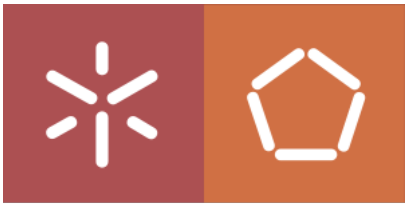

Universidade do Minho
Escola de Engenharia

Planeamento de uma Infraestrutura Wiki.js

Infraestruturas de Centros de Dados

Engenharia de Aplicações

Grupo 4

Alexandre Miranda a84462

Alexandre Ferreira a84961

João Azevedo a85227

Paulo Araújo a85729

Braga,
Janeiro 2021

Conteúdo

Lista de Figuras

Lista de Tabelas

1 Introdução

O desenvolvimento de *software* constitui uma primeira fase do processo iterativo de construção de uma infraestrutura que inclui planear, desenvolver, configurar e automatizar a instalação e manutenção de aplicações em grande escala.

O planeamento de todo este processo vem trazer formas de garantir um conjunto de propriedades/requisitos como permitir alta disponibilidade, alto desempenho e bastante resiliência dos componentes que compõem o sistema de modo a evitar custos relacionados com *Downtime* e consequente inacessibilidade no acesso aos dados, serviços, aplicações, entre outros problemas que queremos evitar quando levantamos questões como "O nosso negócio consegue sobreviver se os dados estiverem inacessíveis?", "Quanto tempo demora a recuperar os servidores?" e "Como manter uma boa reputação?", pois não queremos correr o risco de, constantemente, forçarmos os clientes a optar por um serviço alternativo.

Ao longo deste documento vamos apresentar todo o processo de planeamento e operacionalização da plataforma *Wikijs*, com vista a apresentar alta disponibilidade de serviços por ela prestados e, ao mesmo tempo, alto desempenho através do balanceamento de carga entre servidores.

Esta aplicação constitui um componente de *software* extremamente fléxivel e *open-source*, desenvolvida em *Node.js* e escrita, deste modo, em *Javascript*, pelo que todo o processo de configuração de várias bases de dados, desenvolvimento simples e modular de páginas, adaptação para ferramentas de controlo de versão como *git*, ambiente bastante responsivo para utilizadores, controlo de autenticação, controlos de acesso e permissões de *superuser*, entre outras facilidades, tornam este sistema uma opção bastante viável para empresas que pretendem criar documentação estática para programadores/ engenheiros de *software*, uma necessidade de qualquer infraestrutura em grande escala.

Um segundo processo aliado a todo o planeamento de um sistema como este é a definição de uma fase iterativa de testes de desempenho para perceber se as decisões de *design* tomadas primariamente permitem proteger a aplicação de pontos únicos de falha, também perceber se a carga está a ser distribuída de forma eficiente, isto é, garantir uma correta utilização dos recursos balanceando a escalabilidade horizontal e vertical de cada máquina e, por fim, garantir mecanismos e políticas de tolerância a faltas.

Mais se diz que achamos por bem demonstrar este processo extensivo, iterativo e incremental sob a forma de várias implementações, ou seja, não adotamos diretamente a estrutura mais eficiente, em termos teóricos, no sistema, mas fomos experimentando e tirando conclusões de modo a perceber aquela que se adequa mais a uma solução que cumpra todos os requisitos especificados.

2 Arquitetura e Componentes da Aplicação

A aplicação estudada inclui-se num conjunto de componentes de *software* com vista a disponibilizar um ambiente responsivo e bastante customizável para equipas de programadores e engenheiros, segundo uma política *open-source*, de criação de páginas estáticas de documentação em *Markdown*.

A *Wikijs* é, deste modo, uma aplicação moderna e extremamente poderosa, desenvolvida em *Node.js*, escrita em *Javascript*, com imenso suporte para uma configuração fléxivel:

- Contém mecanismos para uma completa revisão e controlo de versões (ex: *git*);
- Suporte para imensas linguagens diferentes o que a torna globalmente acessível;
- Ferramentas e motores de pesquisa bastante adaptados e rápidos, podendo usar componentes *third-party* como *Azure Search* e *ElasticSearch*;
- Módulos de armazenamento com uma vasta escolha de motores de bases de dados;
- Sistema de gestão de utilizadores, permissões e controlo de acesso a *media*;
- Sistema de *feedback* com base em comentários de utilizadores em páginas/conteúdos;

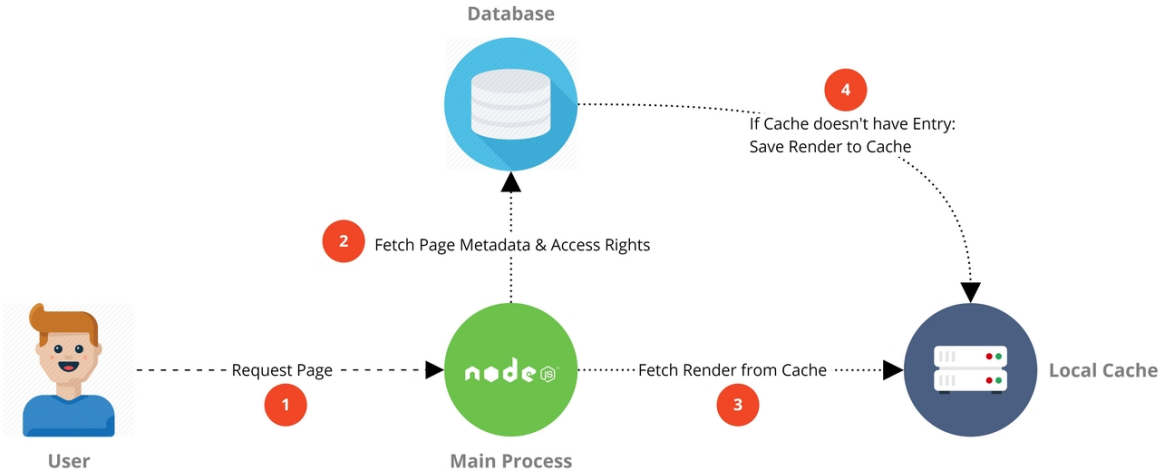

Figura 1: Exemplo de um *request*: "Ver Página".

O software da aplicação não se encontra dividido entre *frontend* e *backend*, relativamente ao armazenamento dos dados tem suporte para diversas base de dados em especial para *PostgreSQL*, escolhida por nós.

Como podemos observar pela figura anterior, o pedido de visualização de uma página, e a grande maioria dos pedidos, primeiramente comunicam com o *frontend*, o *Nodejs* e, se este em seguida necessitar de dados que não se encontram em *cache*, comunica então com a base de dados para obter os dados em falta.

3 Evolução da infraestrutura

3.1 Soluções intermédias e Pontos Críticos

Nesta secção iremos falar sobre os pontos críticos de falha e desempenho que identificamos numa fase inicial do projeto, onde tentamos obter a melhor solução possível, isto é, aquela que permitisse atingir alto desempenho e colmatasse possíveis pontos críticos.

3.1.1 Infraestrutura centralizada

Naturalmente, começamos com uma solução teste onde toda a infraestrutura *wiki.js* se encontrava apenas numa máquina.

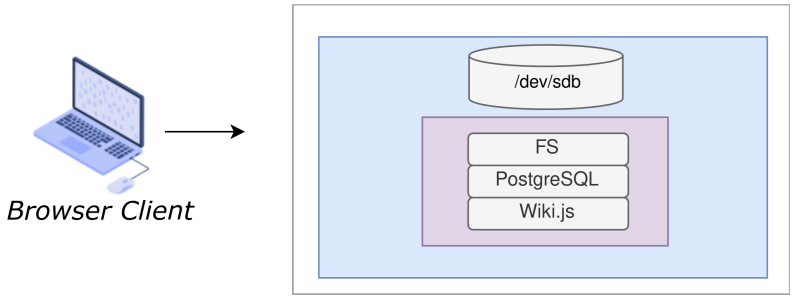

Figura 2: Infraestrutura e componentes centralizada.

Descrição da máquina utilizada

Para esta infraestrutura, visto ainda não estarmos focados no desempenho da aplicação, mas sim apenas em testes iniciais de disponibilidade, foi utilizada uma máquina *f1-micro (1 vCPUj, 0,6 GB de memória).*

Pontos críticos

Facilmente conseguimos perceber o porquê de esta solução não ser a mais aconselhável. Toda a estrutura da aplicação encontra-se em apenas uma máquina, assim sendo, qualquer falha que ocorra fará com que o serviço deixe de estar disponível, até que a máquina volte a estar saudável. Seria ainda penoso para o aumento do desempenho da aplicação, esta infraestrutura apenas poderia ser escalável verticalmente, aumentando os recursos da única máquina existente.

Uma das razões que não permite este crescimento é o facto de não existir divisão entre o serviço *frontend/backend graphql* e a base de dados, impedindo que diferentes instâncias tenham acesso aos mesmos dados. Sendo assim o próximo passo a ser realizado, é a divisão descrita anteriormente.

