinas virtuais, instalação da *Docker Engine*, instalação das ferramentas de monitorização, criação do *Docker Swarm*, entre outros.
- ***Playbook* 2: *scale.yml* -** Permite efetuar o escalonamento do número de *containers* a correr dos serviços de *Web*, *Celery* e *Misago*.

Decidimos também recorrer à utilização de um inventário dinâmico, configurado de um modo modular pelo Ansible. Iremos começar por aí a explicar a nossa solução.

3.1.1 Criação do inventário dinâmico

Como foi referido, neste projeto optámos por recorrer às capacidades de inventário dinâmico do *Ansible* para gerir a criação, manutenção, e eliminação das máquinas virtuais criadas na *Google Cloud*. Assim, temos uma *role*, *manage-vms*, que foi construída para que, de um modo modular, seja possível gerir todas as máquinas virtuais existentes relativas a esta aplicação, o seu tipo, nome, entre outras propriedades. Este sistema modular permite também que seja possível adicionar uma nova máquina

sem ser necessário reprovisionar todas as restantes, bastando simplesmente efetuar a edição do ficheiro correspondente.

As máquinas a criar encontram-se definidas num único ficheiro, editável e configurável conforme as necessidades da aplicação, e de modo a evitar a repetição das suas definições para efetuar diferentes operações. Para que esta solução funcionasse corretamente, foi necessário especificar as *tags* de cada uma das operações diretamente na definição das *tasks* que pertencem à *role manage-vms*.

Assim sendo, a definição das máquinas existentes é dada do seguinte modo:

Ficheiro *deployment/ansible/roles/manage-vms/tasks/main.yml*

```
3  - name: Create NFS Storage Disk
4    include: create-nfs.yml
5
6  - name: Manage Monitoring VM
7    include: manage.yml name=project-vm-monitoring type=e2-standard-2 group=monitor
     ↪  static_ip=true
8
9  - name: Manage Swarm Manager VMs
10   include: manage.yml name=project-vm-manager1 type=e2-standard-2 group=managers
     ↪  static_ip=true
11 - include: manage.yml name=project-vm-manager2 type=e2-standard-2 group=managers
   ↪  static_ip=true
12 - include: manage.yml name=project-vm-manager3 type=e2-standard-2 group=managers
   ↪  static_ip=true
13
14 - name: Manage worker VMs
15   include: manage.yml name=project-vm-worker1 type=e2-standard-2 group=workers static_ip=true
16 - include: manage.yml name=project-vm-worker2 type=e2-standard-2 group=workers
   ↪  static_ip=false
17 - include: manage.yml name=project-vm-worker3 type=e2-standard-2 group=workers
   ↪  static_ip=false
18 - include: manage.yml name=project-vm-worker4 type=e2-standard-2 group=workers
   ↪  static_ip=false
19
20 - name: Delete NFS Storage Disk
21   include: delete-nfs.yml
```

Vamos então analisar este ficheiro. Este ficheiro diz, numa primeira fase, para o *Ansible* proceder à criação de um *Network File Storage Disk*, onde irão ser guardados ficheiros estáticos pelo *Misago*. A ordem desta *task* é importante: é necessário que apareça antes da criação das máquinas virtuais em si, e é também necessário que apareça antes da *task* que irá apagar o disco. É de realçar que esta *task* não está marcada com uma *tag*: as marcações de *tags*, nesta role, aparecem diretamente nas *tasks* que irão ser executadas. Isto é feito para fazer com que seja possível indicar ao *Ansible* exatamente quais as *tasks* que queremos efetuar com quais *tags*.

Como podemos ver, a definição de uma máquina virtual é simples: para criar uma máquina, basta criar uma tarefa que inclui o ficheiro *manage.yml*, e especificando os parâmetros da máquina necessários (*name*, *type*, *group*, *static_ip*).

O ficheiro *manage.yml* é depois responsável por importar os ficheiros *create.yml*, *delete.yml*, *start.yml* ou *stop.yml*, dependendo se a *tag* selecionada pelo utilizador é *create-vms*, *delete-vms*, *start-vms* ou *stop-vms*, respetivamente. Cada um dos ficheiros referidos depois define as *tasks* necessárias para proceder à ação especificada (i.e., o ficheiro *create.yml* contém as tarefas necessárias para criar uma VM na *Google Cloud*, o ficheiro *delete.yml* contém as tarefas para apagar uma VM, e igual para os ficheiros *start.yml* e *stop.yml*).

Com este sistema, para proceder à criação das VMs na *Google Cloud*, basta executar:

```
ANSIBLE_CONFIG=ansible.cfg ansible-playbook playbook.yml ---tags "create-vms"
```

Para proceder à sua destruição, início, ou encerramento, basta executar, com as devidas adaptações:

```
ANSIBLE_CONFIG=ansible.cfg ansible-playbook playbook.yml --tags
↪  "<delete,start,stop>-vms"
```

A eliminação das máquinas virtuais também procede à eliminação do disco de rede criado para os ficheiros estáticos do *Misago*, como se pode verificar no final do ficheiro de configuração das máquinas virtuais.

3.1.2 Gestão do inventário após a criação das máquinas virtuais

Cada VM criada pelo processo descrito anteriormente é alvo de uma *tag* colocada no próprio *Google Cloud*, de modo a identificar qual é o tipo da mesma: é uma VM de monitorização, *Swarm Manager*, ou *worker*?

