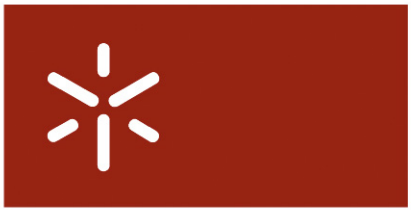

Universidade do Minho

Universidade do Minho

Infraestruturas de Centros de Dados

High Availability Wiki.js Deployment

Angélica Freitas, A83761
Catarina Gil, A85266
Gonçalo Ferreira, A84073
Rui Cunha, A86009

Grupo 8

24 de janeiro de 2021

Conteúdo

Lista de Figuras

Lista de Tabelas

1 Introdução

Na atualidade, uma das principais preocupações com qualquer aplicação é assegurar que esta está disponível a maior quantidade de tempo possível, assegurando assim um serviço contínuo e sem interrupções. No entanto, garantir alta disponibilidade não é fácil, dado que é necessário garantir o funcionamento de todo o serviço mesmo na eventualidade de uma falha num componente da infraestrutura.

Para além de assegurar alta disponibilidade, um dos pontos mais importantes em qualquer aplicação é o seu desempenho, sobretudo quando estas fornecem serviços a um grande número de utilizadores e onde lidar com tamanho *workload* é um desafio.

De forma a lidar com tais problemas, primeiramente é necessário identificar quais os *SPOFs* (*Single Point Of Failures*) destas aplicações, visto que se tratam de uma parte do sistema que em caso de falha, comprometem o sistema como um todo. E, por fim, identificar os componentes mais críticos das aplicações que comprometem o desempenho das mesmas, para que posteriormente estes possam ser tratados.

Assim, ao longo deste relatório é realizada uma análise da aplicação *Wiki.js*, uma plataforma para construir páginas *wiki*, com vista a construir uma solução de alta disponibilidade e alto desempenho da mesma com recurso à plataforma da *Google Cloud*.

Para terminar, são ainda apresentados alguns testes de *performance* de forma a poder compreender qual o impacto das soluções utilizadas para tratar os problemas apresentados.

2 Arquitetura da Aplicação Wiki.js

Como referido anteriormente, a *Wiki.js* é uma aplicação desenvolvida para criar páginas *wiki*, desde a sua construção ao seu *host*. Assim, esta é construída utilizando *JavaScript* e é executada em *NodeJS*.

De forma a poder funcionar, esta aplicação requer, como muitas outras, um sistema de Base de Dados, que neste caso pode ser um de entre múltiplas opções. No entanto, aquele que é recomendado é o *PostgreSQL*, pelo que decidimos utilizar este para a nossa análise e instalação.

Deste modo, a aplicação pode ser dividida em dois componentes principais: o *Web Server* encarregue de disponibilizar a *interface web* e de lidar com todas as interacções do utilizadores e a Base de Dados usada para armazenar toda a informação necessária para o funcionamento da *wiki*, como todas as páginas e informações sobre administração.

3 Pontos Críticos e Falhas de Desempenho

Face à análise da arquitectura, nota-se que é susceptível a falhas de desempenho com os componentes base do sistema. Assumindo uma arquitectura base constituída apenas por um *Web Server* e uma Base de Dados, existem diversos pontos críticos, que podem comprometer a disponibilidade do serviço e limitar o seu desempenho, e assim, de forma a resolvê-los é primeiramente necessário identificá-los.

O primeiro ponto crítico do sistema trata-se precisamente da existência de um único servidor *web*, dado que na eventualidade deste sofrer uma falha e ficar indisponível, deixa de ser possível continuar a usufruir do serviço.

Da mesma forma, a existência de um só servidor encarregue de correr a Base de Dados sofre o exactamente o mesmo problema, levando assim a que uma falha neste cause a paragem completa de todo o sistema. Existindo também o risco acrescido de perder todos os dados contidos na Base de Dados sem a hipótese de serem recuperados, caso essa falha se tratar de algo grave.