3.1.2 Infraestrutura com divisão explícita da base de dados

De modo a evoluir com a infraestrutura teremos que solucionar os pontos críticos encontrados na primeira solução apresentada (Centralizada).

Para conseguir escalar horizontalmente a nossa infraestrutura necessitamos de fazer uma divisão para conseguir obter, de um lado a base de dados e do outro o serviço em si, que comunica com o cliente e faz os respetivos pedidos à base de dados.

Tal como é apresentado no Guião 2 e 3 das aulas práticas, criamos 2 máquinas *drdb* primárias para permitir uma persistência de dados, no caso de falha de uma destas máquinas. Isto é possível porque sempre que ocorre uma escrita numa das máquinas, essa mesma escrita é replicada na outra. Pensamos não existir dúvidas quanto à importância da replicação dos dados e a tolerância a falhas que os mesmos disponibilizam, por isso é que o grupo partiu, desde logo para a solução com duas máquinas *drdb*, evitando o estágio em que apenas se iria utilizar uma máquina para armazenamento dos dados.

De modo a permitir uma melhor gestão, tal como no guião, criamos uma máquina *iSCSI*, onde se encontra o *File system*, o serviço base de dados *Postgres* e o *iSCSI client - multipath*.

Tendo as máquinas que gerem e armazenam a base de dados já definidas, tivemos agora que nos preocupar com o *web server*, ou seja, a máquina onde estaria a aplicação *wiki.js*. Como não há uma divisão explícita entre o *frontend* e o *backend*, esta máquina será idêntica à solução centralizada, diferindo apenas na divisão da base de dados.

Facilmente conseguimos realizar a ligação entre o servidor web, que contêm o serviço, e a sua base de dados *postgres*, já que a *wiki.js* permite indicar onde se encontra a sua base de dados, ou seja, o endereço da máquina e a porta onde o serviço de base de dados se encontra a correr. Neste caso, foi apenas indicar que a máquina a utilizar seria a *iSCSI*.

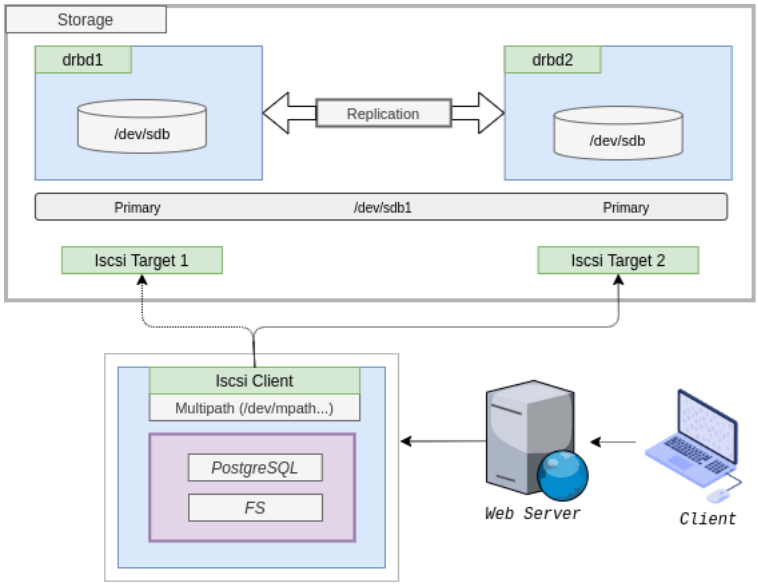

Figura 3: Infraestrutura com divisão explícita do serviço base de dados.

Descrição das máquina utilizadas

Com esta implementação a nossa infraestrutura ficou organizada com duas máquinas *drdb - f1-micro (1 vCPUj, 0,6 GB de memória)*, uma máquina iSCSI - *f1-micro (1 vCPUj, 0,6 GB de memória)* e uma máquina *web server - f1-micro (1 vCPUj, 0,6 GB de memória)*.

Pontos críticos

Em relação à estrutura apresentada, anteriormente, conseguimos identificar alguns pontos críticos, nomeadamente, na máquina *iSCSI*. Na possibilidade de falha deixaria de existir o serviço de base de dados, impossibilitando o normal funcionamento do serviço.

Mesmo tendo replicação do conteúdo da base de dados, isto é *drbd*, a máquina *iSCSI* é um ponto de 'estrangulamento' que gere todo o serviço de base de dados.

Para além disso, também existe outro ponto crítico que é o *web server*, este é, tal como a máquina *iSCSI*, um ponto onde qualquer falha deixaria o serviço completamente indisponível.

3.1.3 Infraestrutura com *failover* na *storage* - *iSCSI cluster*

Tal como exposto na subsecção anterior, na possibilidade da máquina *iSCSI* falhar, todo o acesso à base de dados seria impossibilitado, deixando a aplicação indisponível.

Como tal, existe a necessidade de implementar uma solução que permita colmatar esta possível indisponibilidade. Para tal, e seguindo o Guião 5 realizado nas aulas práticas, procedemos à criação de um *cluster iSCSI*. Teríamos então 2 máquinas *cli1* e *cli2*, onde uma funcionaria como *backup* da outra, neste caso *cli2* seria *backup* de *cli1*. Caso a máquina a servir o *web server* (*cli1*) falhasse, todos os serviços na máquina *cli1* seriam transferidos para a máquina *cli2*, para que a disponibilidade da aplicação se mantivesse e os serviços continuassem a correr.

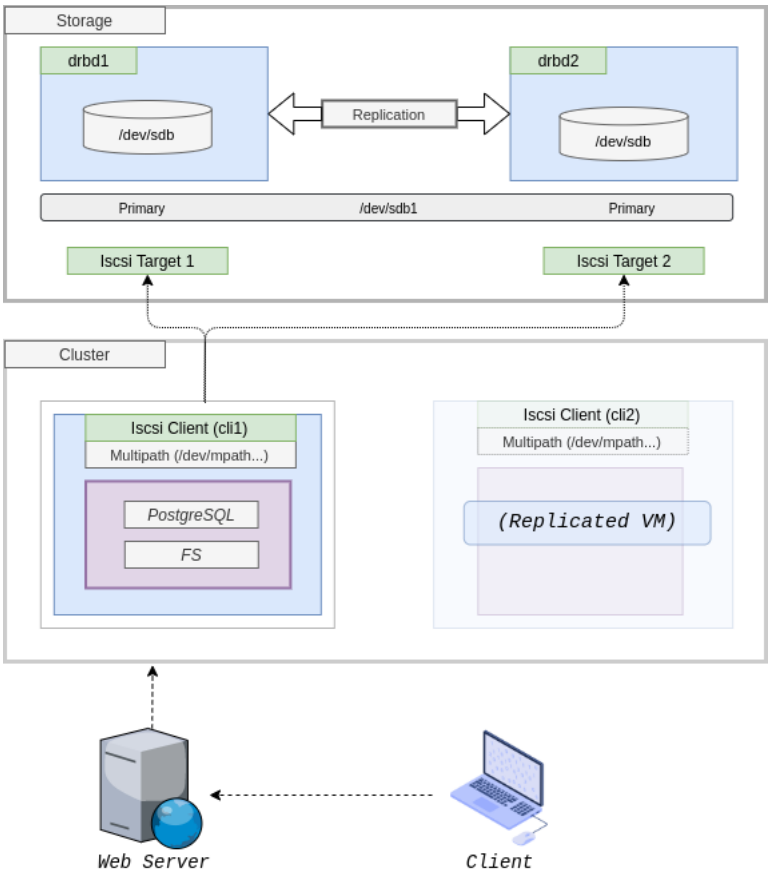

Figura 4: Infraestrutura com *failover* na *storage*

Durante a instalação dos serviços deparámo-nos com problemas de memória que não permitiam a finalização da implementação dos serviços de *failover*. Para ultrapassar o problema tivemos de aumentar a memória RAM da máquina que se encontrava a 640MB, para 1GB, o que nos permitiu acabar a configuração e testar se os serviços continuavam *online* quando a máquina *cli1* falhava. Depois deste problema decidimos, não só melhorar

as máquinas referentes ao *cluster*, mas também as máquinas referentes ao armazenamento dos dados (*drbd1* e *drbd2*).

No entanto, e enquanto implementávamos o serviço de *failover*, deparámo-nos com um problema referente ao *ipaddr2*, já que usando os serviços da GCP -*Google Cloud Platform*, não conseguíamos criar uma interface de rede que permitisse ser partilhada pelas duas máquinas, ou seja, ao qual o cliente se ligaria sempre, independentemente se era a máquina *cli1* ou *cli2* que prestava o serviço. Cada uma das máquinas, como é óbvio, continuaria a ter o seu próprio endereço de rede privativo.

Perante este obstáculo, expusemos esse problema ao docente que nos indicou que poderíamos usar recursos da GCP de modo a conseguirmos ter uma ferramenta que tivesse a mesma funcionalidade do *ipaddr2*.