Esta *tag* é depois utilizada pelo *Ansible* para identificar corretamente a qual grupo de hosts a VM pertence.

Assim, é possível o *Ansible* verificar quais as VMs já criadas, e pode depois efetuar o provisionamento ou efetuar alterações nas VMs, sem ser obrigado a verificar a sua existência uma a uma.

3.1.3 Instalação do software necessário

As tarefas de provisionamento das máquinas estão divididas em várias roles, sendo que cada role é executada em diferentes grupos de *hosts*:

- ***Role manage-vms*** - É a role responsável por gerir o inventário dinâmico. O seu funcionamento já foi explicado;
- ***Role common*** - Esta role é executada por todos os hosts, e contém tarefas comuns a todos, como a atualização dos repositórios do sistema, a instalação de pacotes necessários para o bom funcionamento do *Ansible*, e é também responsável pela montagem do disco NFS;
- ***Role install-monitoring-tools*** - É a role responsável por efetuar a instalação das ferramentas de monitorização (*Elasticsearch* e *Kibana*) na VM responsável por essa instalação;

- ***Role install-docker*** - É a role responsável pela instalação do *Docker Engine* em todas as VMs que irão executar *containers*;
- ***Role install-stack*** - É a role responsável por instalar os ficheiros do *Docker Stack* nas VMs que irão servir como *Swarm Managers*.

Existem também duas *tasks* soltas, presentes no *playbook*, que têm funções específicas:

- **Task 1** - É uma *task* solta que corre em todas as VMs marcadas como *worker*, cujo objetivo é juntar estas VMs ao *Docker Swarm* criado anteriormente;
- **Task 2** - É uma *task* solta que corre apenas num dos *managers* e cujo objetivo é inicializar o *Docker Stack*, e, consequentemente, a aplicação.

3.1.4 *Scaling* dos serviços

Por fim, existe um *playbook* diferente, *scale.yml*, cujo objetivo é efetuar o *scaling* dos serviços que são passíveis de sofrer *scaling*: *web*, *celery* e *misago*. Assim, para efetuar o escalonamento dos serviços, basta efetuar o seguinte comando:

```
ANSIBLE_CONFIG=ansible.cfg ansible-playbook -i hosts.gcp.yml scale.yml -e
↪  "web=3 celery=1 misago=3"
```

Este comando, tal como é evidente, altera o número de réplicas do serviço *web* para 3, o número de réplicas do serviço *celery* para 1, e o número de réplicas do serviço *misago* para 3.

É possível omitir qualquer um destes componentes, selecionando apenas aqueles que se quer alterar, sem alterar os restantes:

```
ANSIBLE_CONFIG=ansible.cfg ansible-playbook -i hosts.gcp.yml scale.yml -e
↪  "web=3"
```

Este comando apenas irá alterar o número de réplicas do serviço *web*.

3.2 *Docker / Docker Swarm*

De forma a simplificar e agilizar os processos de automatização da instalação de toda a infraestrutura, decidiu-se utilizar, em primeira instância, *containers*, tendo em conta que nos permitem ter ambientes isolados e leves para cada uma das camadas da aplicação. A utilização de *Docker containers* permitiu-nos também colocar os nossos tópicos de configuração em *templates*, mais simples de utilizar, adaptar e migrar, caso necessário.

Contudo, a simples utilização de *Docker* não servia os nossos requisitos de orquestração de serviços. É nesta parte que entra o *Docker Swarm*. O objetivo desta ferramenta é a orquestração de serviços, e

através da mesma conseguimos, com recurso a ficheiros *YAML*, construir toda a nossa infraestrutura de serviços.

Desta forma, decidimos dividir cada uma das camadas num serviço próprio, que é executado em *containers* próprios e isolados uns dos outros, pelo que tudo o que será apresentado nas próximas secções, por uma questão de simplicidade de exposição e leitura, abstrai-se do facto de ter sido tudo construído com imagens e *containers Docker*.

Por fim, de forma a ter um *Docker Environment* mais eficiente, após concluídas, as imagens de cada um dos serviços foram construídas (fazendo o seu *build*) e publicadas no registo público do *Docker*, *Docker Hub*, (através do comando *docker push*), para que, quando fossem necessárias, ser apenas preciso fazer o respetivo *pull*.

3.3 Nginx

Tal como é referido no relatório da fase anterior, e tal como é possível observar na figura 2.2, os clientes não comunicam diretamente com a camada aplicacional: existe uma camada (*web*) no meio. Esta camada tem a dupla funcionalidade: *load balancing* e *reverse proxy*, e foi implementada através de servidores *Nginx*.

Para além disto, a camada *web* tem mais algumas responsabilidades. Sendo o servidor aplicacional em *Django*, este, quando *deployed*, deve ter a sua configuração como *DEBUG=False*, o que significa que o seu servidor não é capaz de servir ficheiros estáticos, nem o deve fazer. Assim, decidiu-se utilizar um sistema de ficheiros distribuído (*NFS*) para guardar os mesmos. Deste modo, para acelerar o consumo desses recursos, tanto os servidores aplicacionais têm acesso ao NFS (de leitura e escrita), como os servidores *web* também têm (apenas de leitura). Com esta solução e configurando o *Nginx* para tal, ao detetar um pedido de um ficheiro estático, o *Nginx* responde diretamente com o ficheiro em vez de encaminhar esse pedido para os servidores aplicacionais, reduzindo os tempos de resposta associados a este tipo de pedidos e reduzindo a carga dos servidores aplicacionais.

