Universidade do Minho
Escola de Engenharia

Seleção, caracterização e análise de uma aplicação distribuída com vista à automatização do seu *deployment*

System Deployment & Benchmarking

Trabalho realizado por:

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

Janeiro de 2021

Índice

Lista de Figuras ii

Listagens iii

1 Introdução 1

1.1 Contextualização . . . 1

1.2 Estrutura do Relatório . . . 1

2 Apresentação da Aplicação 2

3 Análise da aplicação Misago 3

3.1 Arquitetura do Misago . . . 3

3.1.1 Descrição . . . 3

3.1.2 Componentes . . . 3

3.1.2.1 Frontend . . . 3

3.1.2.2 Rest API . . . 4

3.1.2.3 Bases de Dados . . . 4

3.1.2.4 Job Queue . . . 4

3.2 Padrões de distribuição . . . 4

3.3 Formas de comunicação . . . 5

3.4 Pontos de configuração . . . 7

3.5 Operações de desempenho crítico . . . 8

4 Instalação da Aplicação 9

5 Conclusão 11

Bibliografia 12

Lista de Figuras

3.1 Arquitetura básica Misago 3
3.2 Possível arquitetura de comunicação/*deployment* 7

Listagens

Ficheiro *.env*, linhas 1-30 7
Ficheiro *Vagrantfile*, linhas 22-33 9
Ficheiro *Vagrantfile*, linhas 35-50 10

Capítulo 1

Introdução

1.1 Contextualização

Na sequência da UC Complementar de *System Deployment & Benchmarking*, foi proposta a realização de um trabalho prático que consiste no *deployment* e *benchmarking* de uma aplicação distribuída num *Cloud Provider*, sendo que neste caso irá ser utilizada a *Google Cloud Platform*.

No entanto, antes de ser realizado o *deployment*, deveremos avaliar a aplicação escolhida (neste caso o *Misago*), e deve ser feita uma avaliação dos requisitos a cumprir durante a automatização do seu *deployment*.

Este relatório contém os resultados da avaliação desses mesmos requisitos, bem como aspetos importantes relacionados com todo este processo.

1.2 Estrutura do Relatório

Este relatório encontra-se dividido em 5 partes importantes.

A introdução faz uma contextualização do relatório e apresenta a sua estrutura básica.

O segundo capítulo faz uma breve apresentação da aplicação que foi escolhida (*Misago*), i.e., o que é, para que serve, casos de uso, etc.

No terceiro capítulo é feita uma análise da aplicação: é analisada a sua arquitetura, componentes, padrões de distribuição, formas de comunicação, pontos de configuração e operações de desempenho crítico.

No quarto capítulo é descrito o processo de instalação da aplicação com recurso ao *Vagrant*. São depois tiradas conclusões relativamente à instalação, que virão a ser úteis quando a aplicação tiver de ser instalada *(deployed)* na *Google Cloud*.

Por fim, na conclusão são retiradas as conclusões mais importantes do trabalho realizado nos restantes capítulos.

Capítulo 2

Apresentação da Aplicação

O Misago é um fórum *online open-source* desenvolvido maioritariamente em *Python*, através da *framework Django*. Criado há cerca de oito anos com o objetivo de servir comunidades de pequeno e médio tamanho e de ser facilmente integrado em plataformas já existentes que queiram fornecer este serviço aos seus utilizadores, tem todas as funcionalidades esperadas de um serviço deste género e destaca-se pelo alto nível de customização oferecido. Como permite modificar completamente o visual do fórum, definir quais as informações que queremos pedir aos utilizadores, aceita múltiplas formas de autenticação, entre outros, é uma aplicação popular para a criação de fóruns *online*.

Capítulo 3

Análise da aplicação Misago

3.1 Arquitetura do Misago

3.1.1 Descrição

A arquitetura pode ser dividida em 3 componentes principais: *Frontend*, *Rest API* e Bases de Dados. O *Frontend* é baseado em *React*, uma *framework* em *JavaScript*, a *Rest API* é baseada 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 subdivisão e a distribuição de cada um destes componentes é descrita mais à frente.