Para tal, utilizamos o mecanismo denominado 'Balanceamento de carga', que permite indicar qual as máquinas a 'balancear', qual as definições de integridade, e se algumas das máquinas pertenceria a um grupo de *failover*. Como o que necessitamos é que o serviço de base de dados continue ativo, mesmo durante uma falha (*failover*), tivemos então de indicar que a máquina *cli1* seria a principal e que a *cli2* pertenceria ao grupo de *failover*. No final da configuração o balanceador ficou com as seguintes características:

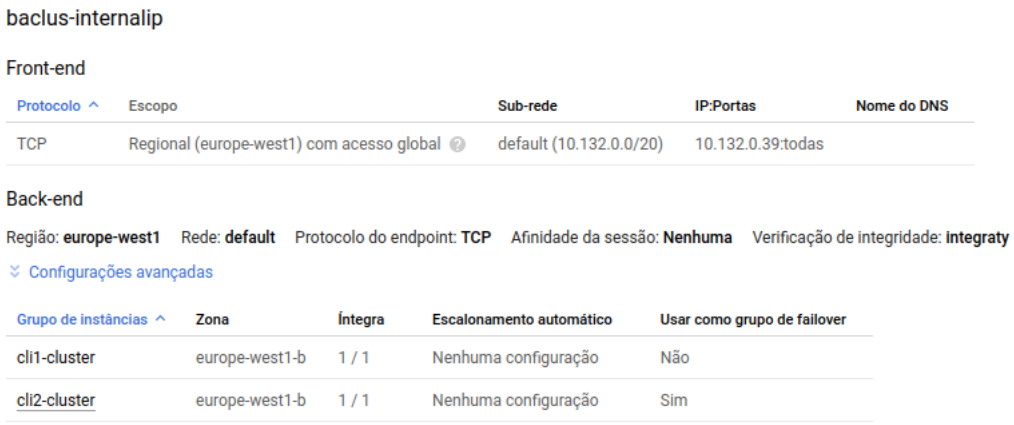

Figura 5: Características do balanceador *failover* interno para a máquina *cli1* e *cli2*.

O balanceador funciona da seguinte maneira, quando acontece uma falha na máquina *cli1* os serviços vão passar, automaticamente, para a máquina *cli2* e o balanceador irá detetar que a máquina *cli1* não se encontra íntegra e passará os pedidos para a máquina *failover cli2*. Deste modo conseguimos ter um serviço com a mesma funcionalidade do *ipaddr2*, totalmente funcional.

Descrição das máquinas utilizadas

Com esta implementação a nossa infraestrutura ficou organizada com duas máquinas *drdb - e2-micro (2 vCPUs, 1 GB de memória)*, duas máquinas iSCSI - *e2-micro (2 vCPUs, 1 GB de memória)* e uma máquina *web server - f1-micro (1 vCPUj, 0,6 GB de memória)*.

Pontos críticos

Na solução apresentada ainda continuamos com o ponto crítico referente ao *web server*, pois o serviço rapidamente ficará congestionado com uma grande quantidade de pedidos, e em caso de falha desta máquina não existirá serviço a apresentar ao cliente, sendo então um *single point of failure*.

3.1.4 Infraestrutura com elevado desempenho nos *web servers*

Como referido no ponto anterior, o facto de existir apenas uma máquina que faz a gestão de todos os pedidos recebidos e retorna o resultado pretendido para os clientes, faz com que os serviços possam ficar rapidamente indisponíveis.

Portanto a solução adotada pelo grupo afim de eliminar esta lacuna, foi a de fazer uma replicação dos serviços da *wiki.js* para uma outra máquina. Ou seja, teríamos duas ou mais máquinas às quais os clientes poderiam efetuar pedidos, obtendo a mesma resposta esperada, independentemente da máquina à qual foi efetuado o pedido, já que previamente fizemos a divisão entre a aplicação em si e a base de dados.

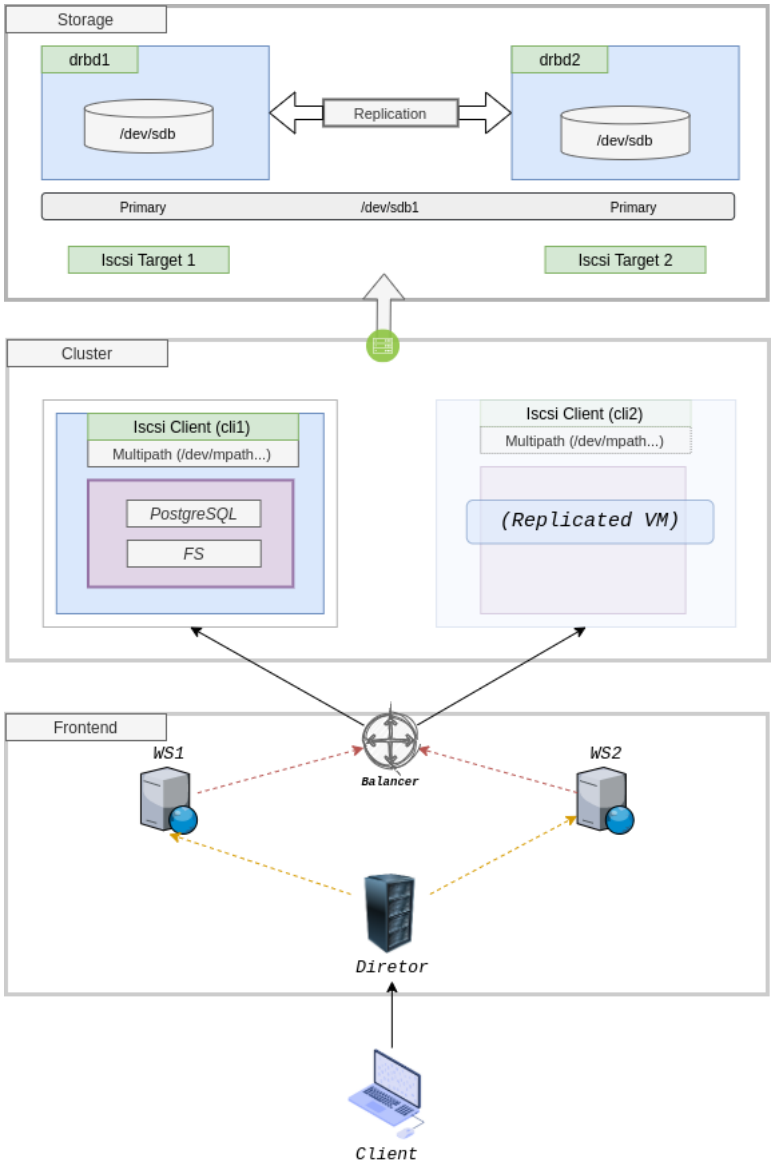

Figura 6: Infraestrutura com elevado desempenho.

Feito isto, era necessário agora fazer o balanceamento de carga entre os *web servers* existentes. O objetivo passaria pela utilização do recurso *IPVS* para obter *load balancing* entre os diversos *web servers*. No entanto, nas máquinas da GCP não conseguimos implementar este recurso, logo tivemos de recorrer a uma outra ferramenta sugerida

pelo docente, o *nginx*, que suporta confiabilidade, escalabilidade e disponibilidade para serviços críticos de produção.

Foi por isso necessário criar uma máquina para suportar esta ferramenta, ou seja, que tivesse implementada uma configuração do *nginx* e permitisse correr estes serviços. Feita esta implementação seria agora possível fazer o balanceamento de carga entre os *web servers*, recorrendo a esta nova máquina denominada diretor.

Descrição das máquinas utilizadas

Apresentada esta nova resolução, a nossa infraestrutura passa assim por conter mais duas máquinas, mais um *web server* (ws2) e o balanceador (diretor). Como já tinham sido implementadas máquinas com melhores recursos para o *cluster*, decidimos desde logo alterar/criar os *web servers* com as seguintes configurações - *e2-micro (2 vCPUs, 1 GB de memória)* e uma máquina balanceador diretor - *g1-small (1 vCPUj, 1,7 GB de memória)*.

Pontos críticos

Com esta nova infraestrutura, conseguimos perceber que ainda existe um ponto crítico capaz de indisponibilizar os nossos serviços. Isto porque ainda é necessário encontrar uma solução para que o balanceador de carga dos *web servers* se encontre sempre ativo.

3.1.5 Infraestrutura com *failover* no balanceador diretor

Como vimos nos pontos críticos da solução anterior, identificamos que a infraestrutura apresentava um *bottleneck* referente ao balanceador. Tal como aconteceu nas máquinas *iSCSI cluster*, vamos ter que criar uma máquina diretor *backup*, que será igual à máquina balanceadora diretor, mas que assumirá os serviços na eventualidade de falha do diretor principal. Para o efeito, e como aprendido nas aulas práticas da unidade curricular, tentamos utilizar o *Keepalive - Virtual Redundancy Routing Protocol*, para conseguirmos ter uma interface a servir as duas máquinas (diretor e diretor *backup*). No entanto, e como aconteceu para o *ipaddr2* nas máquinas *iSCSI*, não conseguimos criar uma interface que servisse as 2 máquinas diferentes. Mais uma vez, recorremos à utilização da ferramenta 'balanceador de carga' do GCP, colocando uma máquina no grupo de *failover*.