Ficheiro *deployment/docker/web/misago*

```
40  location /static/ {
41      # gzip
42      gzip on;
43      gzip_static on;
44      gzip_disable "msie6";
45      gzip_vary on;
46      gzip_proxied any;
47      gzip_comp_level 6;
48      gzip_buffers 16 8k;
49      gzip_http_version 1.1;
```

```
50      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;
51      expires 365d;
52      add_header Cache-Control "public, no-transform";
53      alias /static/;
54  }
```

Por fim, de forma a conseguirmos ver como a camada se está a comportar (tanto esta como a camada aplicacional), decidimos aplicar monitorização nos servidores de *Nginx*. Para isso usámos dois tipos de monitorização: métricas e *logs*. Para monitorizar as métricas associadas aos servidores recorremos ao ***Metricbeat***, e recolhemos tanto as métricas de sistema como as métricas recolhidas pelo próprio plugin do *Metricbeat* para *Nginx*. Para a recolha e análise dos *logs*, usámos o ***Filebeat***, mais uma vez através do próprio plugin que o *Filebeat* fornece para o *Nginx*.

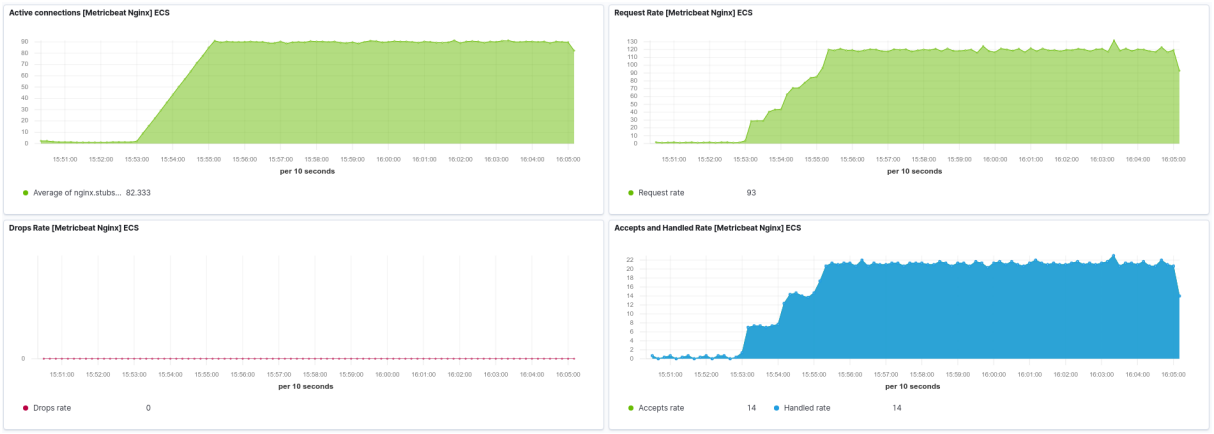

Figura 3.1: Monitorização do *NGINX* - *Metricbeat*

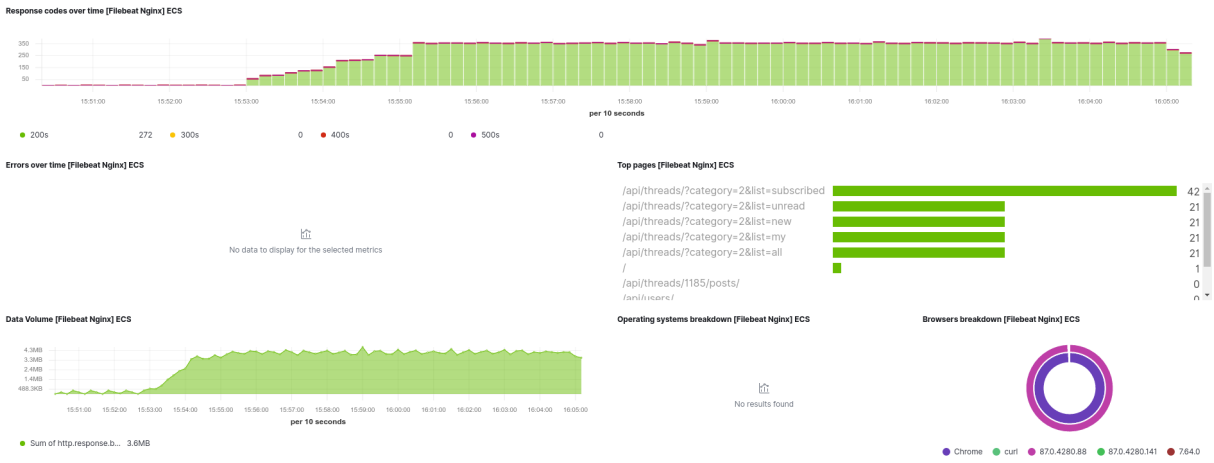

Figura 3.2: Monitorização do *NGINX* - *Filebeat*

As figuras 3.1 e 3.2 demonstram o que se consegue obter e analisar através das ferramentas de monitorização utilizadas.

