UNIVERSIDADE DO MINHO
Departamento de Informática
Mestrado integrado em Engenharia Informática

System Deployment and Benchmarking

Ghost

Grupo 9

Daniel Fernandes (a78377)
Maria Helena Poleri (a78633)
Mariana Miranda (a77782)

Dezembro 2018
Braga

Resumo

O presente documento tem o objetivo de caracterizar e analisar a aplicação distribuída de código aberto Ghost, assim como descrever o processo de instalação e configuração da mesma na Google Cloud Platform.

Irão-se descrever os padrões de distribuição e comunicação entre componentes que julgamos serem os ideais para um desempenho melhorado do sistema e para garantir que este se encontra sempre em pleno funcionamento. Posto isto, é importante discutir como foi feita a instalação e configuração automáticas na Google Cloud Platform, usando a ferramenta Ansible.

Pretende-se ainda descrever o processo de utilização do Elasticsearch como ferramenta de monitorização da mesma e efetuar uma avaliação da aplicação mediante o uso da ferramenta Apache JMeter.

Conteúdo

Capítulo 1

Introdução

O presente documento tem o intuito de fazer uma caracterização e análise da aplicação distribuída de código aberto Ghost e descrever o processo de instalação e configuração da mesma na Google Cloud Platform, dotando-a ainda de ferramentas de monitorização e avaliação.

O Ghost trata-se de uma plataforma de *blogging* moderna escrita em JavaScript, que depende de uma interface de utilizador minimal. Oferece as funcionalidades mais relevantes neste tipo de serviço, como a possibilidade de criar um *blog* colaborativo, planeamento de conteúdos, e um editor de publicações rico que permite não só a formatação de texto, mas também a inserção de blocos de conteúdo dinâmico como imagens, vídeos e áudio. Para além disso, é possível usar um editor *Markdown* para escrever as publicações. Hoje em dia, esta plataforma alimenta desde *blogs* pessoais a equipas de editores maiores, incluindo entre os seus clientes a **DuckDuckGo** e a **Mozilla**.

Após uma descrição inicial da arquitetura e dos vários componentes da aplicação, pretende-se perceber qual a melhor maneira de distribuir estes últimos por diferentes máquinas, tanto por questões de desempenho, paralelismo ou balanceamento de carga, assim como com vista impedir a falha do sistema mediante o uso de réplicas para os seus componentes críticos. Uma vez escolhido o padrão de distribuição ideal, é importante discutir quais os pontos de configuração necessários que garantem o bom funcionamento da aplicação.

Posto isto, pretende-se descrever o processo de instalação e configuração automáticas da aplicação na Google Cloud Platform. Para tal, foi usada a ferramenta Ansible, tendo-se escrito um *playbook* constituído por diversos *roles* com o intuito de correrem em diferentes máquinas na *cloud*.

Uma vez com a aplicação instalada de forma distribuída e achando essencial uma vigia ao sistema, discutimos como usar o Elasticsearch como ferramenta de monitorização e o Kibana como ferramenta de visualização dos dados obtidos. Pretende-se ainda discutir aqui acerca das métricas mais importantes a usar no contexto da aplicação de *blogging* Ghost.

De seguida, descrevemos o uso do Apache JMeter como ferramenta de avaliação da aplicação. Pretende-se expor os *workloads* usados como teste, assim

como fazer uma análise dos resultados obtidos a partir destes testes.

1.1 Estrutura do Relatório

Primeiramente, irá-se analisar a arquitetura da aplicação, nomeadamente a sua divisão em componentes e quais as ferramentas usadas (Capítulo 2). Pretende-se, de seguida discutir, quais componentes da aplicação são os mais críticos, cujo bom desempenho é essencial para que a aplicação corra como esperado (Capítulo 3). Com vista a correr a aplicação num contexto distribuído, pretende-se ainda abordar os padrões de distribuição e comunicação entre componentes (Capítulo 4) e ainda as configurações necessárias para tal (Capítulo 5), necessárias para o bom funcionamento e melhoria de desempenho da aplicação.

Posto isto, irá-se discutir o processo de instalação e configuração automática da aplicação na Google Cloud Platform (Capítulo 6), as ferramentas e métricas de monitorização consideradas (Capítulo 7) e o *benchmark* efetuado à aplicação (Capítulo 8).

Por fim, pretende-se discutir os resultados obtidos na avaliação experimental da aplicação (Capítulo 9).

Capítulo 2

Arquitetura da aplicação

O Ghost é uma aplicação com uma arquitetura simples, encontrando-se dividida no típico modelo de três camadas. Uma ilustração da sua arquitetura pode ser encontrada na Figura 2.1, onde são mostrados todos os componentes e ferramentas usadas. De uma forma geral, cada servidor age como um cliente da próxima camada.

Segue-se uma breve análise de cada camada e componente que constitui esta aplicação.

2.1 1ª Camada

2.1.1 Nginx

O Ghost faz uso do servidor *web* Nginx, mas este apenas atua como *reverse proxy*. Em geral, a sua função é aceitar conexões do exterior e conectá-las à porta onde a instância do Ghost está a correr.

O Nginx pode ainda atuar como balanceador de carga, com vista a otimizar a utilização de recursos, reduzir os tempos de resposta a pedidos e garantir esta resposta mesmo que haja uma falha num servidor aplicacional. Deste ponto de vista, para além de receber as conexões, o servidor deverá conectar ao servidor aplicacional ideal, de forma a distribuir igualmente o tráfego que chega pelos vários servidores aplicacionais.

2.2 2ª Camada

2.2.1 Ghost CORE

Este componente contém toda a camada de negócio da plataforma e está escrito em NodeJS. A sua função é atender os pedidos que recebe da primeira camada, mediante o acesso aos serviços da camada onde se encontra a base de dados.

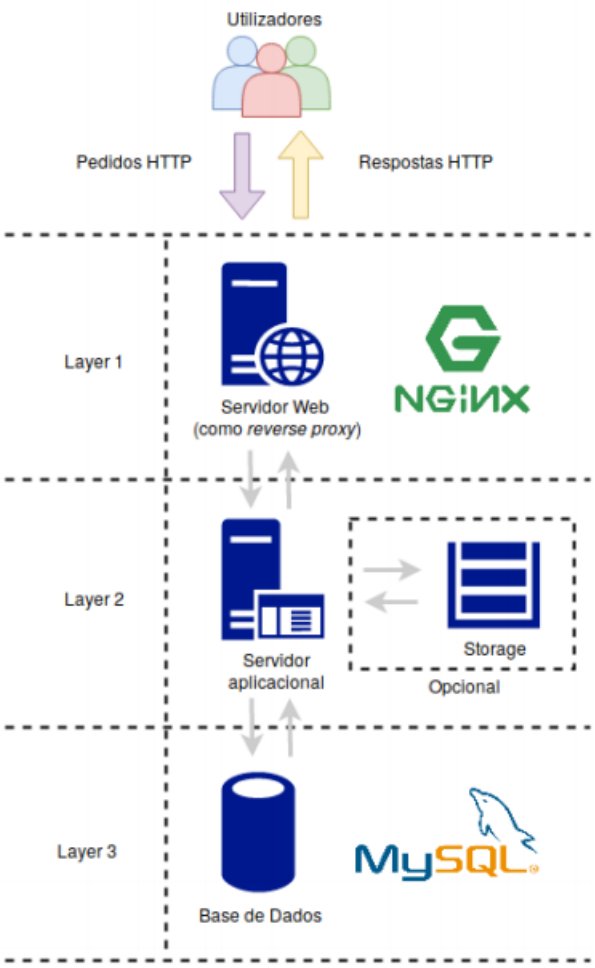


Figura 2.1: Infrastrutura do Ghost

É esta camada que, por exemplo, vai guardando as publicações dos utilizadores como rascunhos enquanto elas estão a ser escritas e as publica quando recebe o pedido da camada de cima para o fazer.

2.2.2 Storage Adapter

O Ghost utiliza, por defeito, o sistema de ficheiros local para, por exemplo, guardar imagens ou páginas HTML e Javascript necessárias para mostrar o *blog* no *browser*. No entanto, é possibilitada também a utilização de um sistema de ficheiros remoto.

As vantagens no uso de um sistema de ficheiros remoto em vez do local

são inúmeras. Sendo estes ficheiros essenciais para que o *blog* se apresente da maneira esperada, o facto de tê-los disponíveis remotamente pode constituir um plano de *backup*, mediante o uso de réplicas desta *storage* remota. Para além disso, com o crescimento dos *blogs* e consequentemente do número de ficheiros usados, os custos de um sistema de ficheiros local aumenta mais rapidamente que os custos da utilização de um sistema de ficheiros remoto. Assim, ganha-se em tolerância a faltas e poupa-se no custo de espaço de armazenamento.

2.3 3ª Camada

2.3.1 Base de Dados

A aplicação encontra-se suportada por uma base de dados, onde toda a informação do sistema é armazenada de forma persistente. Esta informação inclui todos os utilizadores do *blog* e as suas informações, todos as publicações de cada utilizador (tanto as publicadas como os rascunhos), *tags* e até as definições do *blog*. Para desenvolvimento, o Ghost usa o SQLite3, enquanto para produção é usado o MySQL. No entanto, como por cima desta camada existe uma camada ORM a usar o Bookshelf.js, é permitido usar um grande número de motores de bases de dados para além desses dois.

Numa instalação típica do Ghost, a base de dados MySQL estará instalada no mesmo servidor em que o Ghost está a correr, mas será sempre possível uma configuração em que isto não acontece.

Capítulo 3

Operações cujo desempenho é crítico

O Ghost, como qualquer outra plataforma de *blogging* moderna, centra a sua atividade na publicação e disponibilização de conteúdos desenvolvidos pelos autores do *blog*. Para tal, disponibiliza uma interface *web* na qual os utilizadores, quer produtores quer consumidores, interagem com o sistema. O *deployment* local do Ghost possui, contudo, uma particularidade, sendo esta a de permitir a existência de apenas um único *blog*.

Por forma a identificar as operações da aplicação cujo desempenho é crítico foi necessário analisar detalhadamente o sistema Ghost.

3.1 Abordagem

Inicialmente, numa perspetiva de mais alto nível, tivemos de perceber o que é que é e o que não é possível fazer do ponto de vista de cada utilizador. Numa perspetiva de mais baixo nível consideramos importante verificar como é que é realizada a distribuição dos dados, quer da própria aplicação quer dos utilizadores da mesma. Observamos ainda todo o *workflow* de pedidos desde o *browser*, na forma de mensagens *http*, passando pelos servidores *Web* e Aplicacional, até à sua possível tradução em interrogações à base de dados, quando despoletamos algum evento no *website*. A informação destes pedidos foi obtida através da consulta de *logs*, tanto do servidor *web* como da base de dados.

Dividindo a análise pelo tipo de utilizador considerado, tem-se que os leitores possuem essencialmente acesso a uma página principal onde se encontram incluídas todas as publicações que dizem respeito ao *blog*, podendo selecionar qualquer uma das que pretendam ler. Podem inclusive visitar as publicações de um dado autor ou associadas a uma determinada *tag*. Do ponto de vista de quem alimenta o *blog*, existe um único proprietário, mas que pode adicionar outras pessoas e atribuir-lhes diferentes cargos, com mais ou menos permissões. O proprietário e administradores tem acesso a várias configurações do *blog*, con-

tudo, as operações mais frequentemente realizadas serão efetivamente a escrita e atualização de publicações, sendo esta uma funcionalidade básica e que, portanto, está acessível a todo o tipo de colaboradores.