Figura 7: Características do balanceador *failover* para a máquina diretor e diretor *backup*

Tal como o primeiro balanceador de *failover* temos então a verificação de integridade das duas máquinas presentes no grupo, em que a máquina diretor *backup* pertence ao

grupo de *failover*. Porém, neste balanceador tivemos que criar e associar um IP externo estático que atenderá todos os pedidos à aplicação - 34.90.55.186, porta 3000.

Descrição das máquinas utilizadas

À infraestrutura já descrita no ponto anterior, apenas adicionamos mais uma máquina diretor, que como é *backup*, é em tudo idêntica à máquina diretor - *g1-small (1 vCPUj, 1,7 GB de memória)*

3.2 Solução final

Depois de identificar todos os pontos críticos das diversas fases para chegar à solução final e os solucionar um a um, achamos que conseguimos construir uma infraestrutura que não apresenta nenhum *single point of failure*, que apresenta uma escalabilidade horizontal e que dispõe de alta disponibilidade, com tolerância a falhas. Conseguimos também obter elevado desempenho, sendo sempre possível acrescentar recursos, nomeadamente *web servers*, sempre que seja necessário responder a uma carga mais elevada que a infraestrutura tenha que atender.

Nota: Foram ainda realizados alguns *scripts* de inicialização das máquinas para que estas imediatamente após o seu início disponibilizassem os serviços desejados. Estes *scripts* encontram-se na subsecção (6.2).

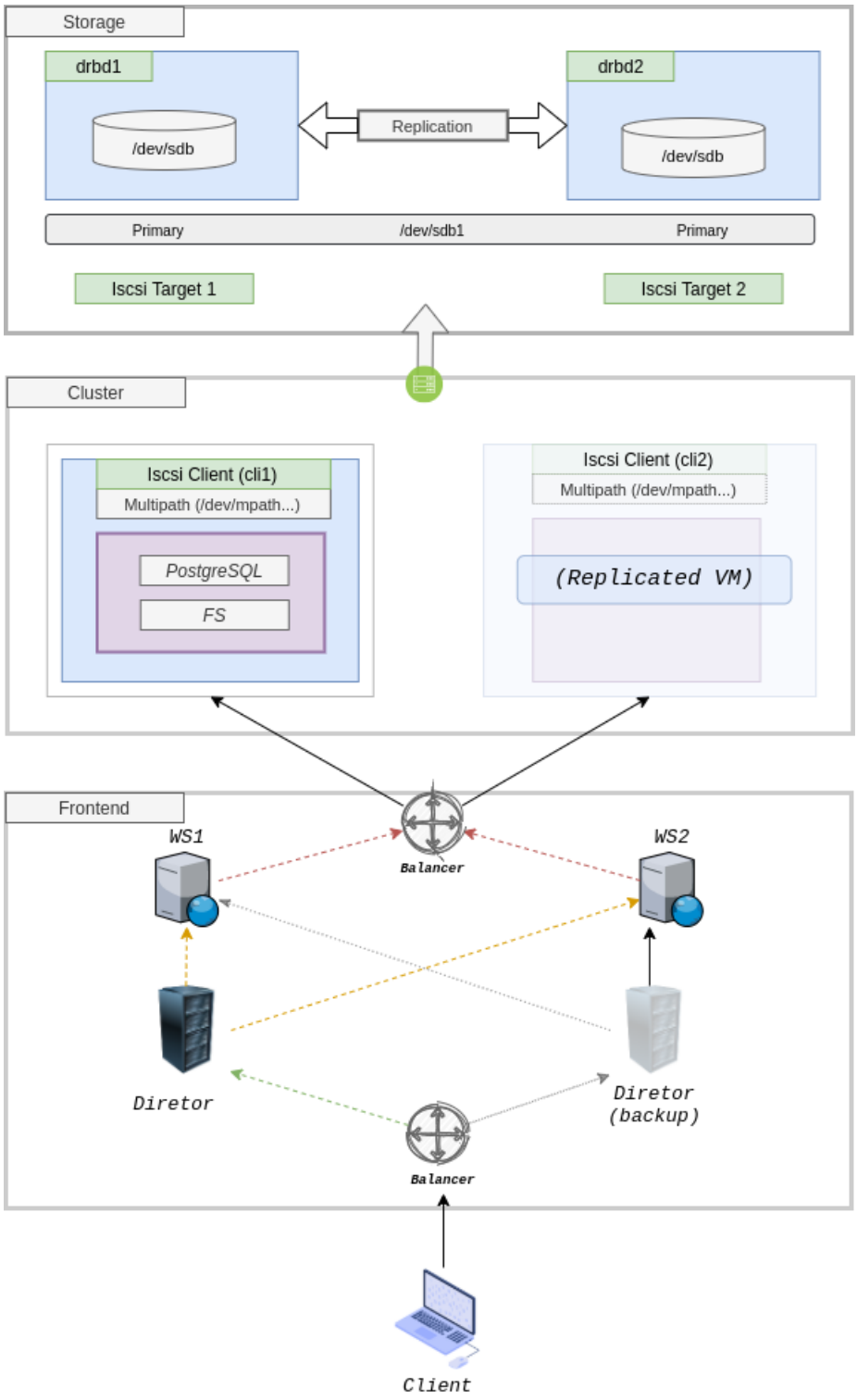

Figura 8: Infraestrutura final de elevado desempenho e alta disponibilidade.

4 Avaliação de Desempenho

Este é um dos pontos mais cruciais no desenvolvimento de uma infraestrutura, visto que uma má avaliação pode levar à indisponibilidade de um serviço ou até, em casos extremos, de toda a aplicação.

Um dos objetivos desta avaliação passa por tentar manter a infraestrutura em funcionamento e com desempenho ideal 24 horas por dia, 7 dias por semana. Portanto o tempo de inatividade ou a latência vão ser características que o grupo vai priorizar para uma melhor avaliação de desempenho, tentando escalar a infraestrutura de forma a diminuir ao máximo estes valores.

Para tal foram utilizadas ferramentas de análise como o *JMeter*, *Selenium* e os gráficos disponibilizados pela *GCP - Google Cloud Platform*.

Quanto aos testes é ainda necessário que estes sejam conclusivos, com isto queremos dizer que é preciso sujeitar o sistema a uma grande carga de trabalho, para percebermos em que medida este responde e como deve ser reestrurado ou não, para obter um melhor desempenho. A realização de testes que não sujeitem o sistema a uma situação de desconforto, pode-nos induzir em erro levando a conclusões precipitadas sobre a performance do sistema.

4.1 Ferramentas utilizadas

A ideia inicial do grupo foi a utilização do *JMeter* para a realização de testes ao *backend* e a utilização do *Selenium* para os testes ao *frontend*.

Visto que a própria aplicação não permite este tipo de divisão, a realização de testes de carga e *stress* utilizando o *Selenium* caiu um pouco por terra, apesar de termos efetuado testes com esta ferramenta.

4.1.1 Selenium

Esta ferramenta permite-nos criar scripts de interação com o *frontend*, através de um *web driver*, ou seja, automatiza a inserção de atributos e o preenchimento de campos da aplicação de forma a que os testes não tenham valores aleatórios, devido ao erro adicionado pela componente humana.

Para a realização destes testes era necessário conhecer um pouco o código *html* das páginas que apresentavam a aplicação da wikijs. De forma a selecionar os elementos nos quais devem ser inseridos ou clicados para a execução do pedido pretendido.

Os testes realizados foram o de *Login*, *Escrever Comentários* e *Recarregar uma página*. Esta foi uma ferramenta bastante interessante no início da sua implementação, mas rapidamente se tornou irrelevante visto que não permitia a utilização de um grande número de *threads*, ou seja, o seu número dependia muito da máquina que estava a ser utilizada.

Ainda mais irrelevante se tornou, porque não existe uma verdadeira distinção entre o *backend* e o *frontend*, mas o conhecimento apreendido pode ser utilizado noutras ocasiões futuras. O código utilizado para fazer os teste encontra-se em (6.1).

4.1.2 JMeter

O JMeter é uma ferramenta que realiza testes de carga e de stress a recursos estáticos ou dinâmicos oferecidos, neste caso, pelo nosso sistema.

A escolha por esta ferramenta prendeu-se pelo grande número de pedidos simultâneos que esta pode realizar à nossa infraestrutura. Poderiam, portanto, ser definidos diferentes número de *threads* para cada teste e verificar o comportamento dos mesmos. Também os dados que são apresentados como resultado foi algo apetecível ao grupo para uma melhor avaliação de desempenho, visto resultar em diferentes variáveis que nos permitem tirar conclusões mais corretas sobre o comportamento do sistema.