3.4 Misago

A camada ***Misago*** representa a camada aplicacional de toda a infraestrutura. Como já foi referido, esta camada está desenvolvida em *Django* e como tal, para ter um *deployment* adequado, é ncessário, na sua configuração, mudar a variável *DEBUG* para *False*. Como contexto, o *Django*, para facilitar o desenvolvimento, traz incorporado um servidor bastante básico e pouco eficiente. Ao mudar aquela variável está-se a desativar esse servidor, o que significa que é necessário configurar um servidor aplicacional para correr a aplicação. Para isso utilizou-se o ***Gunicorn***, que, segundo algumas referências ([4], [5]), é um dos mais rápidos e eficientes servidores compatíveis com aplicações *Django*.

Abaixo é possível ver as configurações do servidor *Gunicorn* (desde o tipo de *logs* usados, o IP e porta de *bind* até ao número de *workers* por servidor).

Ficheiro *deployment/docker/misago/gunicorn.conf.py*

```python
import os

# https://docs.gunicorn.org/en/latest/settings.html#settings

# Variables
DJANGO_WSGI_MODULE = os.environ.get("DJANGO_WSGI_MODULE")
DJANGO_SETTINGS_MODULE = os.environ.get("DJANGO_SETTINGS_MODULE")

# Config File
wsgi_app = f"{DJANGO_WSGI_MODULE}:application"

# Debugging
reload = os.environ.get("RELOAD", False)

# Logging
access_log_format = "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"
accesslog = os.environ.get("ACCESS_LOG_FILE", "-")
errorlog = os.environ.get("ERROR_LOG_FILE", "-")
loglevel = os.environ.get("LOG_LEVEL", "debug")

# Server Socket
bind = "0.0.0.0:80"

# Worker Processes
workers = 5
worker_connections = int(os.environ.get("WORKER_CONNECTIONS", "1000"))


def when_ready(_):
    print("Server is ready")
```

3.5 Redis

Como descrito na primeira fase do relatório, a aplicação utiliza a base de dados *Redis* de modo a garantir *caching* e armazenar dados associados a tarefas assíncronas. Para garantir uma maior resiliência e distribuição de carga, decidimos então por implementar 3 réplicas de servidores *Redis*: um designado por *Master* e os restantes designados por *Slaves*. A comunicação entre o cliente/consumidor de dados (neste caso, a *Rest API*) e o *cluster* de base de dados é feita através de dois servidores *HAProxy*, que no fundo são dois servidores de *proxy*. Para que estes servidores sejam capazes de tomarem as decisões de distribuição de *queries*, estes fazem periodicamente consultas ao *Redis Sentinel*, que é responsável por atualizar as informações que tem acerca dos servidores *Redis*, mas também por realizar a monitorização dos respetivos servidores e administrar os nós do *cluster* desta base de dados, ou efetuar a eleição de um novo *Master*.

De forma a conseguirmos ver o desempenho de toda esta infraestrutura, decidimos aplicar monitorização nos servidores de *Redis*. Para tal, foi utilizada a ferramenta ***Metricbeat*** onde recolhemos as métricas associadas ao *plugin* do *Metricbeat* para o *Redis*.

Figura 3.3: Monitorização do Redis

Concluindo, temos então implementado todo o nosso plano inicial, ou seja, dois nós de servidores *proxy*, que comunicam com os servidores *Redis* e o cliente, e a respetiva monitorização destes servidores, para que toda a infraestrutura funcione com elevado desempenho e resiliência.

3.6 PostgreSQL

Tal como foi explicitado na primeira fase, a plataforma escolhida utiliza uma base de dados PostgreSQL que foi implementada no seu próprio *container* com base na imagem disponibilizada pelo *Docker*. Para efetuarmos a monitorização desta, recorremos à ferramenta *MetricBeats*, pois esta disponibiliza um módulo com várias métricas sobre os processos ativos, escritas e *queries* efetuadas à base de dados. Efetuámos várias tentativas de instalar a ferramenta *MetricBeats* no mesmo *container* que a base de dados, mas tal relevou-se impossível, apesar do facto de conseguirmos executar *scripts* de instalação e

ativação de um serviço no *container* da base de dados. Tais *scripts* apenas são executados uma única vez quando o serviço é montado, mas como achamos que o serviço de monitorização pode ter de ser ativado mais do que uma vez, optámos por fazer a instalação deste num *container* separado.