Posto isto, foi em torno da apresentação de conteúdo e escrita do mesmo que centramos a nossa atenção e estudo. As operações aqui envolvidas são potencialmente realizadas um número muito elevado de vezes, podendo, portanto, constituir gargalos no sistema.

Antes de analisar estas operações e para um melhor entendimento das mesmas, é importante recordar como se encontram distribuídos os dados do sistema. O Ghost faz uso de dois tipos de armazenamento, base de dados e *storage*. A base de dados é essencialmente utilizada para armazenar informação dos utilizadores e publicações, quer os que já foram publicados quer os que ainda se encontram inacabados, catalogados como rascunhos. A *storage*, por outro lado, é utilizada para armazenar as páginas *html* e ficheiros JavaScript necessários ao processamento das páginas *web*. Outros objetos, como imagens inseridas nas publicações, são também lá armazenadas, sendo referenciadas no *html* da publicação presente na base de dados.

Mediante esta organização já se consegue ter uma ideia dos componentes que terão um uso mais intensivo. A base de dados uma vez que inclui toda a informação das publicações será alvo de pedidos constantes por parte do servidor aplicacional. O servidor aplicacional possuirá também bastante carga porque será o intermediário desses pedidos e terá que construir e fazer o processamento das páginas. Analisemos agora em mais detalhe.

3.2 Escrita de publicações

Relativamente à escrita das publicações verificamos que quando um determinado autor pede para iniciar uma nova publicação, nenhum pedido é realizado ao servidor *web*, pelo que admitimos que a página já existe em *cache* no *browser*. Isto muda de figura quando o utilizador inicia a escrita da publicação, levando a que um pedido *http post* circule do *browser* para o servidor *Web*, deste para o servidor aplicacional, o que desencadeia uma grande quantidade de interrogações à base de dados. Estas interrogações têm como finalidade adquirir informação de permissões e tipo de utilizador, informações de sessão, inserção de uma nova publicação, associação de um novo utilizador à publicação, inserção de uma nova revisão da publicação na tabela das revisões, etc. O caso agrava sempre que é realizada uma ligeira pausa na escrita, onde desta vez um pedido *http put* flui outra vez do *browser* até ao servidor aplicacional que inicia novamente inúmeras interrogações com várias verificações, atualização da publicação e inserção de outra revisão.

Num cenário de um *blog* pequeno, com uma equipa de poucos colaboradores, facilmente um servidor aplicacional e uma instância da base de dados suportam os pedidos realizados. Contudo, se estivermos a falar de um dos *blogs* mais conhecidos do mundo, com uma equipa de 200 colaboradores a trabalharem nas mais variadas publicações, torna-se insustentável a carga exercida nestes

componentes. Trata-se, portanto, de uma operação de desempenho crítico.

3.3 Apresentação de conteúdo

A apresentação de publicações aos consumidores é naturalmente realizada em número muito mais elevado que a sua escrita. Utilizando a mesma forma de monitorização do sistema aplicada às escritas, verificou-se que também a apresentação de conteúdo requer alguma carga de trabalho sobre o servidor aplicacional e base de dados. Efetivamente, sempre que se acede à página inicial o servidor aplicacional lança pedidos à base de dados requerendo o conjunto das publicações a apresentar, ordenadas mediante várias propriedades, bem como as *tags* e autores relacionados com cada publicação. Estas interrogações são enviadas separadamente à base de dados e o servidor aplicacional tem o obrigação de processar estes dados e responder devidamente.

A escolha e apresentação de cada publicação individualmente é também muito custosa já que, mais uma vez, são realizados vários pedidos à base de dados, envolvendo, a grande maioria deles, um ou mais *joins* de tabelas bem como ordenações. Revivendo o cenário de milhares de utilizadores utilizar a aplicação em simultâneo, realizando pedidos para visualização de publicações, torna-se impossível para uma instalação do Ghost numa única máquina satisfazer todos os pedidos.

Capítulo 4

Padrão de Distribuição

Tal como mencionado anteriormente, e no primeiro cenário, onde a aplicação é utilizada por um número limitado de utilizadores, facilmente uma instalação do Ghost numa única máquina, à semelhança da realizada pelo grupo para efeitos de análise, seria suficiente para satisfazer todos os pedidos realizados. O padrão de distribuição, neste caso, seria, portanto, o mais simples possível onde efetivamente os componentes não se encontram distribuídos por várias máquinas, mas dispostos de forma centralizada numa única máquina.

Nesta máquina estaria a correr o Nginx, diretamente conectado ao Ghost CORE, isto é, à camada de negócio, que por sua vez acede à base de dados que suporta a aplicação. Apesar deste único serviço centralizado conseguir sem muito esforço satisfazer todos os pedidos para a quantidade de carga considerada e gastar poucos recursos, possui um grande problema que é o de constituir um ponto único de falha. Por esta razão e se quisermos garantir disponibilidade do serviço e tolerância a falhas é necessário que exista replicação, que exista uma diferente distribuição dos componentes do sistema.

No segundo cenário considerado, onde se dispõe a aplicação para milhares de utilizadores, tal distribuição é vista como obrigatória, já que provavelmente não é agora admissível que todo o sistema pare devido a uma falha, isto é, que se encontre indisponível, ou que proporcione más experiências aos leitores porque o serviço está lento, não conseguindo responder a todos os pedidos em tempo útil.

O padrão de distribuição a implementar deve atender a estes requisitos visando, contudo, a minimização dos recursos utilizados. Os gargalos do sistema identificados acima foram, portanto, a principal referência para a distribuição de componentes efetuada, pois nos permitiram decidir que componentes replicar e aqueles onde a replicação não é de todo vantajosa.

4.1 Replicação dos Servidores Web e Aplicacionais

Indo de encontro ao referido na secção inicial facilmente se conclui que a replicação é fundamental, sobretudo no que diz respeito ao núcleo lógico da aplicação. Uma única instância de servidor aplicacional ou *web* não é viável pelas razões já conhecidas de tolerância a faltas e desempenho.

Como também já mencionado em capítulos anteriores o servidor *web* no Ghost age essencialmente como *proxy*, encaminhando os pedidos que lhe chegam para um servidor aplicacional, pelo que este último é quem lida com grande parte da carga. O servidor Web pode ser também facilmente configurado para servir conteúdos estáticos e ainda realizar *caching* de pedidos anteriores, tirando carga do servidor aplicacional mas aumentando a sua própria carga. Uma vez que estes dois componentes equilibram-se bem, um ao outro, em termos de carga, consideramos ser uma boa escolha partilhar os recursos entre eles, isto é, inseri-los numa única máquina.

Ainda como vantagens, tem-se uma menor sobrecarga introduzida pela comunicação, pelo simples facto de se encontrarem na mesma máquina, bem como facilita a replicação. Caso o número de utilizadores aumente muito num período de tempo, facilmente se inserem novas réplicas lado a lado, caso diminua também se reduz para uma configuração com menos máquinas. Esta elasticidade horizontal permite ajustar os recursos do sistema às carga de trabalho e desta forma reduzir custos.

Claramente, tudo isto só é comportável, caso exista uma camada acima desta que balanceie a carga para cada uma das diferentes réplicas. Esta camada, no nosso caso, será implementada através do HAProxy.

4.2 Replicação da Base de Dados

A base de dados, como já referido anteriormente, é alvo de pedidos constantes devido ao facto de que a maioria das operações efetuadas no *blog* requerem isso. Desde a apresentação de conteúdo até à extração de informação de um utilizador, é inevitável que esta seja alvo de pelo menos uma *query*.

Sabendo isso, é fácil concluir que se existir uma falha na base de dados, então o *blog* não funcionará. Este argumento justifica uma replicação da base de dados, já que nesse caso se garantiria que existe sempre uma réplica em funcionamento, providenciando assim um mecanismo de tolerância a faltas efetivo.

4.3 Formas de Comunicação

Tendo em conta tudo mencionado anteriormente neste capítulo e no Capítulo 2, é importante agora discutir a forma como os vários componentes comunicam entre si.

Os componentes da camada 1, os servidores *web* conhecerão os clientes, dos quais recebem pedidos, e comunicarão com os servidores aplicacionais respetivos, aos quais enviam esses pedidos se necessário.

Os servidores aplicacionais da camada 2, por sua vez, recebem os pedidos provenientes da camada 1 e comunicam com as várias réplicas da base de dados, às quais enviam pedidos e das quais recebem respostas.

Na camada 3, as réplicas da base de dados comunicam com os vários servidores aplicacionais, respondendo aos seus pedidos.

Capítulo 5

Pontos de Configuração

Com o objetivo de assegurar o correto funcionamento desta aplicação, bem como o de usufruir de todas as funcionalidades que um sistema distribuído oferece, certos pontos de configuração que relacionam os diversos componentes entre si tornam-se imprescindíveis.

Pela Figura 2.1 e pelo discutido no capítulo anterior (Capítulo 4), facilmente podemos concluir entre quais integrantes da aplicação necessita de ser implementado um canal de comunicação. Para além destas, existem ainda configurações internas de cada componente a serem feitas, tais como um servidor de *e-mail* no servidor aplicacional, replicação da base de dados, entre outros. Estes vários pontos irão ser explicitados nos tópicos seguintes.

5.1 URL

Aquando a instalação do Ghost este requer um URL para o *blog*. Dado o modo como o *deployment* está a ser efetuado, este tem que ser o endereço IP da máquina onde se encontra o servidor aplicacional.

5.2 Base de Dados

A configuração da base de dados encontra-se dividida em 2 pontos essenciais: a sua replicação, com *failover*, e conexão dos servidores aplicacionais a esta. Iremos considerar um cenário com 2 instâncias a suportar a base de dados.

Em primeiro lugar, para permitir o acesso remoto de vários utilizadores é necessário, nas instâncias que irão suportar a camada de dados, criar as bases de dados pretendidas e um utilizador com todos os privilégios de acesso a esta, para que se possam ligar os servidores aplicacionais.

Dado que é importante que as bases de dados sejam dotadas de *failover*, isto é, caso o servidor *master* falhe, seja possível realizar interrogações ao *backup*, é necessário que a replicação seja do tipo *master-master*. Para tal, foi necessário alterar o ficheiro de configuração da base de dados para que qualquer alteração

passe a ser registada num ficheiro de *log*, a fim de ser lido pela outra instância, bem como, através da API disponibilizada pelo Ansible para replicação de servidores MySQL, configurá-los de modo a que cada um seja *slave* e *master* do outro.

Uma vez replicadas as bases de dados, é importante garantir tolerância a faltas, pelo que recorremos à ferramenta Keepalived e a um endereço IP virtual, criado em cada uma das instâncias que suportam a base de dados.

O Keepalived é um serviço que monitoriza servidores e processos, de modo a garantir alta disponibilidade. Abaixo apresenta-se o esqueleto do ficheiro de configuração do Keepalived.