Para a realização de um pedido é necessário identificar vários campos:

- Tipo de protocolo utilizado, *http*
- IP destino
- Número da porta
- Tipo de pedido, e o *path* do mesmo
- *Query* pedida

Consideramos ainda a utilização desta ferramenta devido à sua grande eficiência, e ainda o facto de apresentar um modo *non-gui*, que a permite aumentar ainda mais.

Mais à frente neste documento serão então apresentados os testes utilizando esta ferramenta e os resultados obtidos.

4.2 Limitações da aplicação *wiki.js*

Com a inicialização dos testes à aplicação *wiki.js*, apercebemo-nos que a mesma apresentava algumas restrições permanentes relativamente à realização de certos pedidos.

Isto limitou a grande maioria dos testes que o grupo tinha planeado, visto que alguns pedidos apresentam um *time out* após serem realizados um determinado número de vezes.

Por exemplo, a aplicação só permite cinco *logins* de 60 em 60 segundos vindos do mesmo local de acesso, pelo que não nos foi possível testar esta funcionalidade com diversas *threads*, pois o resultado obtido seria apenas 5 pedidos com sucesso e os restantes pedidos obteriam respostas de erro.

Outra limitação prende-se na inserção de comentários, que só permite a realização de um comentário de 15 em 15 segundos dificultando assim os testes a esta funcionalidade.

Estas foram apenas algumas limitações apresentadas relativamente à aplicação, para o nosso caso específico de estudo. Visto que estas podem nem ser limitações na sua essência mas sim mecanismos de segurança.

O grupo foi ainda, no decorrer da avaliação de desempenho, encontrando novas "limitações", mas pensamos não ser relevante o enumerar de todas elas, pois não é o objeto de estudo deste documento.

Com isto o grupo focou-se um pouco mais nos pedidos de *get* para os teste de carga, devido às limitações apresentadas. Para tal foi necessário criar os pedidos na linguagem de consulta *GraphQL*, que nos permite formatar os nossos pedidos. Não existiu grande dificuldade na utilização da mesma, visto que não tem uma grande curva de aprendizagem.

Assim sendo, o foco dos nossos testes foi principalmente testar a capacidade de resposta da aplicação à carga de muitos pedidos de certas páginas.

4.3 Request de uma página

Como foi explicado na secção anterior, o pedido de páginas foi o grande foco dos nossos testes, isto porque num cenário real esta será utilidade que mais vai ser usada pelos utilizadores da aplicação.

Assim sendo, definimos um conjunto padrão de testes para esta fase:

- Variar o número de *Web Servers* (*WSs*) de 1 até 3;
- Variar o número de pedidos simultâneos (*threads*)/clientes concorrentes, passando por testes de 100, 250, 500, 1000, 2000, 4000, 8000 e até 16000 *threads*.

Desta forma somos capazes de avaliar a diferença nos tempos de resposta, *throughput* e *abort-rate*, quando temos mais ou menos servidores a correr o nosso serviço *Node.js*.

Começamos por definir duas páginas na aplicação: uma com pouco conteúdo e outra com muito, um conceito que fomos aprimorando com os testes, para vermos as diferenças do desempenho também ao nível do tamanho de transferências de dados feitas desde a base de dados até ao utilizador final.

4.3.1 Página com pouco conteúdo

Começamos então os testes de carga com *requests* sucessivos de uma página com pouco conteúdo. O pedido constitui uma consulta à API em GraphQL com a seguinte estrutura:

```
{
    pages {
        single(id:3) { content }
    }
}
```

E passamos ao registo dos tempos de resposta para as diferentes configurações explícitas:

Tempo médio de reposta (ms)		*Nº de threads*							
		100	**250**	**500**	**1000**	**2000**	**4000**	**8000**	**16000**
Nº de WS	**1ws**	567	843	1064	2299	2652	11873	22489	53567
	2ws	233	672	957	1811	5176	4343	18320	21399
	3ws	307	563	436	1781	4702	11503	17751	32472

Tabela 1: Tempo médio de resposta para pedidos "Small Content".

A visualização destes dados torna-se complicada apenas com base na tabela apresentada, pelo que recorremos à sua representação em gráficos de linhas, dividindo-os em duas representações com escalas diferentes:

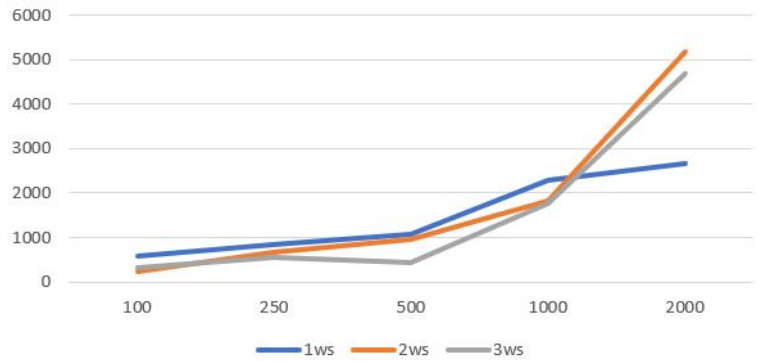

Figura 9: Variação de *threads* entre 100 e 2000.

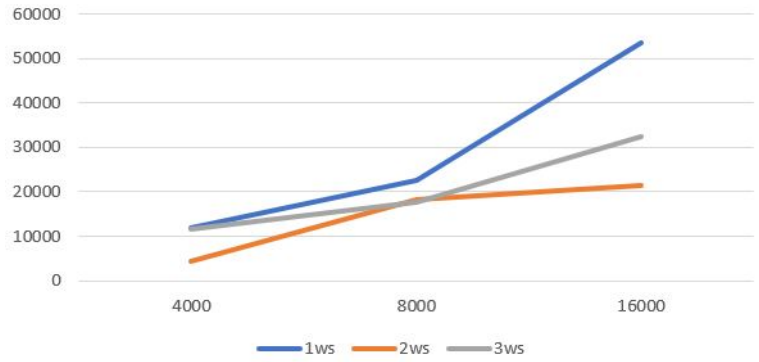

Figura 10: Variação de *threads* entre 4000 e 16000.

Com estes testes, esperávamos que a solução com 3 *Web Servers* fosse sempre melhor que todas as outras e que a de 2 *Web Servers* melhor que a de 1. É possível constatar que isso nem sempre acontece, isto é, alguns valores fogem ao normal, tal situação pode acontecer devido a alguma instabilidade da infraestrutura da *GCP* da qual não podemos controlar. Outro caso que explica esta variância é o facto de que a base de dados continua a ser apenas servida por uma máquina e, portanto, o aumento de *WSs* pode gerar uma maior incidência de pedidos na mesma, que tem como consequência um maior tempo de resposta total dando a ideia que por vezes o aumento destes pode ser prejudicial.

Apesar de tudo podemos ver pelos gráficos que no geral os resultados são como o esperado.

Um outro parâmetro de desempenho fornecido pelo *JMeter* é o *Abort Rate*, no qual se incluem, para as mesmas variações de *threads* e *WSs*, as percentagens de erro obtidas nas diferentes execuções:

Abort Rate (%)		*Nº de threads*							
		100	**250**	**500**	**1000**	**2000**	**4000**	**8000**	**16000**
Nº de WS	**1ws**	0	0	0	0	0	0	6,613	29,781
	2ws	0	0	0	0	0	0	6,463	7,337
	3ws	0	0	0	0	0	0	5,775	9,875

Tabela 2: Variação do *Abort Rate* para pouco conteúdo.

Deste modo, procedemos à visualização destes valores através de um gráfico de linhas:

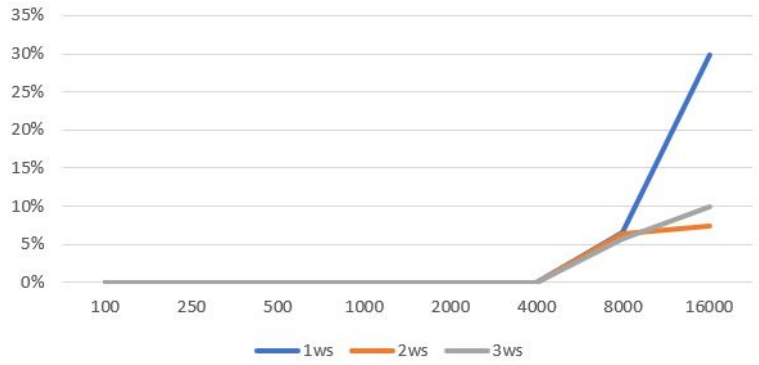

Figura 11: Variação do *Abort Rate* para pouco conteúdo.

É de esperar que o *Abort Rate* diminua com o aumento de *Web Servers* e podemos verificar isso mesmo, de forma clara, para um número de *threads* maior. Inicialmente, este parâmetro mantêm-se nulo, tendo efeitos significativos quando ultrapassámos as 4000 execuções simultâneas, sendo que outros fatores podem também estar envolvidos nos erros obtidos, como a instabilidade da *Google Cloud Platform*.