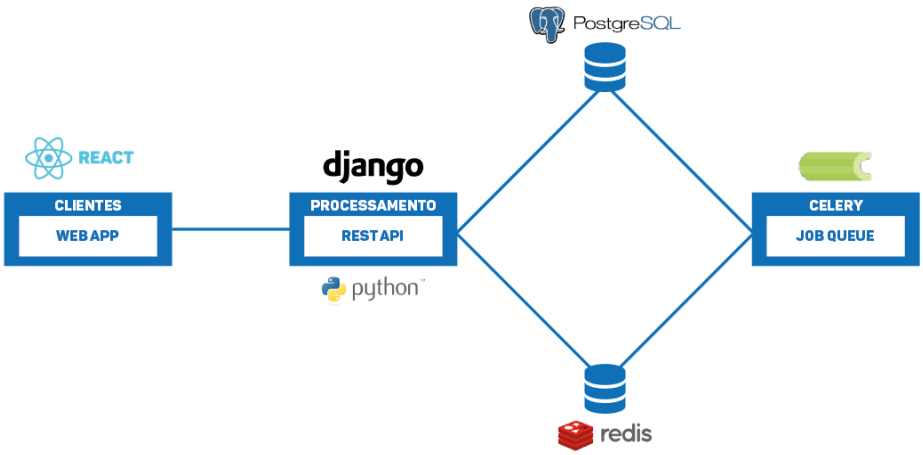

Figura 3.1: Arquitetura básica Misago

3.1.2 Componentes

3.1.2.1 Frontend

A maioria da *User Interface* foi desenvolvida em *React.js*, que é uma escolha bastante comum associada à *Rest API* em *Django*. É possível ainda desenvolver temas personalizados por cima do *Frontend*, de forma a que todos os fóruns não sejam exatamente iguais em diferentes *deployments* da plataforma.

3.1.2.2 Rest API

A *Rest API* está construída em ***Django***, e, tendo em conta o grande número de módulos existentes para as mais diversas necessidades neste *Framework*, todos os serviços do sistema se encontram encapsulados na *Rest API*, tais como:

- autenticação;
- envio de *emails* (sendo que aqui é possível configurar a utilização de serviços de *email* externos);
- ferramentas de estatística;
- controlo de utilizadores;
- controlo e CRUD de categorias, *posts* e *threads* associadas ao fórum, entre outras funcionalidades.

3.1.2.3 Bases de Dados

Esta aplicação faz uso de duas bases de dados, uma base de dados relacional, *PostgreSQL* para guardar toda a informação persistente como as *threads* e as publicações nestas efetuadas. Usa também uma base de dados *Redis* para fazer *caching* e assim guardar os resultados de operações custosas para os poder reusar no futuro, cortando assim tráfego para a base de dados relacional. Além disso esta base de dados armazena informação relativa a tarefas assíncronas a executar.

3.1.2.4 Job Queue

A aplicação Celery é uma *job queue* utilizada para executar as tarefas assíncronas armazenadas na base de dados Redis.

3.2 Padrões de 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.

Relativamente ao **servidor aplicacional**, para este ter um maior desempenho 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*, tal como é discutido nas referências [2, 3], concluímos que é necessário garantir replicação de dados (de forma a não perdermos dados, caso exista um falha, por exemplo num disco), e também garantir alta disponibilidade e desempenho. Para tal, decidiu-se aplicar uma distribuição ***Master-Slave*** em que temos uma instância *PostgreSQL* que serve como *master*, onde são realizadas todas as operações de escrita, e várias outras instâncias, *slave*, que servem os pedidos de leitura à base de dados. Isto sempre com o mecanismo de *Streaming Replication* entre todas as instâncias, que o próprio *PostgreSQL* permite ter. Aliado a isto teríamos também, à semelhança da camada aplicacional, 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*.