Extrato 5.1 Esqueleto do ficheiro de configuração do *Keepalived*

```
vrrp_script mysql {
    script "/bin/pidof mysqld"
    interval 2
}

vrrp_instance floating_ip {
    state {{ state }}
    interface ens4
    track_script {
        mysql
    }
    unicast_src_ip {{ node_int_ip }}
    unicast_peer {
        {{ other_node_int_ip }}
    }
    virtual_router_id {{ keepalived_router_id }}
    priority {{ priority }}
    authentication {
        auth_type PASS
        auth_pass yourpassword
    }
    notify_master /etc/keepalived/master.sh
    notify_backup /etc/keepalived/backup.sh
}
```

Neste caso irá monitorizar o servidor MySQL presente na instância, para que perante a falha do *master*, o servidor *backup* torne-se *master*. Dado o contexto em que aplicação está a ser instalada, só é possível usar comunicação *unicast* entre os 2 servidores de dados.

Se por exemplo, no servidor *backup* notar-se que o servidor *master* MySQL foi abaixo, este torna-se em *master* e invoca o *script master.sh*, e o *backup.sh* na instância em que o serviço falhou.

Este trata primeiro de mover a rota para o endereço flutuante do *master* para o *backup* e cria uma regra de redirecionamento na *firewall*, para que seja

possível reencaminhar pedidos direcionados ao IP virtual para o IP interno da instância.

Extrato 5.2 Excerto do *script master.sh*

```
#!/bin/bash
gcloud compute routes delete floating --quiet
gcloud compute routes create floating \
--destination-range {{ keepalived_shared_ip }}/32\
--network sdb-grupo9-network\
--priority 500\
--next-hop-instance-zone us-east1-b \
--next-hop-instance {{ ansible_hostname }} --quiet
...
```

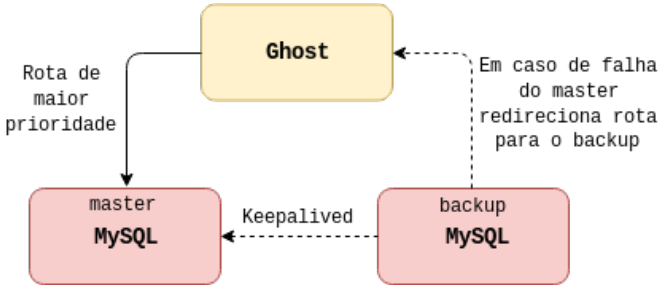


Figura 5.1: Funcionamento do *Keepalived*

E no *backup.sh* apaga a regra de redirecionamento do endereço flutuante.

Extrato 5.3 Excerto do *script backup.sh*

```
#!/bin/bash
iptables -t nat -D PREROUTING -d {{ keepalived_shared_ip }}
    -p tcp -j REDIRECT
```

É de notar que é importante garantir que o endereço flutuante não faz parte de nenhuma sub-rede e configurar a instância de modo a que esta consiga receber pacotes com um destino diferente ao seu IP, bem como permitir acesso a todas APIs pela conta de serviço em uso, caso contrário não terá permissões para criar e destruir rotas.

Do lado do CORE é preciso então, em cada um deles, associar esta base de dados, seguindo a seguinte estrutura de configuração:

Extrato 5.4 Esqueleto de configuração da base de dados

```
"database": {
  "client": "mysql",
  "connection": {
    "host": "{{ keepalived_shared_ip }}",
    "user": "{{ username }}",
    "password": "{{ password }}",
    "database": "{{ dbname }}"
  }
}
```

5.3 Servidor de *e-mail*

Configurar um servidor de *e-mail* é dos passos essenciais para garantir o acesso a todas as funcionalidades que o sistema oferece. Por um lado, é crucial para a criação de vários de utilizadores, pois inicialmente existe um único, e só enviando um convite de colaboração por *e-mail* é que se consegue adicionar outros. Por outro, permite ainda a recuperação de *passwords*.

A ferramenta escolhida consistiu no Mailgun, uma API de *e-mails* transacionais que permite enviá-los, recebê-los e segui-los. O primeiro passo consiste em criar uma conta no Mailgun, na qual é configurado automaticamente um domínio que o utilizador pode usufruir. Com o utilizador SMTP e *password* defeito criados, simplesmente temos que adicionar esta informação ao ficheiro *config.production.json*, como mostra o exemplo abaixo, e passamos a ter um serviço de *e-mail* funcional.

Extrato 5.5 Exemplo de configuração do servidor de *e-mail*

```
"mail": {
  "transport": "SMTP",
  "options": {
    "service": "Mailgun",
    "auth": {
      "user": "postmaster@example.mailgun.org",
      "pass": "password"
    }
  }
}
```

5.4 Reverse Proxy

O *reverse proxy* foi implementado usando a ferramenta Nginx. Para além das funcionalidades básicas de reencaminhamento dos pedidos recebidos para a

porta 2368, oferece outras características que permitem otimizar o desempenho do sistema, sendo estas a compressão e *caching* de ficheiros.

Facilmente se compreende o porquê da compressão ser benéfica, pois o tempo de carregamento de um *website* depende diretamente do tamanho dos ficheiros que contém, pelo que ao utilizar ficheiros comprimidos e consequentemente mais pequenos, beneficiamos de um melhoramento na rapidez de resposta. Para conseguir tal, podemos recorrer à ferramenta *gzip*, cujo o objetivo é comprimir os ficheiros que são necessários. É possível implementá-lo editando o ficheiro de configuração localizado em */etc/nginx/nginx.conf* e descomentando a configuração sobre este já previamente definida.

Por outro lado, o desempenho pode ser melhorado usando outra funcionalidade que o Nginx oferece, o *caching*, pois este permite servir ficheiros estáticos. Se tal não acontecer, todos os pedidos efetuados, quer sejam imagens ou ficheiros JS e CSS, irão ser reencaminhados para o servidor aplicacional, o que se traduz numa grande sobrecarga para o sistema.

Para tal é preciso alterar o mesmo ficheiro que no ponto anterior adicionando as seguintes linhas.

```
upstream ghost {
  server 127.0.0.1:2368;
  keepalive 64;
}
```

De seguida é preciso alterar o ficheiro de configuração do Nginx para o *blog* Ghost e adicionar as seguintes linhas:

```
location ^~ /assets/ {
    root /var/www/ghost/content/themes/theme_name;
}
```

Uma vez que, na diretoria */assets* estão contidos os ficheiros JS, CSS e fontes, ao adicionar estas linhas, informamos o Nginx que qualquer pedido por estes ficheiros pode ser respondido no *path* descrito e não tem que questionar o CORE. O mesmo pode ser feito para as imagens.

```
location ^~ /content/images/ {
    root /var/www/ghost;
}
```

O passo seguinte é ativar o *caching* e definir preferências tais como o tamanho máximo de disco que os ficheiros em *cache* podem ocupar, o tempo válido de *cache*, entre outros, e passamos a ter um sistema de *caching* funcional.

5.5 Balanceamento de Carga

Com o objetivo de implementar uma distribuição equilibrada dos pedidos que os clientes realizam ao sistema, foi escolhida a ferramenta HAProxy para implementar balanceamento de carga. Para termos acesso às suas funcionalidades é apenas necessário instalá-la na máquina pretendida e alterar o ficheiro de configuração em */etc/haproxy/haproxy.cfg*, de modo a este ir ao encontro dos requisitos necessários.

Existem então 2 tipos de nodos que precisam de ser definidos: o *frontend* e o *backend*. O primeiro é configurado de modo a que o HAProxy receba os pedidos da porta 80 e estes sejam reencaminhados para o *backend* definido. Neste é especificada a estratégia de balanceamento de carga, sendo neste caso *round robin*, assim como os servidores para onde irão ser reencaminhados os pedidos, identificados pelo seu endereço IP e pela porta onde os estão a receber. É de notar que a porta de onde os pedidos do HAProxy estão a ser enviados e a que os recebe no servidor aplicacional tem que ser a mesma, sendo normalmente escolhida a 8080.

É possível ainda definir o modo *httpchk*, que testa se um servidor *web* é válido, não sendo então encaminhados pedidos para este, caso falhe o teste.

Podemos adicionalmente configurar um sistema de estatísticas que nos fornece informação sobre o estado os servidores, os pedidos realizados e outros dados pertinentes.

Capítulo 6

Instalação e Configuração Automática

Uma vez discutido o padrão de distribuição a usar de modo a maximizar o desempenho da aplicação (Capítulo 4), passamos a discutir como fazer uma instalação e configuração automática de todos os seus componentes.

O Ansible é uma ferramenta *open-source* que permite a instalação e atualização de pacotes de uma forma completamente automática. É esta a ferramenta que escolhemos utilizar para a instalação e configuração automática do nosso sistema.

A instalação dos componentes será feita recorrendo a máquinas da Google Cloud Platform, sendo que o nosso *playbook* é constituído por vários *roles* a serem corridas em *hosts* específicos dependendo das suas necessidades. A arquitetura escolhida e estes *roles* serão discutidos brevemente de seguida.

6.1 Arquitetura da instalação

Revisitando a arquitetura escolhida, pretendemos instalar um sistema que seja no total constituído por 5 instâncias, sendo estas:

- 1 instância - balanceamento de carga (HAProxy);
- 2 instâncias - *reverse proxy* (Nginx) e servidor aplicacional (Ghost) num *container*;
- 2 instâncias - servidores de base de dados (MySQL);

Recordando o ponto 4.2, decidiu-se replicar os servidores de base de dados de forma a garantir que caso haja uma falha num servidor, o serviço não se torne indisponível. Quanto aos servidores *web* e aplicacionais, escolheu-se replicá-los pelas razões descritas no ponto 4.1, ou seja, para assegurar tolerâncias a faltas e melhorar o desempenho. Também se decidiu fazer a instalação do servidor

aplicacional dentro de um *container* Docker, para que posteriormente, possamos ter vários servidores aplicacionais a correr na mesma instância.

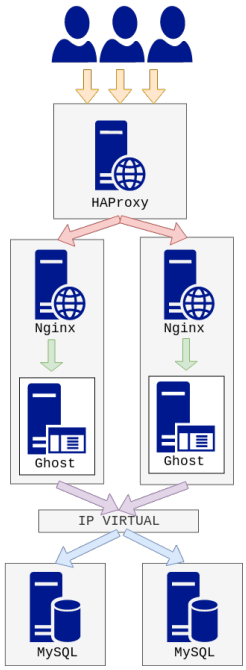


Figura 6.1: Arquitetura escolhida

6.2 *Roles*

6.2.1 `gcp`

Este é responsável por criar os discos, a rede e as regras de *firewall* internas e externas necessárias para a criação das instâncias na Google Cloud Platform (as quais é responsável por criar também, posteriormente). Faz uso dos módulos disponibilizados pelo Ansible que cobrem a criação e gestão de recursos da Google Cloud Platform.

Para cada uma das instâncias criadas associa uma *tag* de acordo o papel que desempenha e conforme esse executa ou não um determinado *role*. A tabela seguinte mostra quais os *roles* aplicados a cada instância.

Role \ Tag	database	app/nginx	haproxy
users	x	x	x
docker		x	
database	x		
db-replication	x		
keepalived	x		
app-ghost		x	
web-nginx		x	
haproxy			x

Tabela 6.1: Roles aplicados a cada instância

6.2.2 `users`

Cria um utilizador com permissões de *root* e que permite autenticação por chave pública SSH.

6.2.3 `docker`

Dado que são várias as situações a que se recorre a imagens e *containers* Docker é crucial preparar as máquinas, isto é, instalar a ferramenta Docker e as dependências necessárias.

6.2.4 `database`

O role `database` é responsável por instalar nas instância que irão suportar a base de dados um servidor MySQL, assim como todas as configurações necessárias para que este seja funcional, tais como definir a password do utilizador *root*, criar a base de dados da aplicação, bem como um utilizador com as devidas permissões para que os servidores aplicacionais se possam conectar e utilizá-la.