Notámos que para 8000 pedidos paralelos quantos mais servidores melhor desempenho, ou seja, menos erros, mas sendo as diferenças entre as linhas muito baixas não achamos que compense uma escolha entre uma ou outra configuração de servidores. Já para 16000 *threads* vemos uma melhoria significativa entre ter 2 servidores em relação a um e, portanto, verificasse claramente que o uso de apenas um *WS* não é muito viável, já 2 e 3 não apresentam diferenças tão significativas e, de seguida, para páginas com mais conteúdo a diferença será mais clara.

4.3.2 Página com muito conteúdo

Um segundo teste de carga envolve também pedidos de conteúdo, no entanto, com mais *bytes* por pedido, sendo neste caso o valor definido para `50k` *bytes* (caracteres) de texto, essencialmente, aleatório. O pedido constitui uma consulta à API em *GraphQL* com a seguinte estrutura:

```
{
    pages {
        single(id:2) { content }
    }
}
```

Após os vários testes feitos para este tipo de pedido, fazendo variar o número de *Web Servers* e o número de *threads*, obtivemos os seguintes resultados em termos de tempo de resposta médio, em milisegundos, para cada execução:

Tempo médio de reposta (ms)		*Nº de threads*							
		100	**250**	**500**	**1000**	**2000**	**4000**	**8000**	**16000**
Nº de WS	**1ws**	724	1093	1588	2200	5015	10953	19454	65661
	2ws	482	763	1180	1995	3536	10424	15460	56802
	3ws	456	617	603	1605	4604	7888	15165	18940

Tabela 3: Tempo médio de resposta para pedidos "Big Content".

A visualização destes dados torna-se complicada apenas com base na tabela apresentada, pelo que recorremos à sua representação em gráficos de linhas, dividindo-os em duas representações com escalas diferentes:

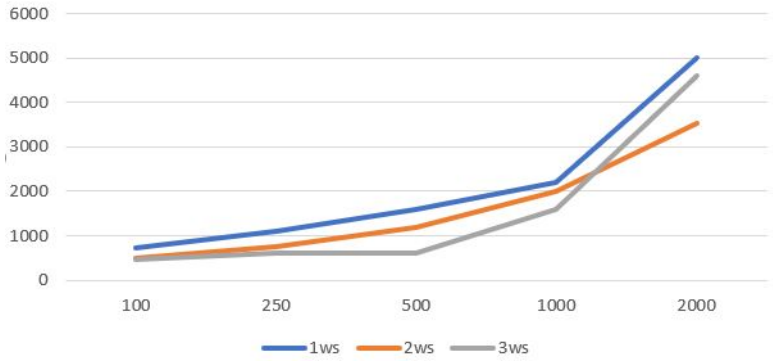

Figura 12: Variação de *threads* entre 100 e 2000.

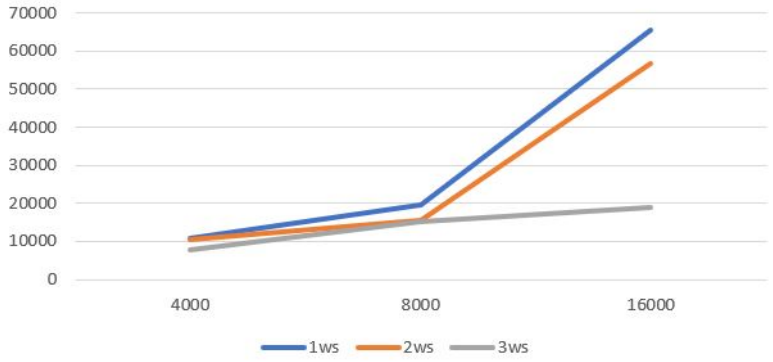

Figura 13: Variação de *threads* entre 4000 e 16000.

A interpretação de ambos os gráficos concentra-se no facto de que, como esperado, para uma definição de 3 servidores a balancear a carga temos um tempo de resposta, em geral, abaixo das demais configurações.

Esta observação permite-nos deduzir que, para muitos clientes em simultâneo, com mais conteúdo requisitado, tiramos mais partido de uma configuração com mais servidores, sendo esta a solução mais escalável e que nos providencia maior desempenho. Em particular, para 16000 *threads* as diferenças são significativas em relação a 1 e 2 servidores, por exemplo, obtemos menos *2 terços* do tempo, em média, observando uma diferença parecida comparando a carga para pouco conteúdo da constatada na secção anterior.

Assim como no caso anterior, procedemos à análise do *Abort Rate* para páginas com muito conteúdo:

Abort Rate (%)		*Nº de threads*							
		100	**250**	**500**	**1000**	**2000**	**4000**	**8000**	**16000**
Nº de WS	**1ws**	0	0	0	0	0	0,325	2,250	37,962
	2ws	0	0	0	0	0	0	4,163	32,362
	3ws	0	0	0	0	0	0	3,938	5,275

Tabela 4: Variação do *Abort Rate* para muito conteúdo.

O seguinte gráfico será bastante esclarecedor no que toca à relação entre número de servidores e a disponibilidade dos serviços:

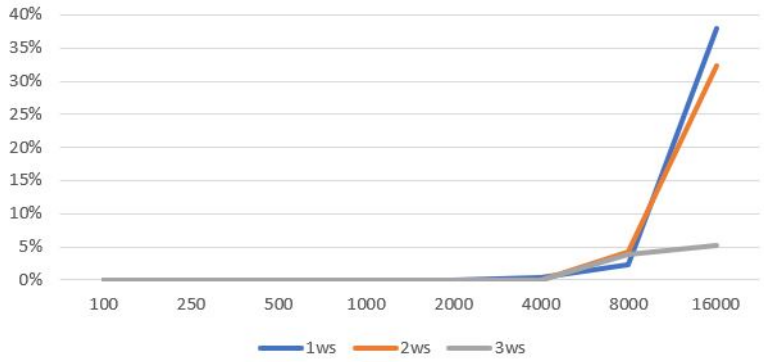

Figura 14: Variação do *Abort Rate* para muito conteúdo.

Desta vez, para execuções de pedidos simultâneos inferiores a 4000 temos, novamente, um registo nulo de *Abort Rate* o que é, à partida, um bom sinal inicial.

Fazendo uma análise para os valores subsequentes, temos uma diferença clara entre usar 3 servidores ou menos, sendo que para 8000 e 16000 fios de execução a percentagem de erro cai de 30%+ para apenas 5%, estabelecendo uma enorme vantagem visto que o serviço deve conseguir responder a todos os clientes mesmo que demore uns segundos a mais.

Analisando, de forma geral, os testes para pouco e muito conteúdo podemos perceber que se torna mais vantajoso utilizar 3 servidores pois compensámos a pouca diferença verificada em *Response Time* com um valor reduzido de *Abort Rate*.

4.4 Atualizações do perfil de utilizadores

Como uma alternativa a pedidos de *consulta* de conteúdo, operações essas que devem corresponder ao processo mais comum neste sistema aplicacional, decidimos introduzir algumas mutações disponibilizadas pelo *GraphQL*, no entanto, apenas nos foi possível apresentar um teste que envolve atualizar o perfil de um utilizador, segundo muitos pedidos paralelos. O pedido encontra-se descrito de seguida:

```
String user_name = "userNovo" + ${__threadNum};
vars.put("user_name", user_name);
```

```
mutation {
  users {
    update(id: 4, name:"${user_name}") {
      responseResult {
        succeeded
        errorCode
        slug
        message
      }
    }
  }
}
```

Na tabela seguinte, apresentamos a usual medição de *Response Time*:

Tempo médio de resposta (ms)		*Nº de threads*							
		100	**250**	**500**	**1000**	**2000**	**4000**	**8000**	**16000**
Nº de WS	**1ws**	152	128	117	116	118	1202	5381	11149
	2ws	132	118	117	216	132	363	3212	10813
	3ws	126	114	113	213	582	1808	4343	10716

Tabela 5: Tempo médio de resposta para pedidos "Update user".

O gráfico que representa a variação observada na tabela anterior pode ser observado de seguida:

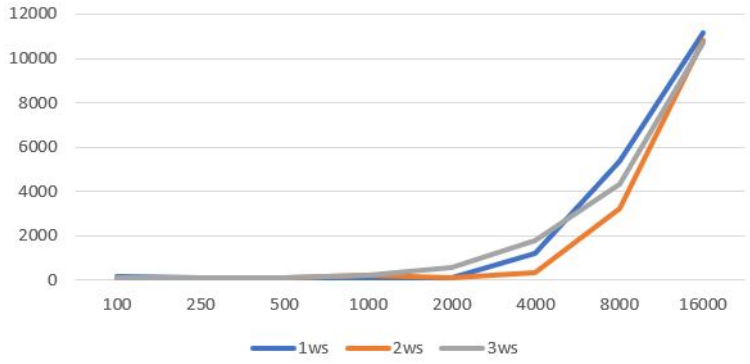

Figura 15: Média de *ResponseTime* entre 100 e 16000 threads

De forma clara, conseguimos perceber que a diferença de tempos é quase insignificante para 1 ou 3 servidores no nosso sistema. Já para 2 servidores temos uma melhoria mais notória no tempo de resposta e portanto parece-nos ser esta a melhor opção, apesar de não ser este o foco da aplicação.