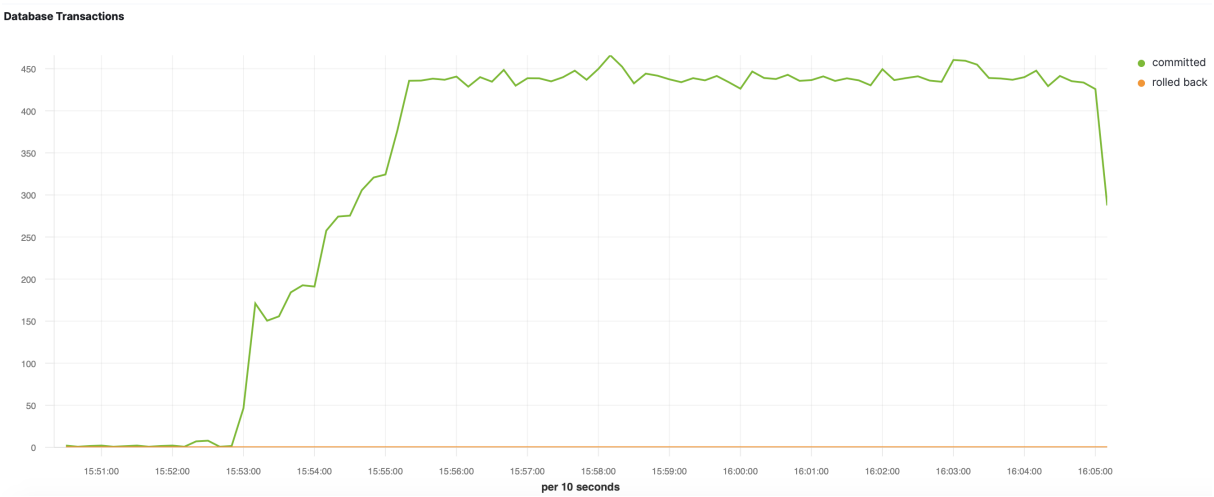

Figura 3.4: Monitorização da Base de Dados, Database Transactions - *Metricbeat*

Figura 3.5: Monitorização da Base de Dados, Rows Inserted Updated Deleted - *Metricbeat*

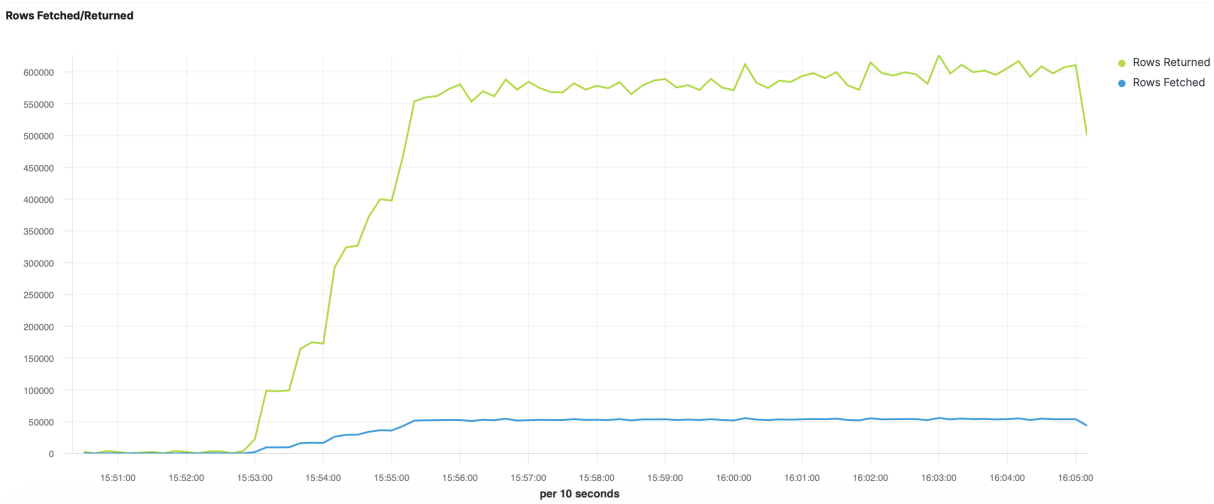

Figura 3.6: Monitorização da Base de Dados, Rows Fetched Returned - *Metricbeat*

3.7 Celery

O *Celery* é apenas uma *job queue* que o *Misago* utiliza durante a sua execução para efetuar tarefas que poderão demorar demasiado tempo. Isto ajuda o *Misago* a manter baixos os tempos de resposta, sem comprometer a funcionalidade da aplicação.

Como o *Celery* não tem estado interno, é simples de replicar: basta correr várias instâncias do *container* que o suporta, e portanto, é isso mesmo que estamos a fazer, ao correr duas réplicas do container que o executa. Se necessário, poderá ser ainda mais aumentado, pois é um dos serviços que permite a escala automática com recurso ao *Ansible*, tal como explicado na secção 3.1.4.

Capítulo 4

Resultados da Avaliação

De forma a fazer testes de carga ao sistema, recorremos ao **Apache JMeter**, onde desenvolvemos vários *test plans*. Inicialmente, crimos dois *test plans* que focam separadamente operações de leitura e operações de escrita. Estes testes não tentam fazer uma replicação da atividade de um utilizador, fazendo portanto várias operações de leitura sem espera entre estas, e operações de escrita no limite do permitido pela plataforma (existe um tempo de espera de 2 segundos entre operações de escrita). No entanto, estes testes não são inúteis, visto que têm o potencial para testar o sistema duma forma mais pesada do que os utilizadores reais conseguirão, mas tendo a desvantagem de não serem representativos da realidade. O *test plan* mais representativo (e que vamos aqui apresentar), tem um fluxo de operações balanceado entre operações de escrita e leitura, com alguns tempos de espera adequados, como iremos ver na figura 4.1, representativa do teste. De forma a ajudar na execução dos testes, foi também criado um script bash que ajuda na inicialização dos testes, visto que o *JMeter* necessita de vários parâmetros para iniciar um teste, deste o *path* do *test plan* até uma diretoria vazia para guardar o *output*. Desta forma, apenas é necessário escolher um dos *test plans* disponíveis, e o script trata de invocar este e gerar a pasta para o *output*.