6.2.5 `db-replication`

O role `db-replication` trata de configurar a replicação da base de dados. Dado que é importante que as base de dados sejam dotadas de *failover*, isto é, caso o servidor *master* falhe, seja possível realizar interrogações ao *backup*, é necessário que a replicação seja do tipo *master-master*. Para tal, foi necessário alterar o ficheiro de configuração da base de dados, bem como, através da API disponibilizada pelo Ansible para replicação de servidores MySQL, configurá-los de modo a que cada um seja *slave* e *master* do outro.

6.2.6 `keepalived`

Como referido anteriormente, *failover* é uma propriedade importante para podermos garantir tolerância a falhas da aplicação. Para esse efeito foi criado

o role `keepalived`, que trata de preparar as instâncias para situações em que o servidor *master* pare de funcionar, e faça com que o servidor *backup* se torne em *master* e aceite os pedidos feitos à camada de dados.

Para conseguirmos tal, seguimos os passos descritos no ponto 5.2.

6.2.7 app-ghost

O `app-ghost` trata de instalar o servidor aplicacional recorrendo à ferramenta Docker que permite a geração de imagens Docker de uma forma simples e automática, assim como correr *containers* a partir desta.

No entanto, antes de criar uma imagem e colocar um *container* a correr é necessário preparar a instância da Google Cloud Platform a fim de ter todos os recursos necessários para que seja possível utilizar o Docker. Tal é feito previamente pelo *role* descrito na secção 6.2.3. De seguida, cria-se uma diretoria onde se colocará toda a informação para que seja possível criar uma imagem Docker. A esta é adicionado o *Dockerfile* a partir do qual se vai criar a imagem com o Ghost, assim como o ficheiro de configuração de produção que especifica os recursos necessários, tais como explicados na secção 5. Posteriormente, é gerada uma imagem usando o *Dockerfile* adicionado, à qual se deu o nome de `my/ghost`.

Extrato 6.1 Dockerfile utilizado para a criação da imagem my/ghost

```
FROM mm97/ghost_install:v1
ENV NODE_ENV production
WORKDIR /ghost
ADD config.production.json config.production.json
RUN npx knex-migrator init
CMD ["node", "index.js"]
```

A fim de facilitar o processo de criação da imagem e torná-lo mais rápido, foi criada e colocada no *Docker Hub* uma imagem auxiliar que trata de instalar todas as dependências necessárias, pelo que no *Dockerfile* apenas é necessário fazer referência a esta por `FROM mm97/ghost_install:v1`, definir que se trata de um cenário de produção, mudar de diretoria, importar o ficheiro de configuração, ligar à base de dados e definir o comando que inicializa o servidor.

Posteriormente, usando a imagem criada, é colocado a correr um *container* na porta 2368, ao qual se deu o nome de `ghost_blog`, e temos um servidor aplicacional completamente funcional, que pode ser facilmente replicado.

6.2.8 web-nginx

Instala na instância a ferramenta Nginx e copia ficheiros de configuração estruturados de modo a que seja possível a compressão de ficheiros e o *caching*.

6.2.9 `haproxy`

Este *role* trata de instalar e configurar a ferramenta HAProxy que permite obter uma distribuição equilibrada dos pedidos que os clientes realizam ao sistema. Deste modo é feito o *download* e instalação da ferramenta e adicionado um ficheiro de configuração para que faça a distribuição da carga para cada um dos *reverse proxies*.

6.3 Cenário de produção *vs* Cenário de desenvolvimento

Dependendo do objetivo final, existem 2 cenários diferentes de onde podemos escolher ao fazer a instalação da aplicação, sendo estes de produção ou de desenvolvimento.

Para o primeiro, devemos fazer uma instalação de todos os componentes, utilizando o MySQL como servidor de base de dados e o `config.production.json` como ficheiro de configuração. Uma vez que é do nosso interesse observar a atividade da infraestrutura, é importante também implementar um sistema de monitorização.

Quanto ao cenários de desenvolvimento, existem algumas diferenças deste primeiro. Por um lado, devemos utilizar como servidor de base de dados o SQLite3 e de ficheiro de configuração o `config.development.json`. Por outro, para além das ferramentas de monitorização, é também importante prover o sistema de ferramentas de avaliação, para que possamos efetuar testes de uso do sistema.

Para conseguirmos uma instalação automática de cada um destes cenários, é importante configurar o projeto Ansible, seguindo a seguinte estrutura:

```
.
|— ansible.cfg
|— environments/
|   |
|   |— dev/  #cenário de testes
|   |   |— ...
|   |
|   |— prod/ #cenário de produção
|       |— ...
|
|— playbook.yml
|— ...
```

Assim, apenas precisamos de definir ao correr o *playbook* qual é o cenário que pretendemos fazer *deployment*, e conseguimos uma instalação automática de vários cenários de teste.

Capítulo 7

Monitorização

Com o objetivo de observar a atividade do sistema, o próximo passo foi dotá-lo de uma ferramenta de monitorização.

Para esta, utilizamos o **Elasticsearch** para indexação e armazenamento, alguns dos Beats oferecidos pela Elastic para recolha de dados (nomeadamente **Metricbeat**, **Packetbeat**, **Filebeat** e **Heartbeat**) e o **Kibana** para visualização destes (Figura 7.1).

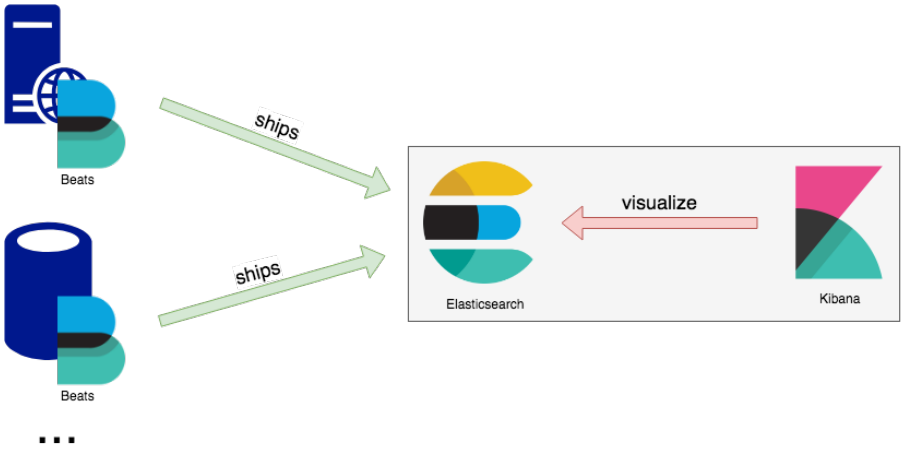


Figura 7.1: Monitorização com Beats, Elasticsearch e Kibana

7.1 Instalação e Configuração

Para a sua instalação e configuração automática, foi necessária a criação de uma nova máquina na Google Cloud Platform (destinada a ser usada tanto pelo Elasticsearch como pelo Kibana) e de dois novos *roles*, que passamos a explicar brevemente de seguida.

- `elasticsearch`

  Este *role*, destinado a correr na nova máquina, propõe-se a instalar as imagens do Docker para o **Elasticsearch** e **Kibana**. Uma vez usando o Docker, a configuração necessária é mínima, pelo que apenas foi necessário configurar as portas nas quais o Elasticsearch e o Kibana iam correr, para além da alteração de duas variáveis do `sysctl`.

  Ainda foi preciso instalar nesta mesma máquina o Heartbeat, pois ao contrário dos outros *beats* este deve correr na máquina do Elasticsearch/-Kibana de forma a interrogar as máquinas *edge* acerca do seu funcionamento.

- `beats`

  Este *role*, destinado a correr em todas as máquinas que se pretende monitorizar, servirá para instalar, configurar e por a correr os transportadores responsáveis por enviar os dados das máquina *edge* para o Elasticsearch.

7.2 Métricas de Uso de Recursos

7.2.1 CPU, Memória, Disco, Tráfego Entrada/Saída

Para diagnóstico de problemas, estas métricas são muito importantes, especialmente quando analisadas em conjunto. Para o Ghost, estamos a monitorizar o uso do CPU, da memória e de disco e ainda os tráfegos de entrada e saída (Figura 7.2).

Se o uso do CPU, Memória ou Disco se encontrar alto, é uma garantia que o desempenho da aplicação está comprometido, pois estas dão-nos indicação da habilidade do sistema ficar estável após um aumento de carga. A monitorização usada permite-nos observar estas estatísticas para cada uma das instâncias, mas também para todas as instâncias agregadas.

Estas métricas são especialmente importantes em ambiente de produção, pois indicam o uso de recursos num contexto com utilizadores reais, isto é, onde o uso destes reflete um uso real.

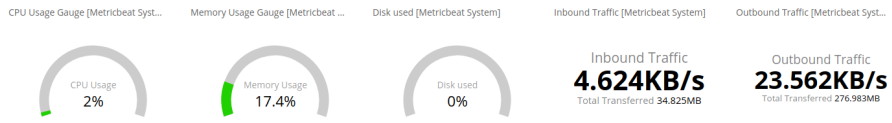


Figura 7.2: Métricas relacionadas com o uso de recursos do sistema

7.3 Métricas de Desempenho

7.3.1 Tempo de Resposta

Para perceber o desempenho do ponto de vista do utilizador do *blog*, quer seja autor quer seja leitor, uma métrica muito importante é a análise dos tempos de resposta. Isto deve-se ao facto de o tempo de resposta indica se tratar de quanto tempo a aplicação demora a responder a um pedido do utilizador.

Esta métrica é muito importante em ambiente de desenvolvimento, pois dependendo do número de utilizadores, o tempo de resposta pode diminuir drasticamente. É assim crucial, em teste, perceber como os tempos de resposta variam consoante o número de utilizadores concorrentes e otimizar a aplicação de forma a que o desempenho da aplicação não fique comprometido com um número mais alto de utilizadores concorrentes.

E assim, ela é também importante em ambiente de produção, pois permite-nos perceber mais acerca da experiência de utilizador num contexto real.

A Figura 7.3 mostra os tempos de resposta ao longo de um determinado período de tempo, sendo este um dos gráficos que podemos observar no Kibana relativos ao desempenho da aplicação Ghost. A predominância verde no gráfico é um bom indicativo, já que indica que a maioria dos pedidos são respondidos em tempo perto do nulo (0 ms). No entanto, este gráfico não representa tempos de resposta num cenário com muitos utilizadores concorrentes, servindo apenas para ilustrar a importância do tempo de resposta como métrica para o desempenho. Uma análise dos tempos de resposta neste tipo de cenários será feito no Capítulo 9.