3.3 Formas de comunicação

Tendo em conta as estratégias de distribuição e a arquitetura do sistema, tal como se pode ver na figura 3.1, o sistema tem 3 principais vias de comunicação: o cliente comunica com o servidor aplicacional e, quando necessário, este comunica com os servidores de *PostgreSQL* e de *Redis*. Ao introduzir as diferentes estratégias de distribuição, esta rede de comunicação já não é tão simples.

Assim, na topologia geral, o cliente passa a comunicar com o servidor *proxy* que distribui a carga para as diferentes réplicas aplicacionais. Estas, por sua vez, comunicam com os servidores *proxy* responsáveis por distribuir a carga das bases de dados *PostgreSQL* e *Redis*.

Na camada aplicacional, não há grande necessidade de uma estrutura interna muito complicada. Assim, é simples ter-se várias réplicas que são designadas pelo servidor de *proxy* que vai distribuindo a carga entre elas.

Relativamente aos servidores de base de dados transacionais (*PostgreSQL*), como dito acima esta segue uma arquitetura do tipo *Master-Slave*, onde estarão presentes dois tipos de servidores: o *master*, que fica com o cargo de leitura e escrita na base de dados, sendo a escrita exclusiva a este servidor; e os

slaves, que terão um papel essencial no sentido em que garantem a rápida leitura de dados, na base de dados.

Para que este *cluster* de base de dados represente um estado consistente dos dados é essencial ter um mecanismo de replicação entre os vários nós do *cluster*. Para o efeito, é válido utilizar a funcionalidade de *Streaming Replication*, disponibilizada pelo próprio *PostgreSQL*, que permite distribuir, de forma contínua, os *records* do *WAL (Write Ahead Log)* do *master* pelos *slaves* do *cluster*.

Contudo, há ainda dois aspetos que falta analisar relativamente ao *cluster* de base de dados transacional, a distribuição de carga e eleição do *master*, caso o atual falhe. Estes dois problemas são passíveis de ser resolvidos com a utilização de ferramentas próprias para o *PostgresSQL*, como o *Pgpool-II* - um *middleware* que atua entre os clientes da base de dados e o *cluster* onde estão armazenados os dados. Este género de ferramentas funcionam tanto como servidor de *proxy*, como órgão de decisão em *clusters* de base de dados. No caso específico do *Pgpool-II*, este permite:

- *Load balancing*: dispõe de mecanismos de balanceamento de carga, que permite distribuir as *queries* pelos vários nós do *cluster*, tendo em contas as restrições relativas ao tipo de operações que cada nó pode realizar.

- Decisão após falha de nós: toma decisões, tais como, reeleição do *master* ou bloqueio do envio de *queries* para nós desativados ou com falhas.

No caso dos servidores de base de dados *Redis*, é possível utilizar uma estrutura de comunicação semelhante ao do *cluster* de *PostgreSQL*, tendo também em conta que a arquitetura base é semelhante. Como ferramentas próprias a utilizar, temos como possibilidade a inter-operação do *Redis Sentinel* para monitorizar e administrar os nós do *cluster* de base de dados *Redis* (eleger novo nó *master*, por exemplo) e o *HAProxy* para servir como servidor de *proxy* para mediar a comunicação entre o cliente consumidor de dados (neste caso, a *Rest API*) e o *cluster* de base de dados. Para que o *HAProxy* tome as decisões corretas de distribuição de *queries* este deverá periodicamente consultar o *Redis Sentinel* de cada nó do cluster e atualizar as informações que tem acerca do mesmo de acordo com os dados dos *Redis Sentinel*. Todas as decisões tomadas neste contexto tiveram como auxilio as seguintes referências [4, 5]

Para as instâncias do *Celery*, onde é necessária a comunicação com os servidores *Redis* e *PostgreSQL* para que todas as tarefas a serem executadas sejam feitas com sucesso. Com, pelo menos, duas instâncias *Celery*, conseguimos distribuir a carga que as tarefas representam pelas várias instâncias.