Esta operação é, desta forma, um processo mais pesado que envolve não apenas ler os dados de uma base de dados como também modificá-los, no entanto, em pequena escala pois o conteúdo alterado é muito pouco.

Quanto à análise do *Abort Rate* não é necessário apresentar grande informação visto que os valores obtidos para o mesmo foram sempre nulos.

4.5 Outros testes realizados

No nosso plano de testes, para além dos casos apresentados até então, temos mais um conjunto de consultas e mutações que não tiveram o resultado esperado ou revelaram-se muito complicados de testar com base nas limitações existentes na *Wikijs*, destancando os seguintes:

1. Pedidos de *Log-in* em grande escala: Já referido na secção de limitações, representa um *use case* pouco prático visto que a própria aplicação limita este tráfego;
2. Criar comentários em publicações: Explicação semelhante ao caso anterior;
3. Criar novas páginas: A criação de publicações é também uma grande limitação na realização de testes deste tipo visto que, em média, para cada página, o tempo de resposta foi constante e de cerca de 20s / página.
4. Criar/Remover utilizadores: Para um mesmo padrão de testes como o anterior estabelecido, este teste limitava-se a apresentar resultados, em termos de tempo de resposta, mais razoáveis, para valores inferiores a 300 *threads*, e tempos com um crescimento exponencial para pedidos maiores;

4.6 Outros Resultados

Para além do tempo médio de resposta e do *Abort-Rate*, o *JMeter* permite-nos obter muitos outros resultados dos testes, onde iremos analisar alguns destes de seguida.

O primeiro é o tempo médio de resposta de 95% das últimas *threads* executadas, ou seja, é um resultado que ignora os resultados das primeiras 5% das *threads*.

Ao analisarmos estes valores, em todos os testes os resultados para 1, 2 e 3 *WSs* são muito parecidos e isto acontece pois a diferença faz-se quando só algumas *threads* estão a chegar, quando a maioria das *threads* já foi lançada, o grande local onde vai haver espera é na base de dados.

Isto acontece pois, tal como já foi explicada, esta continua a ser servida por apenas uma máquina, deste modo aumentar os *Web Servers* é benéfico enquanto o tráfego na base de dados tem um débito maior, pois quando esta deixa de conseguir despachar os pedidos, o número de *WS* acaba por não fazer diferença visto que a certa altura teríamos que melhorar a máquina da base de dados ou então usar uma sistema de armazenamento distribuído. Aumentar o número de *WS* infinitamente pode até ser prejudicial pois gera mais pedidos simultâneos ao servidor de base de dados.

Para clarificar as ideias apresentadas, temos de seguida a tabela relativa a esta métrica para o teste de *Get* de uma página com muito conteúdo:

TMR *95% threads (ms)*		*Nº de threads*							
		100	**250**	**500**	**1000**	**2000**	**4000**	**8000**	**16000**
Nº de WS	**1ws**	857	1405	2032	3784	15702	32125	65191	131166
	2ws	574	1202	3175	7382	7835	31902	64642	131050
	3ws	627	1198	1201	7386	15784	31640	65863	130268

Tabela 6: Tempo médio de resposta das últimas 95% threads para pedidos "Big Content".

4.7 Tolerância a Faltas

Para a verificação da tolerância a falhas que a nossa infraestrutura suporta, foi utilizado um valor fixo de dois *web servers*. O intuito destes testes passa por verificar como os serviços se mantêm ativos após desligar certas máquinas e também verificar como a carga é balanceada entre os *web servers*.

Com estes testes queremos garantir que definimos uma infraestrutura de alta disponibilidade e desempenho, mantendo sempre todos os serviços ativos.

Como tal realizamos dois tipos de testes que consistia em desligar a máquina "principal", onde estavam a correr os serviços, e observar que os serviços migravam para a máquina *backup*. Verificar ainda que o balanceador está a funcionar corretamente distribuindo o trabalho pelos *web servers* existentes.

4.7.1 Clientes - *iSCSI Cluster*

1.º Teste

Com este teste o grupo espera simular que, após ocorrência de uma falha na máquina *cli1* todos os serviços sejam migrados para a *cli2*, e que se consiga observar que os serviços continuam ativos, ou seja, os pedidos continuam a ser processados.

```
Test Log:

[18:05] Teste Iniciado
  ...   Pedidos são processados
[18:08] cli1 desligada
  ...   Interrupção no processamento de pedidos
[18:08] Migração dos serviços para cli2
  ...   Continuação da execução de pedidos
[18:11] Teste Finalizado
```

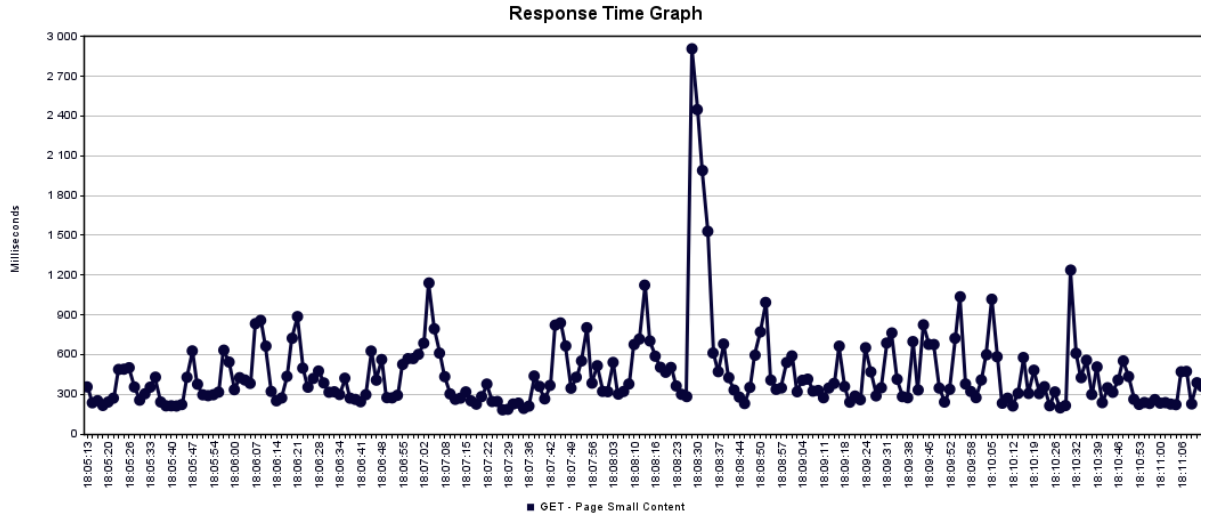

Figura 16: Evolução do *Response Time*

Com o gráfico apresentado anteriormente, relativamente ao *Response Time* dos pedidos efetuados, conseguimos tirar conclusões importantes. Conseguimos perceber que inicialmente o *Response Time* mantém-se, apesar de algumas pequenas oscilações, quase constante. O único pico observável, e que aumenta quase em 100 vezes o tempo de resposta, ocorre no instante de tempo 18:08:23, e que coincide com o desligar da máquina *cli1*. O intervalo de tempo que decorre entre o instante 18:08:23 e o instante 18:08:37 corresponde à migração dos serviços da máquina *cli1* para a máquina *cli2*.

Após este pico pode-se observar que os tempos de resposta voltam a estabilizar, ou seja, que apesar de ter ocorrido uma falha em *cli1* os pedidos continuaram a ser processados, agora na máquina *cli2*.

Os gráficos apresentados a seguir representam a utilização do CPU das máquinas *cli1* e *cli2*, e foram retirados da ferramenta de monitorização que a GCP disponibiliza para cada uma das máquinas. Através da observação destes gráficos conseguimos observar de forma eficaz quando ocorre a falha na primeira máquina.

Figura 17: Utilização do CPU na *cli1*

O primeiro gráfico permite-nos perceber que *cli1* começou a ter o seu processador mais ativo no instante de tempo 18:05, ou seja, começou a fazer processamento de pedidos nesse instante e que essa carga aumenta até atingir um nível mais estável de processamento. De seguida o declínio acentuado da utilização do CPU no instante de tempo 18:08 permite-nos perceber que ocorreu uma falha nesta máquina, que neste caso foi provocado por nós, e que após esse incidente esta máquina nunca mais voltou a estar ativa.