Em termos de desempenho, mais uma vez o grande problema encontra-se num único servidor *web*, que necessita de ser capaz de lidar com todos os pedidos enviados à aplicação, gerando assim um *bottleneck*, pelo qual o funcionamento de todo o sistema é afectado.

Estes pontos críticos irão assim afectar a disponibilidade e o desempenho do sistema como um todo, contribuindo assim para a sua eventual inactividade e impedimento de serviço aos utilizadores.

Para contornar estes obstáculos, é imperativo a utilização de múltiplos servidores *web*, pelos quais a carga de pedidos à aplicação possa ser repartida de forma assegurar melhor desempenho, mas também mecanismos que permitam tornar a aplicação resiliente, garantindo que a mesma seja capaz de continuar em funcionamento mesmo no evento de múltiplos dos seus componentes falharem.

4 Solução de Alta Disponibilidade e Desempenho

Após analisados quais os pontos críticos de funcionamento da aplicação, que podem corromper a sua disponibilidade e desempenho, é agora necessário criar uma solução que permita resolver estes problemas. Para tal, serão introduzidos sistemas que garantem um balanceamento de carga por diversos servidores alternativos e outros que asseguram a continuação de serviço na eventualidade da falha de um componente do sistema.

4.1 Arquitetura

Assim, após diversas alternativas estudadas e testes realizados, obtemos uma arquitectura que continua a poder ser divida em duas partes distintas: *Front End* e *Back End*. No entanto, o sistema é um pouco mais complexo. O *Front End* é agora constituída por 3 *Web Servers* distintos capazes de fornecer o mesmo serviço em simultâneo.

Por sua vez, existem agora dois servidores de Base de Dados, no entanto, apenas um se encontra em funcionamento de cada vez. Assim, os *Web Servers* comunicam apenas com aquele que se encontra activo no momento. Existe também agora uma grande diferença no Sistema de Base de Dados atual que é o facto do *file system* não se encontrar na mesma máquina onde a Base de Dados se encontra a correr, mas sim em duas outras que funcionam em modo de replicação.

De forma a atingir este funcionamento, foi necessário recorrer a várias ferramentas e a diversos mecanismos que permitem garantir que todas estas configurações funcionam em conjunto. Para distribuir todos os pedidos dirigidos à aplicação pelos diversos *web servers* recorremos ao uso do *Cloud Load Balancing*, um serviço da *Google Cloud* explicitamente desenvolvido para realizar o balanceamento de carga do tráfego externo dirigido a aplicações. Por sua vez, de forma a grantir que o tráfego é apenas direccionado ao servidores que estão disponíveis, existe também um serviço de *Health Check* que realiza uma verificação periódica do funcionamento dos servidores *web*, garantindo que estes respondem a pedidos *HTTP*.

No que toca ao funcionamento da Base de Dados, existem agora duas máquinas capazes de correr o *SBD*, sendo elas o *ha-cli1* e o *ha-cli2*. Estas funcionam em modo de *failover*, de tal modo que na eventualidade daquela que se encontra ativa falhar, imediatamente o seu lugar é tomado pela outra. Para ser possível que tal aconteça, estas são geridas por um *High-Availability Cluster* da *Red Hat*, que monitoriza o funcionamento de ambas, e que na necessidade de migrar a Base de Dados de uma máquina para a outra, este encarrega-se de migrar os serviços necessários, sendo neste caso o próprio *PostgreSQL* e de montar o sistema de ficheiros correto. De forma a garantir que esta troca de *host* de Base de Dados seja *invisível* para os clientes, existem dois outros sistemas que garantem tal funcionalidade. Em primeiro lugar os pedidos provenientes dos *web servers* são realizados através de um *Cloud Load Balancing* desta vez interno, mas que em vez de executar um balanceamento de carga simplesmente se limita a redireccionar os pedidos para o servidor que se encontra activo. Mais uma vez a verificação de qual o servidor que se encontra ativo é realizada com um serviço de *Health Check*, mas que desta vez testa a conexão *TCP* na porta correspondente à Base de Dados. Em segundo lugar, temos o *Distributed Replicated Block Device* para realizar o armazenamento e replicação dos dados nas máquinas *drdb1* e *drdb2*, com as quais ambos os servidores de Base de Dados podem comunicar graças ao uso de *iSCSI Multipath*, que garante uma conexão constante e é também capaz de balancear carga entre as duas réplicas usufruindo da melhor.