Na figura 3.2 é possível ver o exemplo da arquitetura de distribuição e comunicação descrita acima.

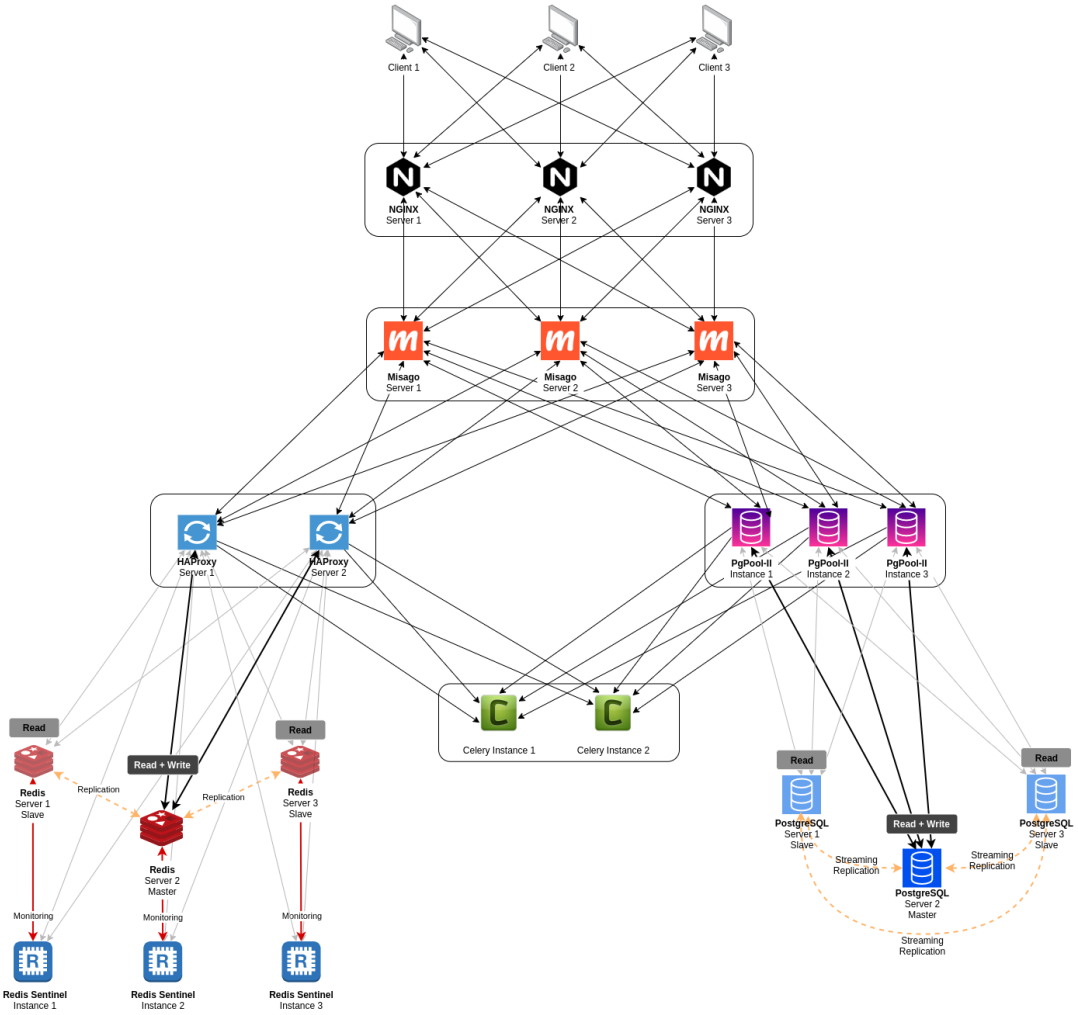

Figura 3.2: Possível arquitetura de comunicação/*deployment*

3.4 Pontos de configuração

Esta aplicação não tem muitos pontos de configuração: apenas a API REST necessita de uma grande configuração. Esta configuração é realizada com ficheiros *.env*, ou seja, com variáveis de ambiente:

Ficheiro *.env*

```
POSTGRES_DB=misago
POSTGRES_USER=misago
POSTGRES_PASSWORD=secret
POSTGRES_HOST=postgres
POSTGRES_PORT=5432

REDIS_HOST=redis
REDIS_PORT=6379
REDIS_PASSWORD=alsosecret
```

De resto, é obviamente necessário criar o utilizador para a base de dados, bem como a base de dados em si onde a aplicação irá guardar os seus dados. O *Misago* é capaz de efetuar migrações na base de

dados para depois criar as tabelas necessárias ao seu funcionamento.

3.5 Operações de desempenho crítico

Em qualquer aplicação deste género, a falha ou o mau desempenho de qualquer um dos pontos poderá provocar uma perda no número de utilizadores da aplicação. No entanto, existem certos pontos cujo desempenho é mais importante do que outros.

No caso desta aplicação, existe um ponto cujo desempenho é de importância crítica: a base de dados. Isto deve-se ao facto de ser a base de dados que faz a sincronização de todo o estado da aplicação; e quase sempre que um utilizador precisa de efetuar uma ação no fórum, vai ser sempre necessário pelo menos um acesso à base de dados.

Os restantes pontos não são tão críticos como a base de dados:

- Redis Cache: Se a cache tiver mau desempenho ou simplesmente falhar, os seus dados podem sempre ser calculados novamente a partir da base de dados. Podem também ser utilizadas estratégias de replicação ou de *clustering* para permitir a alta disponibilidade e alto desempenho desta cache.

- Django REST API: Se a API REST falhar, visto que esta não tem estado interno, podemos sempre utilizar *load balancing* e ter depois vários servidores a correr a API REST. Assim, não existe um *single point of failure* que irá fazer com que a aplicação inteira fique indisponível se apenas este ponto ficar indisponível. A sincronização das várias instâncias da API são feitas com recurso à base de dados.

O facto de a base de dados ser sempre mencionada como o método para restaurar o funcionamento da cache e da API REST caso estes tenham algum problema é novamente mais evidência a suportar a ideia de que a base de dados é realmente o ponto mais crítico desta aplicação.

Assim sendo, é de extrema importância que a base de dados utilize mecanismos como *streaming replication* e *clusters* com *load balancing* para garantir a alta disponibilidade, alta fiabilidade e alta tolerância a falhas da mesma.

Capítulo 4

Instalação da Aplicação

Para ser possível ter uma perspetiva da modularidade de todo o sistema associado ao Misago, fez-se uma instalação distribuída relativamente simples. Para tal, foram utilizadas as ferramentas aprendidas nas aulas até ao momento, tais como, *Vagrant* e conceitos de aprovisionamento. Para além disso, é também de referir que para o desenvolvimento desta instalação foram consultadas as referências de instalação presentes em [6, 7].

Deste modo, fez-se uma instalação distribuída por duas máquinas virtuais. **Uma** delas **contendo** os dois servidores de bases de dados: *PostgreSQL* (para toda a gestão transacional do sistema) e *Redis* (para cache e sessões) e a **outra máquina virtual** com o servidor aplicacional/web com a *interface* e *Rest API*. Isto permite que o cliente (*browser*) comunique com o servidor aplicacional (que se encontra numa das máquinas virtuais) e este, por sua vez, comunica, quando necessário, com as bases de dados (que estão na outra máquina virtual).

Os códigos 2 e 3 mostram as configurações específicas a cada uma das VMs, no *Vagrantfile*. De referir também que, para facilitar a execução do servidor aplicação, foi criado um serviço *Linux* para este.