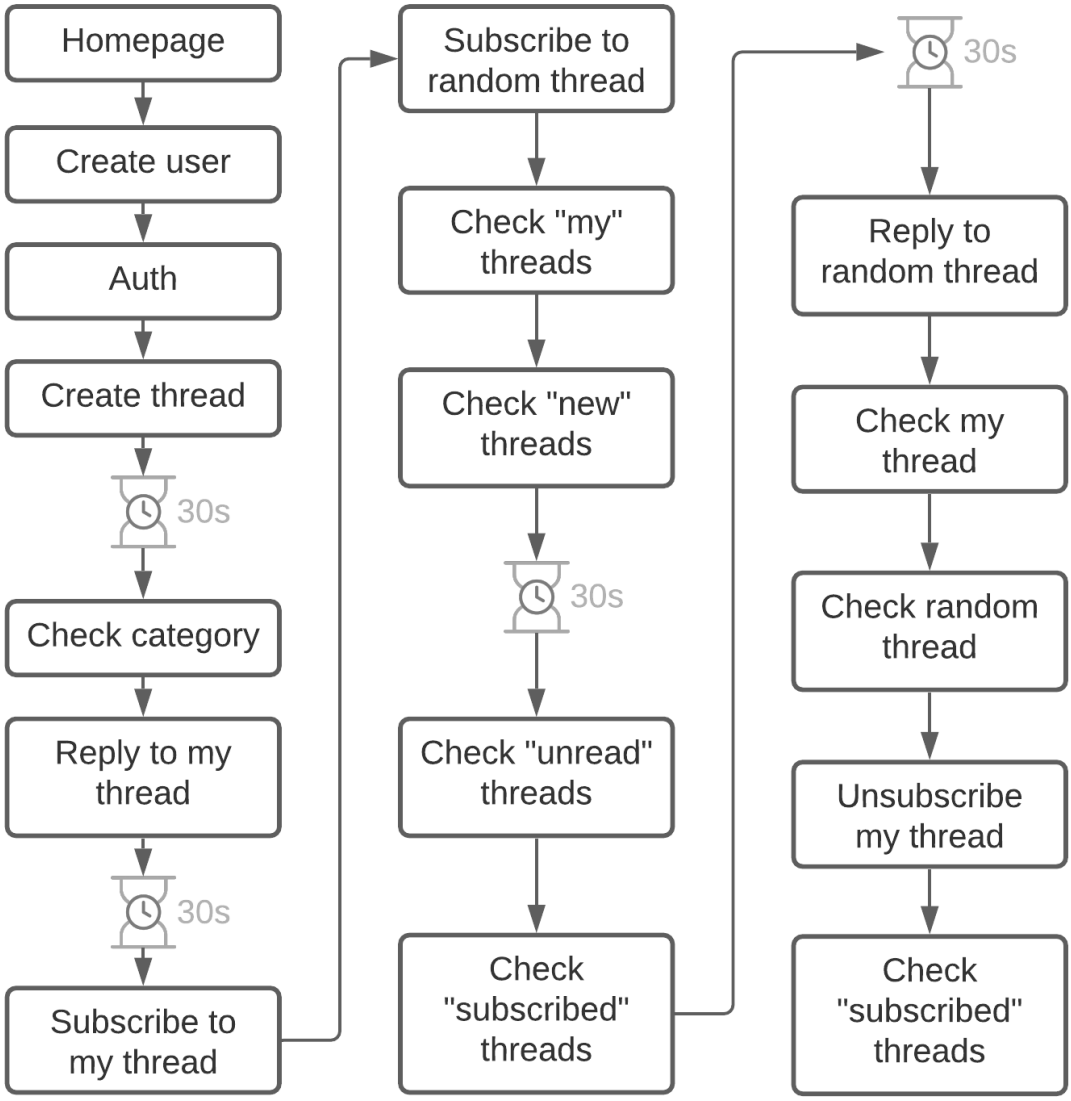

Figura 4.1: Fluxo do teste escolhido para teste de carga

Nos resultados que iremos apresentar a seguir, o teste foi executado com 1500 utilizadores, ou seja, 1500 *threads* independentes a seguir o fluxo apresentado na figura 4.1. Foi escolhido também um *ramp-up period* de 12 minutos, o que significa que o *JMeter* irá criar um novo utilizador sensivelmente a cada 2 segundos.

Load Testing Statistics					
Ação	**Execuções**	**Falhas %**	**Média tempo resposta**	**Mínimo tempo resposta**	**Máximo tempo resposta**
Homepage	1500	0%	375.39 ms	265 ms	1506 ms
Create random user	1500	0%	817.55 ms	594 ms	1616 ms
Auth	1500	0%	129.66 ms	88 ms	684 ms
Create thread	1500	0%	190.47 ms	123 ms	1114 ms
Check category	1500	0%	325.65 ms	230 ms	1429 ms
Reply to my thread	1500	0%	228.50 ms	144 ms	1345 ms
Subscribe to my thread	1500	0%	196.46 ms	142 ms	528 ms
Subscribe to random thread	1500	0%	150.22 ms	103 ms	570 ms
Check ”my” threads	1500	0%	174.95 ms	109 ms	1047 ms
Check ”new” threads	1500	0%	344.28 ms	234 ms	1442 ms
Check ”unread” threads	1500	0%	183.50 ms	118 ms	525 ms
Check ”subscribed” threads	3000	0%	179.1 ms	114 ms	408 ms
Reply to random thread	1500	0.13%	273.97 ms	177 ms	1479 ms
Check my thread	1500	0%	219.89 ms	133 ms	840 ms
Check random thread	1500	0%	221.18 ms	130 ms	729 ms
Unsubscribe my thread	1500	0%	146.27 ms	93 ms	640 ms
Total	*25500*	*< 0.01%*	*240.91 ms*	*93 ms*	*1616 ms*

Tendo em conta os dados apresentados acima, concluimos os testes com apenas 2 *requests* falhados (500/Internal Server Error), dando uma percentagem de sucesso superior a 99.99%. Além disso, os tempos de resposta também são bastante aceitáveis, tendo em conta o número de utilizadores que estão a ser simulados.