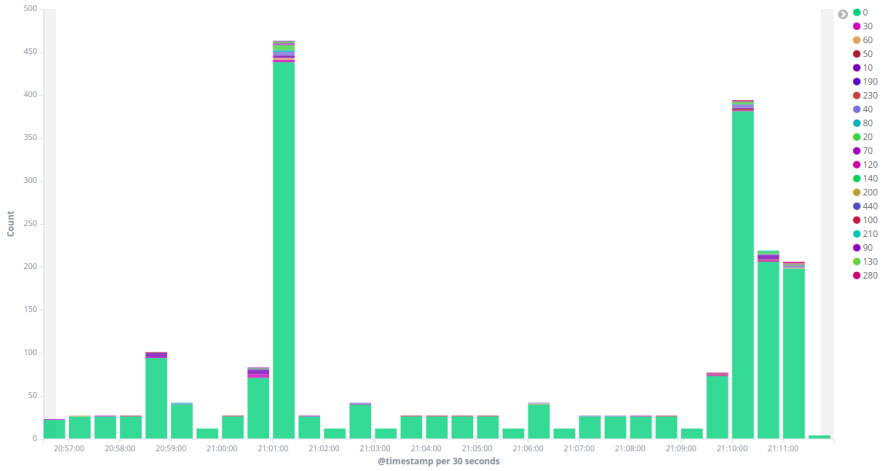


Figura 7.3: Gráfico de tempos de resposta

7.3.2 Latência

A latência é outra métrica de desempenho importante na análise dos tempos de resposta. O gráfico da Figura 7.4, permite-nos saber o número de pedidos que teve resposta dentro de um determinado tempo de resposta. Por exemplo, consegue-se ver que o maior número de pedidos tem resposta em menos de 10 ms.

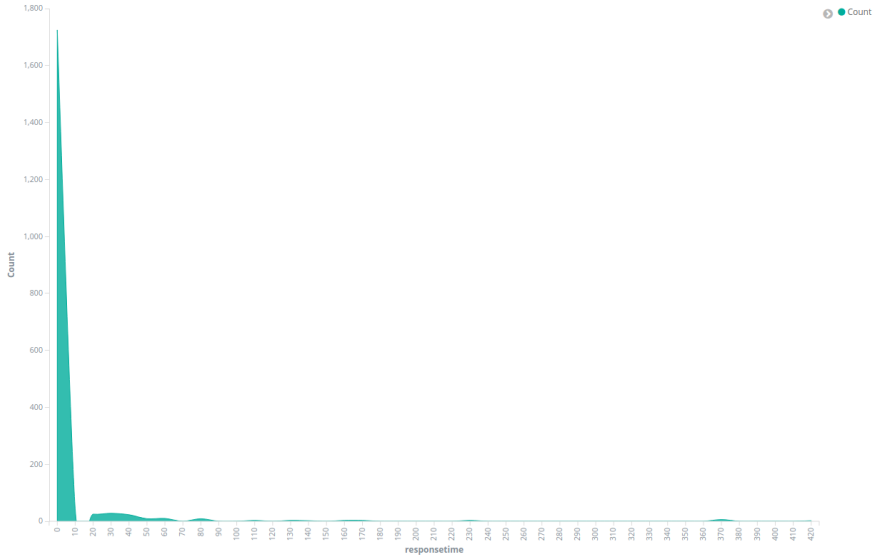


Figura 7.4: Gráfico da latência

7.3.3 Percentagem de Erros

Quando usamos aplicações como utilizadores, neste caso se estivermos a ver publicações de um *blog* ou a escrever para um, não desejamos ver erros.

Outra métrica importante para a análise de desempenho da nossa aplicação é a observação da percentagem de erros que apareceram aos utilizadores. Esta métrica é muito importante em produção, pois permite saber que tipo de erros os utilizadores obtém num contexto real e potencialmente descobrir a fonte destes erros.

A nossa monitorização permite perceber, a cada período de tempo de 30 segundos, quantas transações tiveram sucesso e quantas acabam em erro (Figura 7.5).

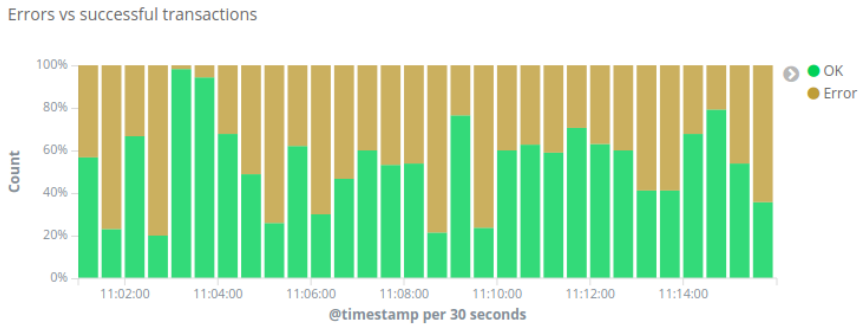


Figura 7.5: Gráfico de percentagem de transações com sucesso/erro

É ainda possível saber de que forma são distribuídos estes erros por tipos (Figura 7.6), podendo-se assim mais facilmente perceber as maiores fontes de erros da aplicação. Para este caso, observa-se que a maioria dos erros são erros HTTP.

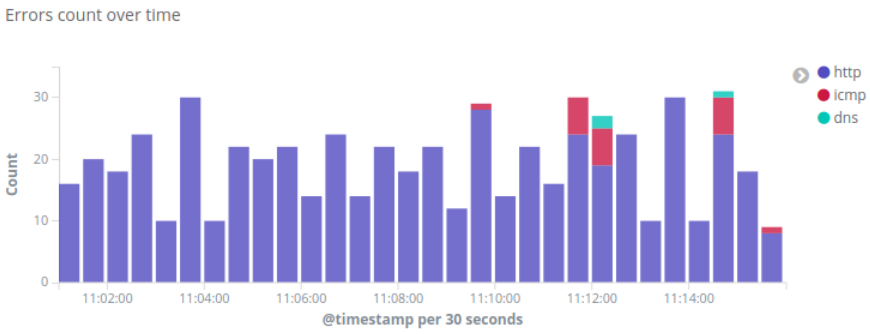


Figura 7.6: Gráfico de contagem do tipo de erros ocorridos

7.3.4 *Uptime*

Mediante o uso do Heartbeat, que ao contrário dos outros *beats* não é instalado nas máquinas *edge* mas sim na mesma máquina que o Elasticsearch/Kibana, conseguimos monitorizar o *uptime* (percentagem de tempo que um servidor está disponível para uso) das outras máquinas.

Esta métrica é crítica em ambiente de produção, uma vez que queremos que o nosso serviço esteja sempre disponível para os utilizadores. Idealmente, queremos que este valor se mantenha sempre perto dos 100%. Com isto, caso haja alguma falha, podemos saber logo qual exatamente foi a máquina que deixou de estar disponível.

A Figura 7.7 mostra a página do Kibana com os resultados da monitorização do *uptime* dos servidores *web* e do balanceador de carga. No gráfico debaixo

é possível observar a quantidade das máquinas disponíveis em determinados instantes de tempo.

Figura 7.7: *Dashboard* para o Heartbeat

7.4 Análise de *Logs*

Uma vez que reconhecemos a importância dos *logs* num ambiente de desenvolvimento, decidimos usar o Filebeat nas máquinas com o servidor *web* e base de dados.

O facto de ser possível ler os *logs* no Kibana, permite, num ambiente de desenvolvimento, ser mais fácil encontrar problemas na aplicação, já que todos eles ficam acessíveis num só sítio, não sendo necessário aceder a cada máquina individualmente.

O Filebeat permite especificar os *logs* desejados, pelo que para o servidor *web* estamos a ver os *logs* de acesso e erro e para a base de dados o *log* geral.

Na Figura 7.8 conseguimos ver o *log* de acesso do servidor *web* a ser mostrado no Kibana. Este dá-nos informações sobre os pedidos que passaram pelo servidor *web*, assim como as respostas HTTP que originaram.

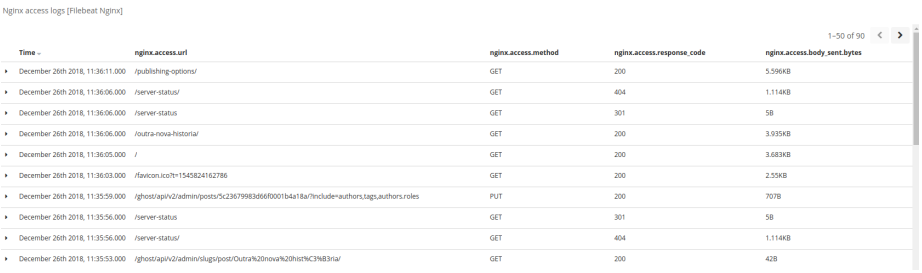


Figura 7.8: *Log* de acesso do Nginx

Capítulo 8

Avaliação

Avaliar o desempenho de uma aplicação é um tópico de extrema importância no que diz respeito a assegurar a qualidade dos serviços que oferecemos. Esta avaliação é realizada com o intuito de garantirmos que a aplicação aguenta com determinada quantidade de utilizadores em simultâneo, isto é, que o tempo de resposta não se degrade de tal forma a proporcionar más experiências aos utilizadores.

Os testes que serão realizados são, portanto, testes de carga, onde vão ser simulados picos de utilizadores a aceder ao sistema, com o intuito de o saturar e investigar até onde ele suporta. Os testes vão ser iniciados com uma carga baixa que vai aumentando gradualmente.

8.1 Procedimento e Ferramentas Adotadas

Para a avaliação da aplicação, optamos por utilizar o **Apache JMeter**. Esta trata-se de uma aplicação de testes de *performance* desenvolvida para aplicações *web*, nas quais se enquadra perfeitamente o Ghost. Esta ferramenta foi criada pela Apache Software e é a mais utilizada para este seguimento.

Para que a simulação possa ser realizada com testes mais complexos, em vez de descrevermos manualmente cada pedido na interface gráfica do JMeter, utilizámos o BlazeMeter, uma extensão do Chrome, que permite gravar os pedidos que são interativamente realizados no *browser*.

O primeiro passo do procedimento de realização dos testes foi, portanto, a criação dos próprios testes. Para tal como mencionado, colocamos o BlazeMeter a gravar e procedemos à realização de uma simulação para cada tipo de teste delineado. Os testes são guardados no formato **jmx**, específico do JMeter.

Para que os testes fossem realizados de forma controlada, evitando o *overhead* de os executar a partir de uma interface gráfica, bem como o *overhead* relacionado com atrasos de comunicação, os ficheiros de teste foram enviados para uma máquina virtual dedicada à execução do JMeter. Nesta máquina os testes são realizados via terminal e os resultados enviados para a nossa máquina local

para análise.

O último passo constitui, efetivamente, a análise dos resultados que é realizada, agora sim, na interface gráfica do JMeter.

8.2 Criação dos *scripts* de teste

Assim como em qualquer outro tipo de teste, é importante decidir o que deve ser testado e como proceder a essas avaliações. Não nos interessando avaliar todas as funcionalidades do sistema, uma vez que o esforço seria alto e desnecessário, foi fundamental avaliar quais funcionalidades priorizar. Posto isto, sendo a aplicação em estudo um *blog*, consideramos de especial relevância dois tipos de teste: acesso ao site e leitura de publicações; autenticação no *blog* e escrita de uma publicação.

Estas duas funcionalidades constituem papéis desenrolados por dois diferentes tipos de utilizadores, leitores e autores. Estes utilizadores são os que possuem a maior representatividade no sistema, contudo, certamente a percentagem de leitores será bastante maior que a de autores do *blog*. Tal como mencionado anteriormente, os testes foram desenvolvidos utilizando a funcionalidade de gravação disponibilizada pelo BlazeMeter, onde simulamos o tipo de utilizador em questão.

8.2.1 Acesso ao site e leitura de publicações

Adquirindo o papel de leitor, a gravação deste tipo de teste foi realizada acedendo à página inicial do *Ghost* e seguidamente à publicação "welcome", contemplando, portanto, dois únicos pedidos http, tal como demonstra a figura 8.1.