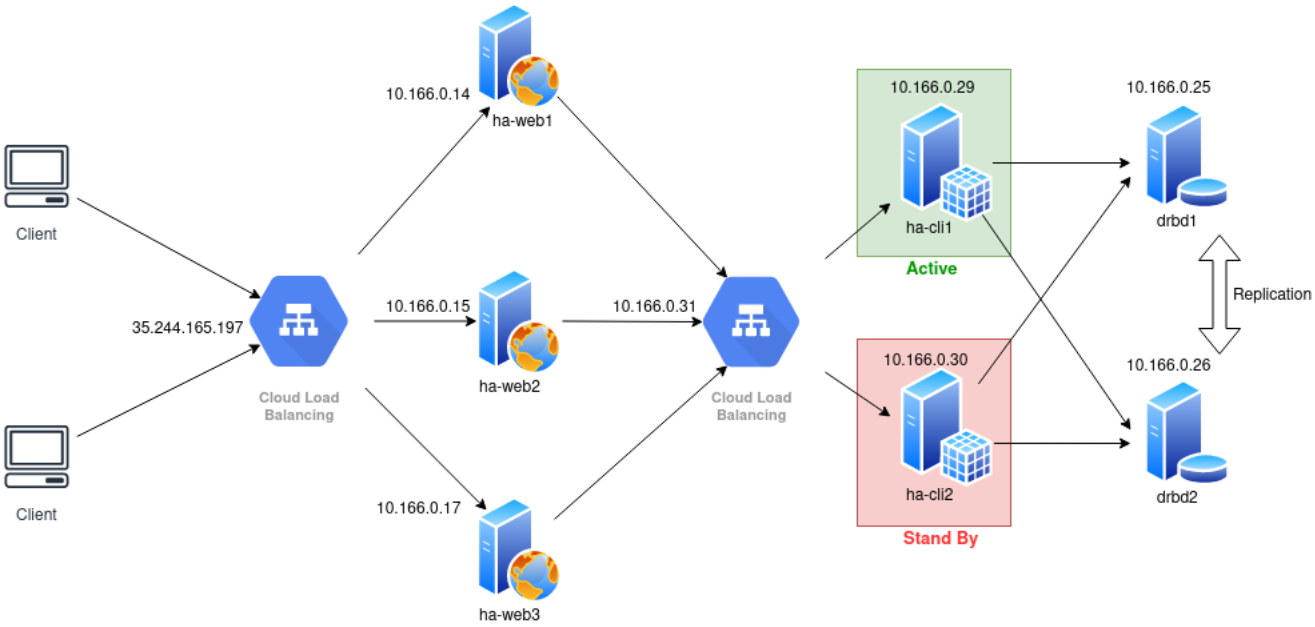

Figura 1: Arquitectura Implementada

4.2 Alto Desempenho

A razão pela qual a implementação desta arquitectura resolve o problema do alto desempenho deve-se ao facto da existência de múltiplos servidores *web*, que passam agora a ser três, contrariamente a um numa arquitectura básica. Todos os pedidos que são direccionados à aplicação são tratados pelos *web servers*, pelo que estes são um ponto fundamental de desempenho. Assim, ao possuirmos múltiplos destes podemos repartir toda a carga de pedidos pelos mesmos, sendo este balanceamento de carga realizado graças ao serviço de *Load Balancing* fornecido pela *Google Cloud*.

4.3 Tolerância a Faltas

A tolerância às faltas dos sistema é garantida graças a um conjunto de medidas. Em primeiro lugar deve-se à existência dos múltiplos servidores *web* já mencionados, embora o principal objectivo destes seja o alto desempenho. Na eventualidade de um destes sofrer uma falha, o *Health Check* imediatamente remove-o das opções do *Load Balance*, assegurando assim a continuação da prestação de serviço apenas com menos um *web server*. Assim, é possível suportar até duas falhas de servidores *web* antes de existir uma perda total de serviço.