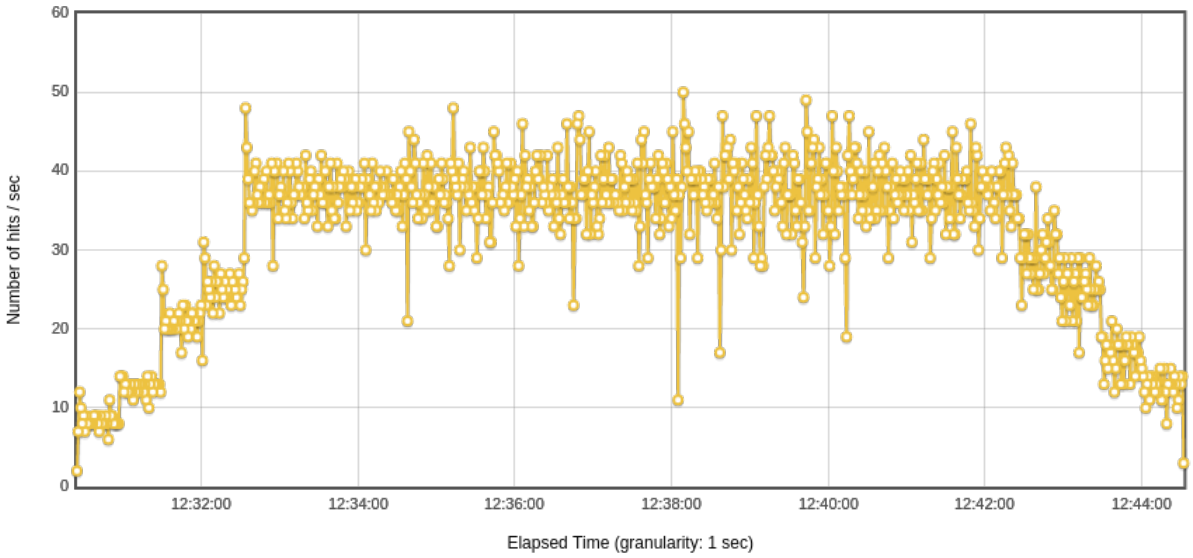

Figura 4.2: Hits per Second - HTTP requests enviados por segundo

Visto que os nossos testes estão a replicar o comportamento de um utilizador fazendo *requests* à *REST API*, os Hits per Second são uma boa métrica para analisar carga simulada por estes utilizadores.

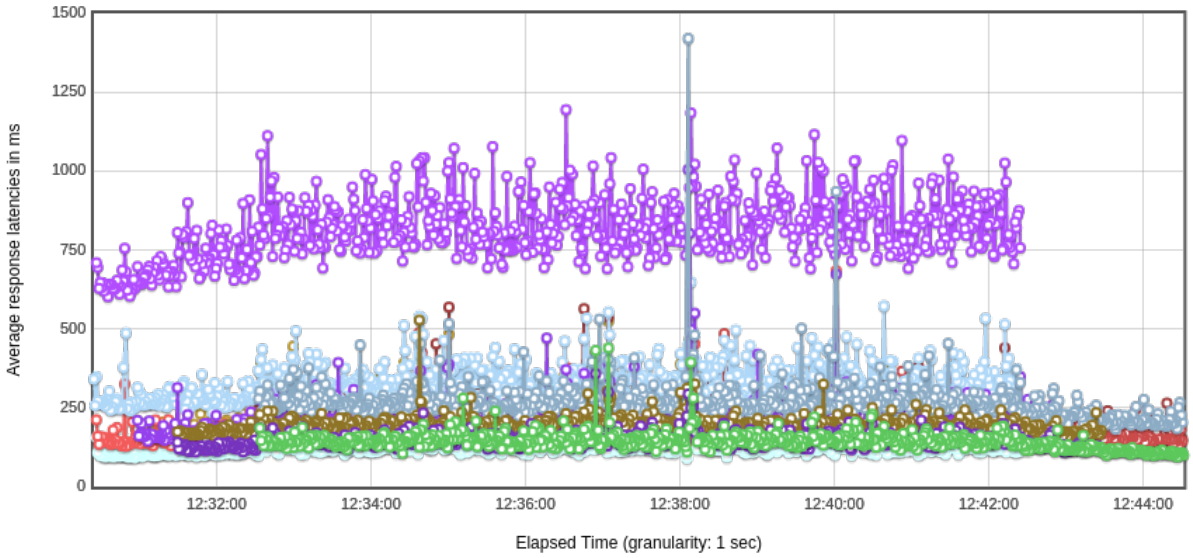

Figura 4.3: Latencies Over Time - Latência de operações

Relativamente à figura 4.3, os pontos a roxo correspondem à operação "Create random user", que são também as com tempo de resposta mais longo, de acordo com a tabela acima. As restantes operações, têm uma latência mais baixa, sem grandes flutuações no decorrer do teste.