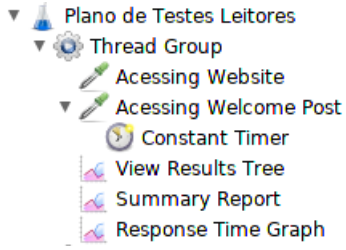


Figura 8.1: *Script* de testes - cenário leitores

Tal como é possível verificar pela figura, faz-se uso de um temporizador no acesso à publicação pretendida. Isto acontece de forma a simular o tempo que o utilizador disponibiliza na escolha da publicação que pretende ler, neste caso cerca de 9 segundos.

Com estes testes pretendíamos, essencialmente, visualizar o impacto no débito e no tempo de resposta dos pedidos de aumentar o número de utilizadores. Por esta razão delineamos um período de *ramp-up* de 300 segundos,

isto para as *threads* serem sucessivamente acrescentadas. Definimos o *loop count* para "sempre", isto é, os utilizadores desde que são criados encontram-se continuamente, até ao fim do teste, a fazer os mesmos pedidos de forma a que não se perdesse carga (utilizadores) à medida que novos utilizadores eram criados. Definimos ainda para a duração do teste 360 segundos, pelo que, uma vez todos os utilizadores inseridos, o sistema fica com a carga total de todos os utilizadores durante 1 minuto.

Com o intuito de levar o sistema a aninhar e verificar as limitações do mesmo, este teste foi executado para 1000, 3000 e 5000 utilizadores, cargas estas que como veremos na capítulo 9 serão suficientes para atingir os objetivos pretendidos.

Autenticação no *blog* e escrita de uma publicação.

No cenário em que o utilizador se trata de um escritor, a gravação foi realizada acedendo de início diretamente à página de autenticação. Depois, inserimos as credenciais de um utilizador previamente criado, escrevemos uma publicação com um tamanho razoável (5 parágrafos) e procedemos, por fim, à sua publicação.

Uma vez no JMeter constatamos a necessidade de proceder a várias alterações para que fosse possível a execução dos testes para vários autores do *blog*. Tal deve-se ao facto da aplicação em estudo não permitir utilizar a conta de um mesmo utilizador e criar várias publicações em simultâneo.

De entre as alterações que foram necessárias realizar destacam-se as seguintes:

- Criação de um *script* de povoamento base de dados de forma a que cerca de uma centena de autores pudesse ser inserida com as corretas permissões de publicação no blog;
- Criação de um *script* para popular um ficheiro .csv com algumas informações dos autores criados. Este ficheiro .csv é utilizado para fornecer alguns valores de variáveis a utilizar nos pedidos *http* do teste. Para tal utilizou-se o componente "CSV Data Set Config" do JMeter;
- Utilização do componente "Regular Expression Extrator" do JMeter para construir algumas variáveis a partir de respostas a pedidos, variáveis essas a serem utilizadas em pedidos subsequentes.

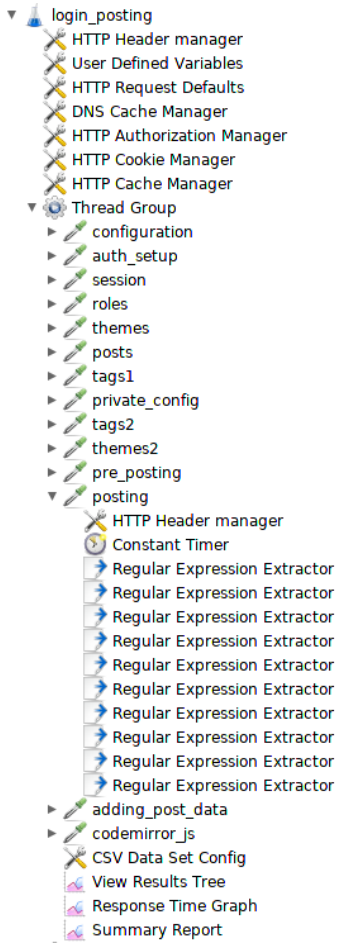


Figura 8.2: *Script* de testes - cenário Escritores

Encontra-se descrito na figura 8.2 a árvore representativa do teste em questão, onde é visível a presença de vários pedidos *http*. Entre outros, estes incluem pedidos de ficheiros JavaScript, pedidos necessários à criação da página principal do utilizador autenticado (publicações que escreveu, *tags*, etc) e ainda pedidos relacionados com a criação da página de autenticação e da própria publicação realizada.

Este teste revelou, contudo, uma grande limitação que se encontra intimamente relacionado com o facto do Ghost apenas permitir 100 pedidos de autenticação a partir de um mesmo endereço IP, por hora. Dado este facto, os testes realizados apenas puderam incidir sobre valores inferiores ou igual ao mencionado. Decidimos, portanto, realizar o teste para exatamente 100 escritores em simultâneo e diminuir caso o sistema não aguentasse, o que não foi o caso (Capítulo 9).

Diferentemente do que se havia definido para os testes dos leitores, este tipo de teste não pode executar constantemente devido à restrição no número de autenticações, pelo que se definiu um *loop count* de 1. Consequentemente, como queríamos os utilizadores a postar todos ao mesmo tempo, foi definido um *ramp-up* de 0.

Capítulo 9

Resultados Finais

Com o intuito de perceber o comportamento da aplicação com diferentes números de clientes concorrentes e diferentes configurações de componentes, corremos vários testes que iremos expor neste capítulo, tendo observado tanto alguns dos dados recolhidos pela monitorização como os resultados dados pela ferramenta de avaliação no que toca a tempos de resposta obtidos.

9.1 Procedimento Adotado

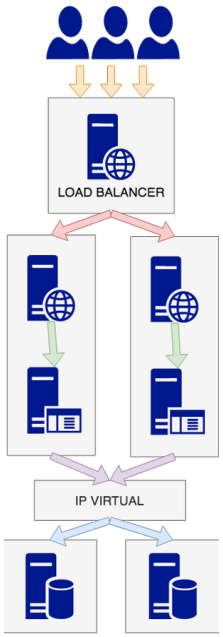


Figura 9.1: Arquitetura alternativa com dois servidores *web*

Para a realização dos testes que se encontram nas secções seguintes, decidiu-se utilizar a arquitetura utilizada até então, que se encontra descrita na Figura 9.1.

Podemos observar que estão a ser utilizados dois servidores *web* (e, consequentemente, dois servidores aplicacionais, já que estes se encontram na mesma máquina) para receber os pedidos vindos dos utilizadores. Está ainda a ser utilizada apenas uma base de dados que, caso falhe, será substituída por outra igual a ela (replicada).

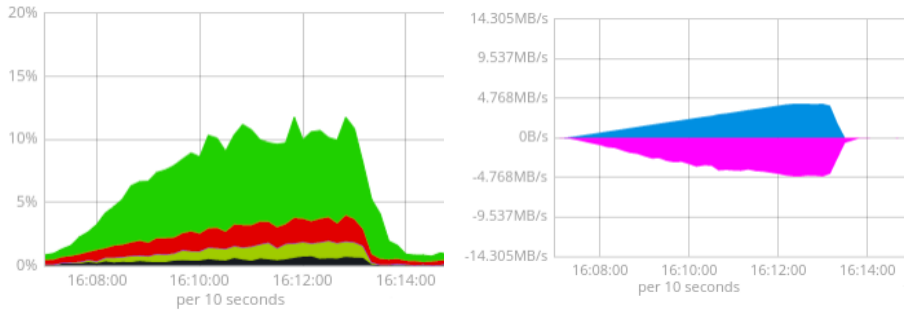
Pretende-se usar os dois *workloads* expostos no Capítulo 8, sendo um deles relativo à leitura de publicações no *blog* e o outro relativo ao *login* e escrita de publicações no mesmo. Com o intuito de ver o efeito de cargas diferentes de utilizadores concorrentes para estes dois *workloads*, os testes a efetuar serão feitos para valores diferentes de leitores/autores do

blog. Para leitura, e uma vez que se espera que mais utilizadores façam esta operação, os testes serão efetuados para 1000, 3000 e 5000 clientes concorrentes. Por sua vez, já que nunca existirão tantos autores quanto existem leitores, o teste efetuado para a escrita será para 100 clientes concorrentes.

9.2 Leitura de publicações

9.2.1 1000 clientes concorrentes

Em termos de uso de CPU e tráfego de entrada e saída (Figura 9.2), dados obtidos com a monitorização do sistema, não é detetada muita instabilidade e o tráfego não está excessivamente alto. A Figura 9.2 reflete os resultados globais (ou seja, uma média de todas as máquinas), no entanto, observamos cada máquina individualmente e concluímos que o uso de CPU é muito elevado nas máquinas onde estão instalados os servidores *web* e aplicacionais, sendo que nas das base de dados mantém-se estável e baixo (cerca de 1%). Este comportamento era expectável uma vez que se está a utilizar *caching*, pelo que as respostas ao pedidos realizados já estão muito provavelmente em memória.



(a) Uso de CPU

(b) Tráfego de entrada e saída

Figura 9.2: Dados obtidos com monitorização para 1000 clientes concorrentes

Relativamente ao tempo de resposta, é possível verificar, no gráfico da figura 9.3, que esta métrica se mantém constante à medida que o número de utilizadores aumenta. Tal acontece porque a carga que 1000 leitores induzem no sistema é muito reduzida. Todos os pedidos são, desta forma, respondidos em poucos milissegundos como aliás se pode comprovar na tabela 9.4 onde o tempo médio de resposta é de 3 ms para o acesso ao *blog* e de 41 ms para o acesso à publicação desejada.

Ainda na tabela é visível que a percentagem de erros é 0% o que significa que todos os pedidos foram respondidos com sucesso e, uma vez mais, permite concluir que o sistema se encontra ainda longe dos limites de saturação. O

débito total é 130.1 pedidos por segundo, débito este que se prevê aumentar passando para um maior número de clientes concorrentes.