A tolerância a falhas da Base de Dados é agora assegurada graças à migração de serviços entre máquinas e ao facto do sistema de ficheiros se encontrar agora separado do sistema de Base de Dados. Quando uma falha no servidor de Base de Dados que se encontra activo acontece, o *cluster* de Alta Disponibilidade da *RedHat* usado no *Back End* detecta a mesma, e imediatamente desloca o serviço para o servidor que se encontra em *standby*, garantindo também que o sistema de ficheiros se encontra agora montado no mesmo. Isto garante assim que a Base de Dados permanece em funcionamento praticamente sem interrupções. Após esta alteração o *Health Check*, que detetou a mesma, informa o *Load Balance*, assegurando assim

que os pedidos à Base de Dados rezalizados pelos *web servers* são agora redireccionados para o servidor de *standby* que passa agora a ser o activo.

Relativamente aos dados da Base de Dados que se encontram agora nas suas próprias máquinas, o uso de *Distributed Replicated Block Device* garante também a continuação do funcionamento do sistema mesma que uma destas sofra uma falha, dado que estando os dados replicados entre as duas, a Base de Dados pode obter os mesmos de qualquer uma.

Embora uma olhar sobre a arquitectura implementada possa apresentar possíveis *spofs* no que toca aos *Cloud Load Balancing* utilizados, este não é o caso, dado que o balanceamento de carga fornecido pela *Google* não se trata de uma máquina física, mas sim de um serviço totalmente distribuído e baseado em *software*, garantindo assim alta disponibilidade em todos os momentos.

5 Testes de Desempenho

De forma analisar qual o impacto dos diversos mecanismos introduzidos na arquitectura desenhada para a nossa solução, decidimos realizar testes de *performance* de forma a avaliar qual o desempenho do sistema face a uma configuração simples. Para tal, criámos assim em primeiro lugar uma configuração base com apenas um *Web Server* e uma Base de Dados e posteriormente uma outra configuração recorrendo à nossa solução. Em ambas estas implementações utilizámos máquinas com configurações idênticas, para que os testes fossem realizados de forma equilibrada, optando assim por instâncias do tipo *n1-standard-2*, equipadas com 2 *vCPUs* e $8GB$ de memória *RAM*.

De forma a possuirmos dados aos quais pudessem ser realizados pedidos *HTTP*, criámos duas páginas *wiki* em ambas as configurações, garantindo que estas são diversificadas em termos de tamanho de conteúdo, numa delas colocámos um pequeno poema de Fernando Pessoa e numa outra o total conteúdo dos Lusíadas.

Para realizar os testes decidimos recorrer ao *jmeter*, uma ferramenta desenhada para realizar testes de carga e analisar o desempenho de aplicações *web*. De seguida, decidimos executar os testes com três diferentes tipos de *workload* para analisarmos o comportamento do sistema em diversos cenários. Em primeiro lugar executámos apenas *GETs* da página dos Lusíadas, nomeadamente 1000 executados por 100 clientes. Posteriormente com a mesma configuração de número de clientes e pedidos executámos apenas *POSTs*, adicionado comentários às páginas. E por fim, ainda com o mesmo número de clientes e pedidos executámos *GETs* e *POSTs* em simultâneo em ambas as páginas criadas.

5.1 Resultados

WorkLoad	# Samples	Avg RT	Min RT	Max RT	Error %	Throughput
Get Lusiadas	100000	1440	133	1867	0.0	69.00344534202593
Post Comment	100000	55	1	492	0.0	1562.4267612455667
Mista	300000	517	1	1728	0.0	191.9551592747934

Tabela 1: Resultados da Configuração Simples

WorkLoad	# Samples	Avg RT	Min RT	Max RT	Error %	Throughput
Get Lusiadas	100000	489	14	2458	0.0	198.787792044115
Write Comment	100000	18	2	238	0.0	3726.198904497522
Mista	300000	181	2	1447	0.0	541.207542268309

Tabela 2: Resultados da Configuração Complexa

Como podemos observar pelos resultados obtidos, a implementação da arquitectura de alto desempenho e disponibilidade permitiu aumentar a *performance* do sistema de forma substancial. Não só foi possível obter um débito muito mais elevado de pedidos, mas os tempos de resposta foram também encurtados de forma bastante significativa.