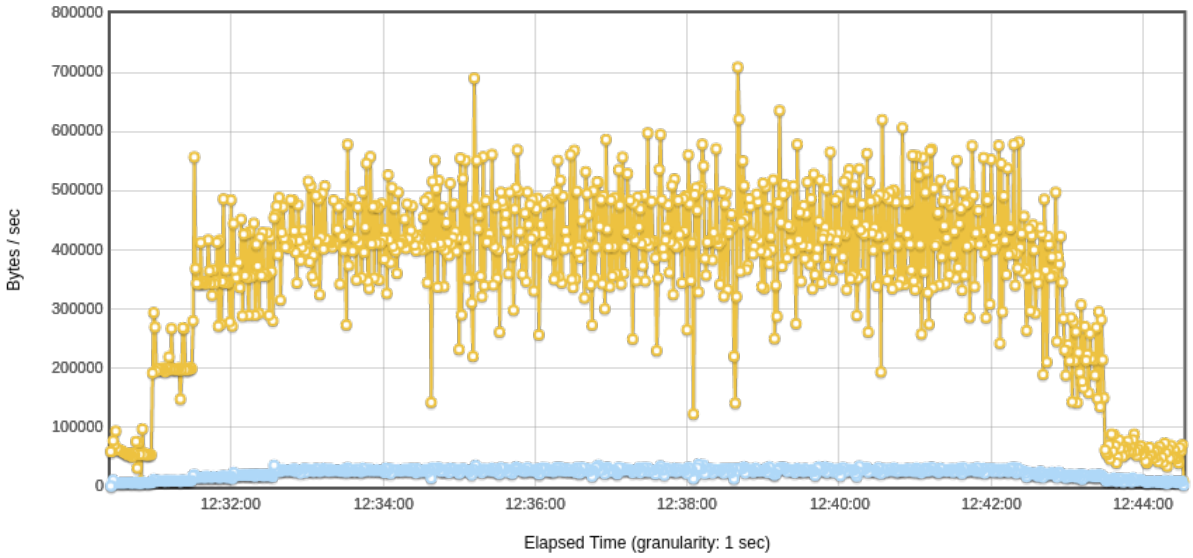

Figura 4.4: Bytes Throughput Over Time - Tráfego de dados

De acordo com este gráfico, conseguimos visualizar os dados enviados e recebidos ao longo da duração do teste. Como é esperado, este gráfico tem uma forma semelhante à do gráfico representado na figura 4.2, dando a entender que existe mais tráfego nos pontos com mais *requests*.

Tendo então em conta os dados fornecidos pelo teste, os resultados obtidos são bastante satisfatórios, visto que uma das métricas que mais afeta diretamente o utilizador é o *response time*, e os valores obtidos nesta categoria são baixos tendo em conta a carga que induzimos na plataforma. De qualquer das formas, caso a carga fosse levada mais ao extremo, o *bottleneck* mais provável que iríamos encontrar seria na comunicação com a base de dados PostgreSQL, visto que só temos uma instância desta num container *Docker*, enquanto que a camada aplicacional tem várias instâncias a processar pedidos.

Capítulo 5

Conclusão

Durante a realização deste trabalho tivemos a oportunidade de contactar com várias ferramentas de instalação, provisionamento, e monitorização de sistemas e aplicações distribuídas. Ficamos com um conhecimento das ferramentas *Ansible* e *Docker*, e também como utilizar a *Google Cloud* para fazer o *deployment* de uma aplicação distribuída.

Com este trabalho foi-nos possível efetuar o *deployment* de uma aplicação distribuída, o *Misago*, sendo que obtivemos uma instalação que nós consideramos ser altamente resiliente e escalável, como se pode verificar pelos valores de *performance* que obtivemos no capítulo 4.

Num trabalho futuro, consideramos como aspetos a melhorar a implementação de mais métricas de desempenho nos componentes que ainda não dispõem das mesmas, bem como uma melhor escalabilidade na base de dados *PostgreSQL*, de modo a poder efetuar esse escalamento de forma automática, tal como é possível efetuar nas componentes *Web*, *Celery* e *Misago* (camada aplicacional).

Bibliografia

[1] *Stateless Applications*, "https://www.redhat.com/en/topics/cloud-native-apps/stateful-vs-stateless", Acedido: 7-11-2020.

[2] *High availability and scalable reads in PostgreSQL*, "https://blog.timescale.com/blog/scalable-postgresql-high-availability-read-scalability-streaming-replication-fb95023e2af/", Acedido: 7-11-2020.

[3] *Building a scalable PostgreSQL solution*, "https://hub.packtpub.com/building-a-scalable-postgresql-solution/", Acedido: 7-11-2020.

[4] *A Performance Analysis of Python WSGI Servers: Part 2*, "https://www.appdynamics.com/blog/engineering/a-performance-analysis-of-python-wsgi-servers-part-2", Acedido: 01-01-2021.

[5] *Django Server Comparison: The Development Server, Mod_WSGI, uWSGI, and Gunicorn*, "https://www.digitalocean.com/community/tutorials/django-server-comparison-the-development-server-mod_wsgi-uwsgi-and-gunicorn", Acedido: 01-01-2021.