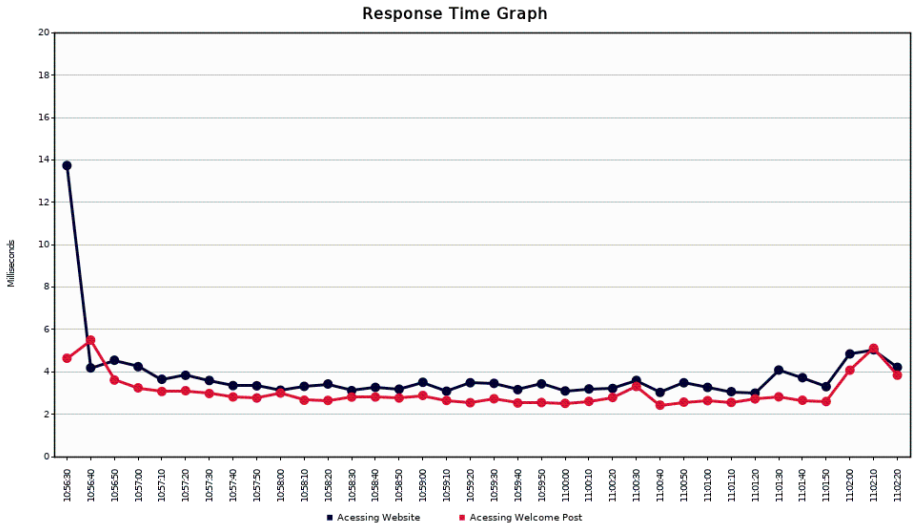


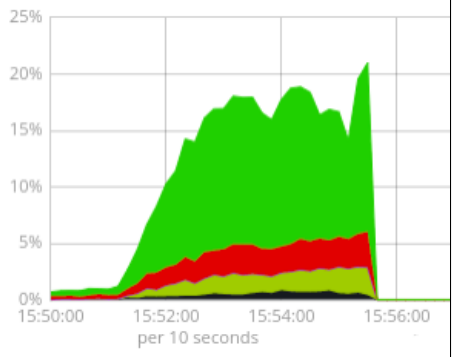
Figura 9.3: Gráfico Tempo de resposta para 1000 clientes concorrentes

Label	# Samples	Average	Min	Max	Std. D...	Error...	Throug...	Receiv...	Sent K...	Avg. B...
Acessing ...	23494	3	2	189	5.34	0.00%	65.3/sec	2758.76	7.40	43230.0
Acessing ...	23494	41	1	1510	197.94	0.00%	66.8/sec	1375.06	8.08	21094.0
TOTAL	46988	22	1	1510	141.28	0.00%	130.1/...	4086.35	15.25	32162.0

Figura 9.4: Tabela sumário para 1000 clientes concorrentes

9.2.2 3000 clientes concorrentes

Com o aumento de clientes para 3000, os resultados na monitorização mostram também um aumento no uso de CPU (de cerca de 10%, o uso do CPU global passou para sensivelmente 18%), o que era de esperar. Mais uma vez, mediante a observação de cada máquina individual, observa-se que o uso do CPU é muito mais elevado nas máquinas onde estão instalados os servidores *web* e aplicacionais.



(a) Uso de CPU

pics/ler-dois-ws/ler3000-networktraffic.png

(b) Tráfego de entrada e saída

Figura 9.5: Dados obtidos com monitorização para 3000 clientes concorrentes

Também a nível do tempo de resposta se presenciaram diferenças significativas, o gráfico abaixo é representativo, evidenciando um ligeiro aumento da instabilidade e dos tempos de resposta à medida que mais utilizadores são acrescentados. Se apenas tivermos em consideração o período em que o número de utilizadores é máximo (3000 leitores), isto é, a parte mais à direita do gráfico, podemos estimar um valor médio de 1.2 segundos para o tempo de resposta, que constitui um valor aceitável no que respeita ao acesso a uma página *web*.

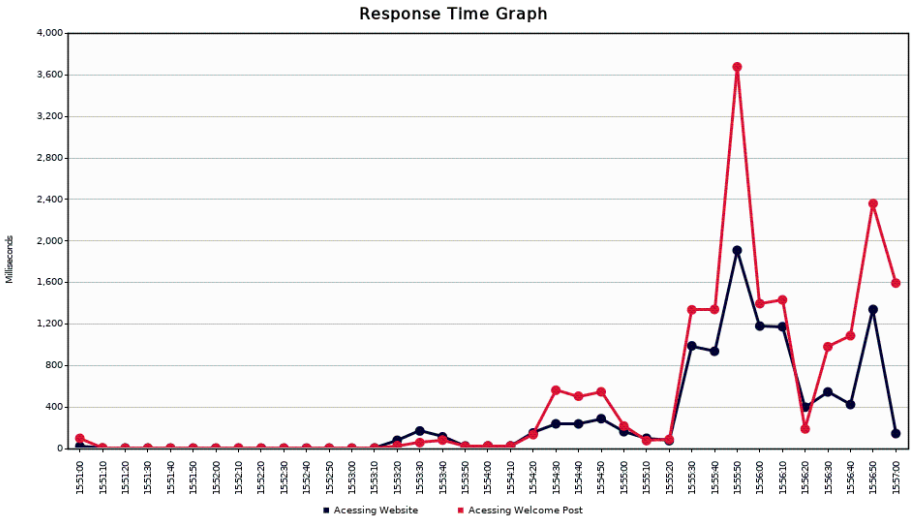


Figura 9.6: Gráfico Tempo de resposta para 3000 clientes concorrentes

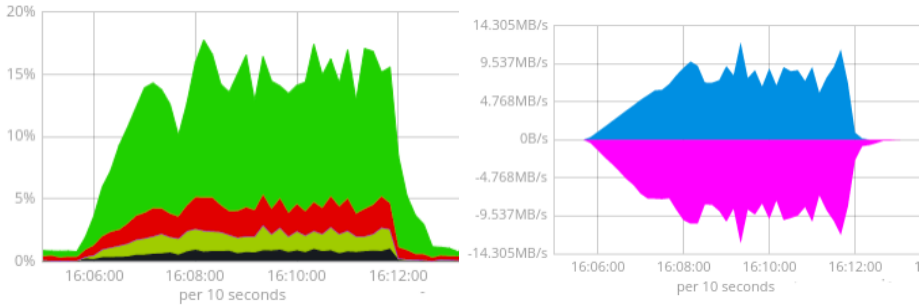
Olhando para a tabela 9.7 é possível verificar que a percentagem de erros continua nula, pelo que o sistema ainda não se encontra saturado. O débito atingiu um valor de 323 pedidos por segundo, sendo já um valor considerável, contudo, espera-se continuar a subir até atingir o tal nível de saturação.

Label	# Sam...	Aver...	Min	Max	Std. D...	Error...	Throug...	Receive...	Sent ...	Avg. ...
Acessing ...	58957	394	1	8876	841.68	0.00%	164.3/...	6938.19	18.61	4325...
Acessing ...	58957	712	1	10577	1295.46	0.00%	165.7/...	3413.03	20.06	2109...
TOTAL	117914	553	1	10577	1103.85	0.00%	323.0/...	10149....	37.85	3217...

Figura 9.7: Tabela sumário para 3000 clientes concorrentes

9.2.3 5000 clientes concorrentes

Com o aumento para 5000 clientes concorrentes, os resultados indicam-nos um valor de uso do CPU muito semelhante ao obtido para 3000 clientes concorrentes (Figura 9.8). Isto pode indicar um cenário de saturação do sistema.



(a) Uso de CPU

(b) Tráfego de entrada e saída

Figura 9.8: Dados obtidos com monitorização para 5000 clientes concorrentes

O gráfico da Figura 9.9 vem comprovar a saturação do sistema na medida em que o tempo de resposta aumenta para mais do dobro do que se obteve com 3000 leitores, chegando até mesmo a aproximar-se dos 10 segundos para obter a publicação pretendida. Quando o número de utilizadores é máximo, a média do tempo de resposta ronda os 4.5 segundos para a obtenção da página principal do *blog* e 2.7 segundos para a publicação, consideramos pois que não se trata de um valor aceitável para os utilizadores da aplicação. O débito também diminuiu relativamente à configuração de 3000 utilizadores e, podemos agora presenciar a ocorrência de alguns erros, vindo uma vez mais enaltecer o grau de saturação.

Concluindo, por observação dos diversos débitos e tempos de resposta obtido para os diferentes números de utilizadores, constata-se que a eficiência máxima do sistema é obtida para um número de utilizadores entre 3000 e 5000.

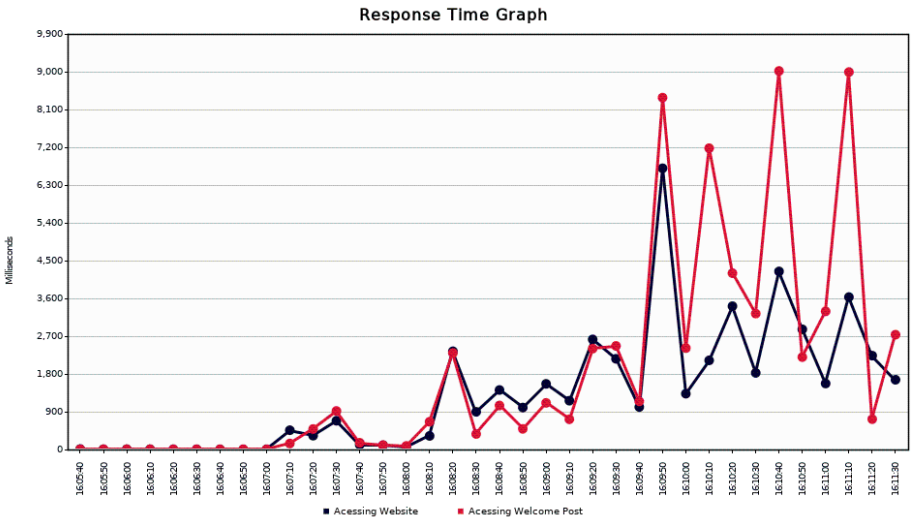


Figura 9.9: Gráfico Tempo de resposta para 5000 clientes concorrentes

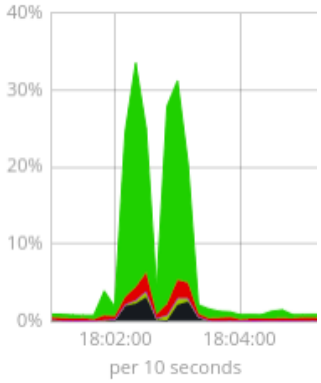
Label	# Sam...	Aver...	Min	Max	Std. D...	Error...	Throug...	Receive...	Sent ...	Avg. ...
Acessing ...	63485	1483	1	132032	4223.81	0.09%	154.2/...	6510.17	17.46	4321...
Acessing ...	62368	2658	1	132746	8008.55	0.35%	162.4/...	3335.36	19.60	2102...
TOTAL	125853	2065	1	132746	6413.16	0.22%	305.8/...	9622.02	35.75	3222...

Figura 9.10: Tabela sumário para 5000 clientes concorrentes

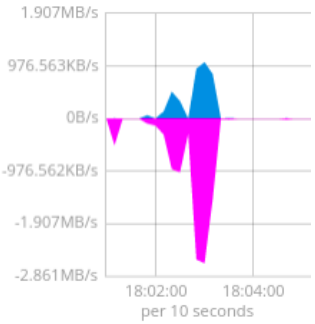
9.3 *Login* e escrita de publicações

9.3.1 100 clientes concorrentes

Em termos de uso de CPU e tráfego de entrada e saída (Figura 9.11), consegue-se observar um uso do CPU bastante elevado (cerca de 30%), o que é justificável pela escrita de uma publicação ser uma operação mais intensiva do que a sua leitura.



(a) Uso de CPU



(b) Tráfego de entrada e saída

Figura 9.11: Dados obtidos com monitorização para 100 clientes concorrentes

Tal como é possível verificar no gráfico 9.12 os tempos de resposta dos vários pedidos mantém-se constantes, não existindo grande variabilidade e encontrando-se o tempo médio em 1 segundo e meio. Também a percentagem de erro não dá quaisquer sinais se saturação, não tendo ocorrido qualquer erro.

Conclui-se desta forma que 100 escritores são suficientes para aninhar o sistema pelo que não nos interessa a realização deste mesmo teste para um número menor de escritores. Contrariamente, para um número maior já seria interessante, mas dada a limitação no número de autenticações permitidas, previamente mencionada, tais testes são impraticáveis. Também no contexto da arquitetura alternativa exposta na secção 9.4 tais testes não se demonstram interessantes pelo mesmo motivo de não se ter atingido o patamar de saturação.