Ficheiro *Vagrantfile*

```
22  config.vm.define "bd" do |subconfig|
23      subconfig.vm.hostname = "bd"
24
25    subconfig.vm.network "private_network", ip: "10.0.0.102"
26
27      subconfig.vm.provider "virtualbox" do |v|
28          v.name = "bd"
29      end
30
31      subconfig.vm.provision "file", source: "./config/dbs", destination: "/tmp/dbs"
32      subconfig.vm.provision "shell", inline: "cd /tmp/dbs && /bin/bash install.sh"
33  end
```

Ficheiro *Vagrantfile*

```
35  config.vm.define "server" do |subconfig|
36      subconfig.vm.hostname = "server"
37
38      subconfig.vm.network "private_network", ip: "10.0.0.101"
39
40      subconfig.vm.provider "virtualbox" do |v|
41          v.name = "server"
42      end
43
44      subconfig.vm.provision "shell", inline: "mkdir -p /opt/misago && chown vagrant:vagrant
        ↪  /opt/misago"
45      subconfig.vm.provision "file", source: "./misago/", destination: "/opt/misago"
46      subconfig.vm.provision "file", source: "./config/server/misago.service", destination:
        ↪  "/opt/misago/misago.service"
47      subconfig.vm.provision "file", source: "./config/server/.env", destination:
        ↪  "/opt/misago/.env"
48      subconfig.vm.provision "file", source: "./config/server/run.sh", destination:
        ↪  "/opt/misago/run.sh"
49      subconfig.vm.provision "shell", inline: "cd /opt/misago && /bin/bash run.sh"
50  end
```

Por fim, para a conclusão deste capítulo, uma configuração desta natureza, apesar de simples, permitiu-nos avaliar o nível de modularidade que o *Misago* comporta e garantir que este nível era suficiente para o que era requerido para o projeto em questão. De facto, o *Misago* suporta um alto nível de modularidade ao ponto de que todas as camadas podem estar em máquinas virtuais e ambientes distintos sem que isso comprometa o correto funcionamento do sistema.

Capítulo 5

Conclusão

De uma forma resumida, escolhemos o ***Misago***, que é um fórum *open-source*, que para além de satisfazer os requisitos do trabalho prático descritos no enunciado, é destacado pela sua modularidade e plasticidade na distribuição.

Relativamente à distribuição do sistema, escolhemos fazer replicação do servidor aplicacional, possibilitado pela implementação de um (ou possivelmente mais) *Proxy-Server*. No caso das bases de dados, tanto no caso do *PostgreSQL* como do *Redis*, optámos por uma distribuição *Master-Slave*, aliada a uma camada adicional *Proxy-Server*, para tratar de balanceamento e distribuição dos pedidos. Visto que a base de dados relacional é um dos pontos mais críticos na aplicação, como é referido na secção 3.5, é importante que esta esteja distribuída, através, por exemplo, da utilização da funcionalidade *Streaming Replication* do *PostgreSQL*, para que não haja um ponto único de falha. Quanto ao *Celery*, este também está preparado para uma distribuição com replicação, de forma a que possa existir execução de tarefas assíncronas em várias instâncias. Assim, chegámos a um resultado final em que temos uma arquitetura escalável e resiliente a um conjunto alargado de possíveis falhas.

No fim de deste relatório, esperamos ter atingido os pontos-alvo de explicar a seleção da aplicação ***Misago***, bem como uma breve caracterização e análise detalhada da arquitetura. Depois da instalação inicial e tendo em conta os requisitos expostos neste relatório, estamos confiantes de que o padrão de distribuição escolhido será uma solução viável para *deployment* e posterior *benchmarking*, como iremos demonstrar na próxima fase do projeto.

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] *Redis round robin HAProxy*. "https://medium.com/@mcrunix/build-a-redis-round-robin-balancing-cluster-using-haproxy-fbdf5ea16cd0". Acedido: 7-11-2020.

[5] *Redis Sentinel Documentation*. "https://redis.io/topics/sentinel". Acedido: 7-11-2020.

[6] *Misago Project Repository*. "https://github.com/rafalp/Misago". Acedido: 12-11-2020.

[7] *Misago Production Repository*. "https://github.com/rafalp/misago_docker". Acedido: 13-11-2020.