No que toca ao *Throughput*, este aumento era esperado de ser observado, já que agora temos essencialmente o triplo de servidores *web* a atender pedidos, permitindo assim que exista uma muito maior capacidade para lidar com os mesmos. No entanto no que toca ao tempo de resposta, inicialmente considerámos que fosse possível observar um ligeiro aumento do mesmo devido aos mecanismos introduzidos, visto que é necessário realizar o *load balancing* de todos os pedidos que chegam à aplicação e a Base de Dados é agora também ligeiramente mais lenta devido à replicação de dados. Claramente isto não se verificou, deixando-nos assim com uma solução muito mais eficaz do que esperávamos inicialmente.

6 Aspetos a Melhorar

Com a arquitetura atual, sempre que um pedido é feito os dados são lidos pela base de dados ativa. Sendo assim, não existe *load balancing* nas mesmas, o que, por sua vez, se torna um *bottleneck*, que irá afectar o desempenho da aplicação.

Ora, para melhorar ainda mais a nossa aplicação, seria uma boa ideia implementar uma Base de Dados numa configuração *Master-Slave*. Esta consiste em ter uma base de dados *Master*, à qual estão ligadas outras bases de dados, denominadas por *Slaves*, existindo replicação de dados da *Master* para as *Slaves*. Dependendo dos objectivos e propósito da aplicação, esta replicação pode ou não ser feita em tempo real.

Assim, como os *Slaves* podem também efectuar pedidos de leitura, isto permite que os pedidos possam ser repartidos pelas diversas Base de Dados, representando uma sobrecarga inferior face a uma única base de dados, melhorando assim o desempenho da aplicação. O único problema desta implementação é que há possibilidade de perda de informação caso a base de dados *Master* esteja a efetuar uma operação de escrita e sofra uma falha durante a mesma.

7 Conclusão

Após a realização deste projecto, observamos não só a importância de aplicações que fornecem alta disponibilidade e desempenho, mas também como esses objectivos podem ser atingidos.

Assim, primeiramente passamos por descrever a arquitectura da plataforma sugerida e identificar os seus pontos críticos e falhas de desempenho. De seguida, desenvolvemos uma arquitectura com o principal objectivo de colmatar os pontos críticos identificados anteriormente. Por fim, avaliamos o desempenho da arquitectura proposta em relação à arquitectura base e concluímos sobre o impacto causado.

Embora nos encontremos satisfeitos com o resultado final, ao longo deste projecto, deparamo-nos com algumas dificuldades. Sendo a primeira vez que o grupo realizou uma aplicação deste calibre na *Google Cloud*, tivemos alguns contratempos com a implementação do *load balancing*. Nomeadamente devido ao facto de inicialmente planearmos utilizar *HAProxy* com recurso a *keepalive* para garantir a alta disponiblidade do mesmo. No entanto, devido às limitações impostas pela rede da *Google Cloud* não fomos capazes de colocarmos estes em funcionamento, levando-nos assim posteriormente a optar pelo *Cloud Load Balancing*.

Semelhante aos problemas com o *keep alive*, tivemos também algumas dificuldades ao tentarmos utilizar *IPs* flutuantes no *cluster* de alta disponibilidade no *Back End*, o que mais uma vez nos levou a recorrer ao serviço de Balanceamento de Carga fornecido pela *Google Cloud* para contornar esse problema.

Através dos testes realizados na secção 5, verificamos um aumento significativo de *throughput* e diminuição de tempo de resposta em relação ao sistema de configuração base, permitindo inferir que o balanceamento de carga ajudou de facto no desempenho. Apesar de obtermos a mesma percentagem de erro em ambas as configurações, a disponibilidade da solução criada é, sem dúvida, maior que a da configuração base.