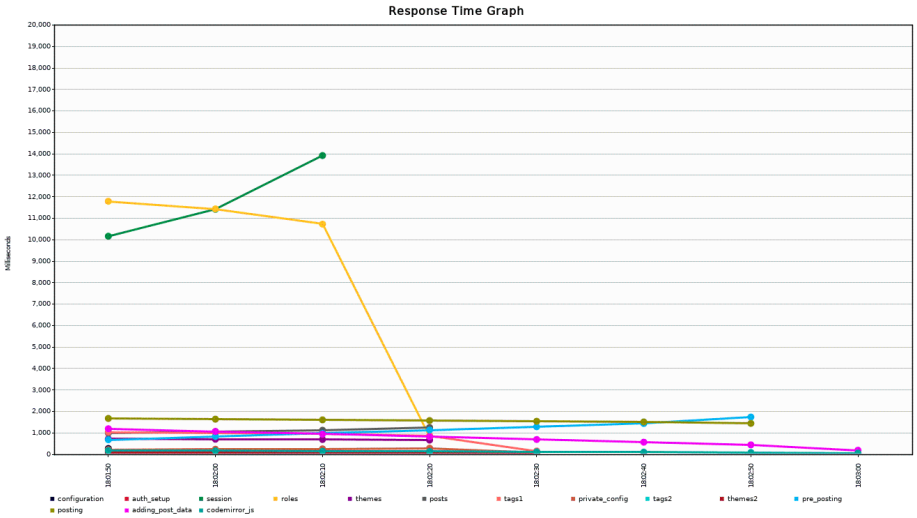


Figura 9.12: Gráfico Tempo de resposta para 100 escritores concorrentes

Label	# S...	Average	Min	Max	Std. D...	Error %	Throu...	Receiv...	Sent ...	Avg. ...
configura...	100	273	133	369	60.48	0.00%	266.7...	132.55	129.69	509.0
auth_setup	100	142	9	226	63.64	0.00%	235.3...	82.95	116.04	361.0
session	100	13919	982	15127	2530.31	0.00%	6.4/sec	3.37	3.64	534.7
roles	100	1284	26	13203	2263.21	0.00%	6.3/sec	4.70	3.85	761.7
themes	100	664	19	1208	288.81	0.00%	23.2/s...	23.79	14.45	1050.3
posts	100	1252	45	1890	480.95	0.00%	17.6/s...	20.46	13.64	1187.5
tags1	100	754	19	1476	497.07	0.00%	21.2/s...	14.10	12.76	680.5
private_c...	100	175	7	506	184.23	0.00%	27.5/s...	9.55	16.71	356.0
tags2	100	82	18	251	65.54	0.00%	27.4/s...	12.73	17.84	476.2
themes2	100	78	15	297	62.67	0.00%	27.4/s...	26.95	16.26	1007.0
pre_posti...	100	1738	44	2749	1041.09	0.00%	15.9/s...	5.85	9.84	377.0
posting	100	1442	97	2171	508.53	0.00%	11.9/s...	11.67	20.55	1002.8
adding_p...	100	403	74	1145	363.88	0.00%	10.7/s...	21.61	47.70	2061.4
codemirr...	100	134	21	454	137.19	0.00%	10.8/s...	751.50	7.44	7147...
TOTAL	1400	1596	7	15127	3599.34	0.00%	17.9/s...	102.17	17.23	5845.7

Figura 9.13: Tabela sumário para 100 escritores concorrentes

9.4 Arquitetura Alternativa

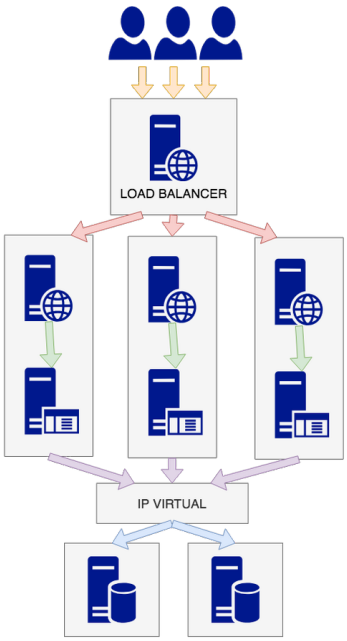


Figura 9.14: Arquitetura alternativa com três servidores *web*

Tendo observado os resultados obtidos e verificando que a carga se reflete negativamente maioritariamente nos servidores *web*, quisemos alterar ligeiramente a arquitetura usada, adicionando um novo servidor *web*. Sendo que agora o balanceamento de carga estará a ser feito entre três servidores *web* ao invés de dois, é esperado que a carga seja melhor distribuída e que, consequentemente, os tempos de resposta diminuam. Isto irá, ou assim esperamos, melhorar a experiência do utilizador da aplicação.

Tal como mencionado em cima os testes realizados em cima apenas incidirão sobre a leitura de publicações. Um vez que o ponto de eficácia máxima deduzido se encontra entre os 3000 e 5000 leitores, os testes que aqui se realizarão serão também relativos a esses dois valores. Pretendemos com estes testes tentar obter um aumento da eficiência do sistema, isto é empurrando a curva que a representa (débito x tempo resposta) segundo o eixo do débito positivo e segundo o eixo do tempo de resposta negativo.

9.4.1 Leitura de publicações

3000 clientes concorrentes

Comparando as tabelas 9.15 e 9.16, para 2 e 3 servidores respetivamente, facilmente se verifica que, quando passámos para a arquitetura alternativa com mais um servidor web/aplicacional, o tempo de resposta diminui e o débito aumenta, face ao mesmo número de utilizadores. Este facto vem comprovar que, efetivamente, ouve um ganho de eficiência significativo.

O tempo de resposta médio diminuiu de 553 milissegundos para 344 milissegundos, isto é, os pedidos passaram a ser respondidos com igual sucesso (0% de erro) em quase metade do tempo. O ganho em débito foi de cerca 27 pedidos respondidos por segundo, atingindo o maior valor alcançado.

Label	# Sam...	Aver...	Min	Max	Std. D...	Error...	Throug...	Receive...	Sent ...	Avg. ...
Acessing ...	58957	394	1	8876	841.68	0.00%	164.3/...	6938.19	18.61	4325...
Acessing ...	58957	712	1	10577	1295.46	0.00%	165.7/...	3413.03	20.06	2109...
TOTAL	117914	553	1	10577	1103.85	0.00%	323.0/...	10149....	37.85	3217...

Figura 9.15: Tabela sumário para 3000 clientes concorrentes e 2 servidores web

Label	# Sam...	Aver...	Min	Max	Std. D...	Error...	Throug...	Receive...	Sent ...	Avg. ...
Acessing ...	63916	261	1	8316	550.34	0.00%	178.1/...	3033.24	20.70	1743...
Acessing ...	63916	427	1	7728	764.42	0.00%	179.9/...	3435.79	22.31	1956...
TOTAL	127832	344	1	8316	671.21	0.00%	350.7/...	6334.66	42.12	1849...

Figura 9.16: Tabela sumário para 3000 clientes concorrentes e 3 servidores web

5000 clientes concorrentes

À semelhança do que aconteceu para 3000 utilizadores, também para 5000 se pode observar uma significativa melhoria na eficiência, uma vez que o tempo de resposta volta a diminuir para configuração de 3 servidores *web* e o débito volta a aumentar. Tendo isto em vista podemos concluir que o nosso objetivo de empurrar a curva de eficiência no sentido da otimização foi alcançado.

É denotar que também nesta arquitetura de 3 servidores *web* o débito decresceu de 3000 para 5000 utilizadores o que uma vez mais demonstra que a eficiência máxima se encontra situada para um número de utilizadores compreendida no intervalo destes dois valores. Contudo, se observarmos atentamente a percentagem de erro nas tabelas 9.17 e 9.18, relativas a 5000 utilizadores, é notório um decréscimo a favor da arquitetura alternativa em questão. Tal facto, evidencia que provavelmente essa eficiência máxima deslocou-se no sentido dos 5000 utilizadores.

Label	# Sam...	Aver...	Min	Max	Std. D...	Error...	Throug...	Receive...	Sent ...	Avg. ...
Acessing ...	63485	1483	1	132032	4223.81	0.09%	154.2/...	6510.17	17.46	4321...
Acessing ...	62368	2658	1	132746	8008.55	0.35%	162.4/...	3335.36	19.60	2102...
TOTAL	125853	2065	1	132746	6413.16	0.22%	305.8/...	9622.02	35.75	3222...

Figura 9.17: Tabela sumário para 5000 clientes concorrentes e 2 servidores web

Label	# Sam...	Aver...	Min	Max	Std. D...	Error...	Throug...	Receive...	Sent ...	Avg. ...
Acessing ...	70066	1493	1	131331	3183.61	0.01%	163.7/...	2787.45	19.03	1743...
Acessing ...	68455	2526	1	131132	6673.05	0.16%	163.5/...	3118.39	20.24	1953...
TOTAL	138521	2003	1	131331	5234.40	0.08%	323.7/...	5838.97	38.84	1847...

Figura 9.18: Tabela sumário para 5000 clientes concorrentes e 3 servidores web

Capítulo 10

Conclusão

Neste trabalho prático, foi realizada uma análise aprofundada à aplicação de *blogging* Ghost e uma instalação completamente automática e distribuída desta na Google Cloud Platform, dotada de ferramentas de monitorização e avaliação experimental. Começamos por estudar a sua arquitetura e averiguamos quais as operações críticas, isto é, os pedidos mais comuns e quais o componentes que caso se aumente drasticamente o número de clientes irão ter um pior desempenho. Essencialmente reconhecemos os recursos com uma carga mais intensiva e que podem constituir um gargalo para o sistema.

Considerando as conclusões tiradas, estabelecemos quais os componentes que necessitam de replicação e o padrão de distribuição que achamos que responde melhor às necessidades do sistema e nos garante as duas propriedades mais importantes de um sistema distribuído: tolerância a faltas e escalabilidade. Posto isto, apresentamos as formas de comunicação dos componentes e quais as configurações necessárias entre eles ou mesmo internas que são necessárias para implementar o anteriormente proposto.

Com a distribuição planeada, a instalação dos componentes na Google Cloud Platform foi-nos facilitada, sendo agora possível instalar e configurar o Ghost executando um número reduzido de passos manuais, graças ao uso da ferramenta Ansible. De forma a melhor observar as atividades no sistema, foram usadas as ferramentas Elasticsearch, Kibana e Beats, que nos permitiram observar a utilização de recursos por parte de cada componente e também o seu desempenho. Com vista a ser usado num cenário de testes, utilizou-se o JMeter para fazer testes de carga à aplicação, avaliando assim o desempenho da nossa instalação face a um grande número de clientes a tentar aceder ao *blog*.

Acreditamos ter atingido os objetivos pretendidos e deixado uma base sólida caso no futuro se considere outro padrão de distribuição, outras configurações ou adição de novos componentes.

Com mais tempo, podíamos ter explorado configurações da base de dados que permitissem aumentar o seu desempenho, uma vez que nos focamos mais na otimização do servidor *web*. Para além disso, na configuração e instalação automáticas, poderíamos ter feito uso de um *storage adapter*, de modo a tirar

partido de um sistema de ficheiros remoto, podendo este ser replicado, por exemplo, para garantir que caso haja uma falha que faça dados serem perdidos, estes possam ser recuperáveis. Outra possibilidade seria a replicação do balanceador de carga, uma vez que este ainda constitui um único ponto de falha.