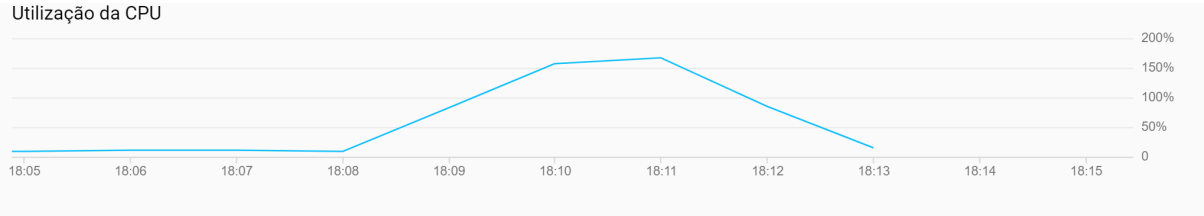

Figura 18: Utilização do CPU na *cli2*

Já o segundo é relativo à utilização do CPU na *cli2*, conseguimos perceber então que esta se encontra sempre ativa durante toda a realização deste teste e que apenas ocorre uma maior utilização do seu CPU no instante de tempo 18:08, ou seja, quando ocorre a falha na *cli1* e os serviços são migrados para esta máquina. É neste instante que começa o processamento dos pedidos, e de forma similar ao que ocorreu na *cli1*, esta estabiliza ao fim de algum tempo. Conseguimos perceber que no instante 18:13 foi quando terminou

o teste e que acabou o processamento de todos os pedidos, isto porque a utilização do CPU encontra-se apenas num nível residual.

Com a análise deste teste conseguimos perceber que os serviços são migrados de forma eficiente, existindo tolerância a falhas por parte dos nossos serviços *iSCSI* e que estes não serão um *SPOF - Single Point Of Failure* na nossa infraestrutura. De acrescentar que este serviço fica na mesma indisponível caso ambas as máquinas falhem.

2.º Teste

Similar ao teste efetuado anteriormente, o objetivo deste consite em observar que após a migração dos serviços da *cli1* para *cli2*, devido à falha em *cli1*, quando a máquina principal (cli1) é reinicida ocorre novamente uma migração dos serviços, desta feita de *cli2* para *cli1*.

```
Test Log :
[19:41] Teste Iniciado
  ...   Pedidos são processados
[19:44] Cli1 desligada
  ...   Interrupção no processamento de pedidos
[19:44] Migração dos serviços para cli2
  ...   Continuação da execução de pedidos
[19:47] Reinicio da cli1
  ...   Interrupção no processamento de pedidos
[19:47] Migração dos serviços para cli1
  ...   Continuação da execução de pedidos
[19:55] Teste Finalizado
```

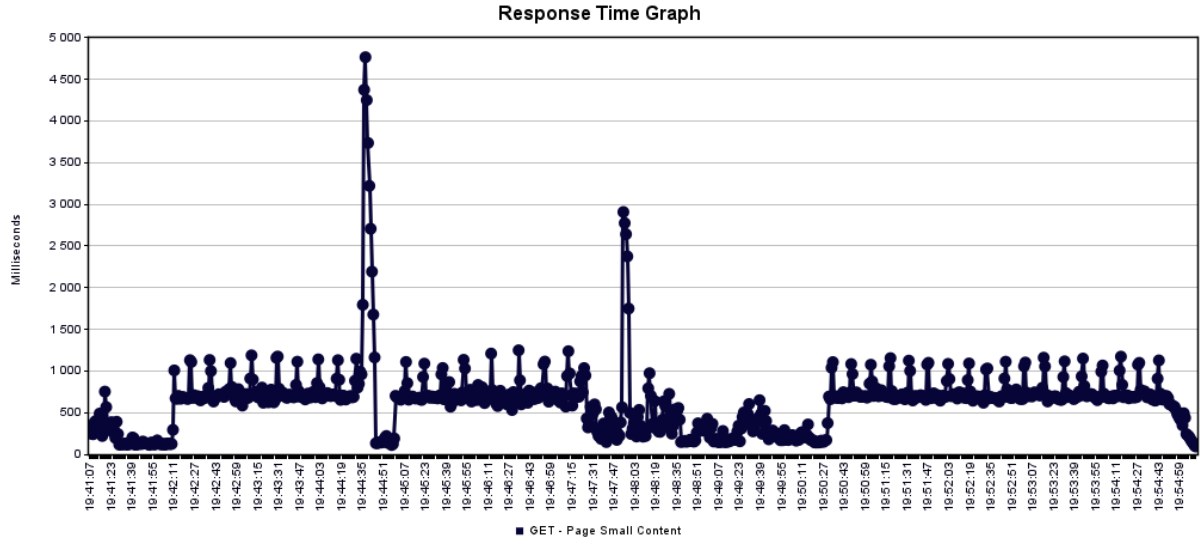

Figura 19: Tempo de Resposta durante o teste

Após a observação deste gráfico relativamente ao tempo de resposta dos pedidos efetuados, tal como para o teste anterior conseguimos obter bons resultados. Conseguimos neste caso verificar então que existem dois picos no gráfico, relativamente às duas grandes ocorrências existentes neste teste.

O primeiro deve-se ao facto de se desligar a *cli1* no instante de tempo 19:44:30, pois até este instante os tempos de resposta eram uniformes. Percebemos que a migração dos serviços de *cli1* para *cli2* termina no instante 19:45:07 pois a partir deste instante os tempos de resposta encontram-se novamente uniformes. Depois de um período de tempo de respostas uniformes ocorre novamente um pico desta feita devido à reinicialização do *cli1*, ou seja, a segunda grande ocorrência neste teste, no instante 19:47:47. Percebemos que esta nova migração termina no instante de tempo 19:48:03, novamente porque os tempos de resposta voltam a estabilizar após este instante. Verificamos que o teste se encontra finalizado no instante de tempo 19:55:00 pois todos os pedidos já se encontram efetuados.

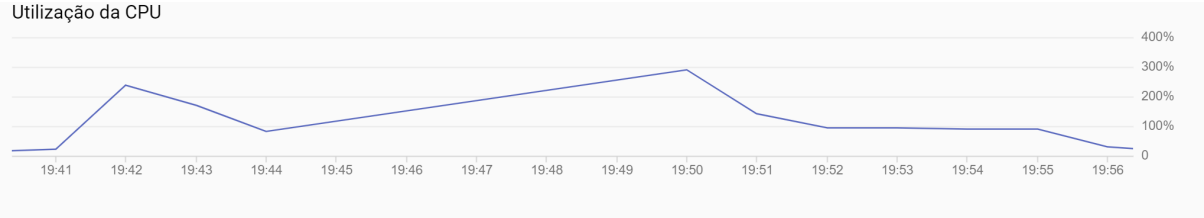

Figura 20: Utilização do CPU no cli1

Verificamos o ínicio do processamento dos pedidos no instante 19:41 pois é no qual a utilização do seu CPU começa a aumentar até estabilizar. Não se trata de um gráfico tão conclusivo como o anterior visto que os valores do processamento nunca chegam a atingir valores, muito próximos de zero. Percebemos então que estes gráficos da utilização do CPU nem sempre fornecem informações relevantes sobre o estado da máquina, visto que a máquina pode estar a ser utilizada não para processamento de pedidos dos clientes, mas sim para processamento interno da máquina.

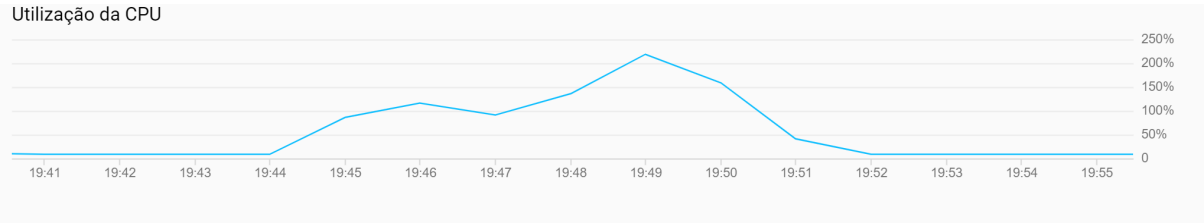

Figura 21: Utilização do CPU no cli2

Já o gráfico de utilização do CPU da máquina *cli2* é-nos mais conclusivo, podendo agora observar quando esta está a servir pedidos de clientes ou quando não está. São verificadas estas duas grandes ocorrências devido ao aumento do processamento da máquina a partir do instante 19:44, devido à migração dos serviços de *cli1* para *cli2*, e devido à diminuição da utilização do CPU apartir do instante 19:49, que nos leva a entender que os serviços foram novamente migrados para a máquina *cli1*.

Com este teste, conseguimos obter os resultados esperados ou seja a existência de duas migrações dos serviços entre as máquinas *cli1* e *cli2*. A primeira migração devido à ocorrência de uma falha em *cli1* e a segunda devido à reinicialização da *cli1*. Feito isto podemos concluir que a nossa infraestrutura é capaz de reagir com grande eficiência

às falhas com ocorrência nos serviços *iSCSI* e que estes não são um entrave nem à performance nem à disponibilidade do nosso sistema.

4.7.2 Diretores

Tal como no teste anterior, agora vamos simular uma falha na máquina diretor com o objetivo de demonstrar que os serviços continuam funcionais.

```
Test Log :
[17:02] Teste Iniciado
  ...   Pedidos são processados
[17:06] Diretor desligado
  ...   Interrupção no processamento de pedidos
[17:06] Migração dos serviços para diretor backup
  ...   Continuação da execução de pedidos
[17:11] Teste Finalizado
```

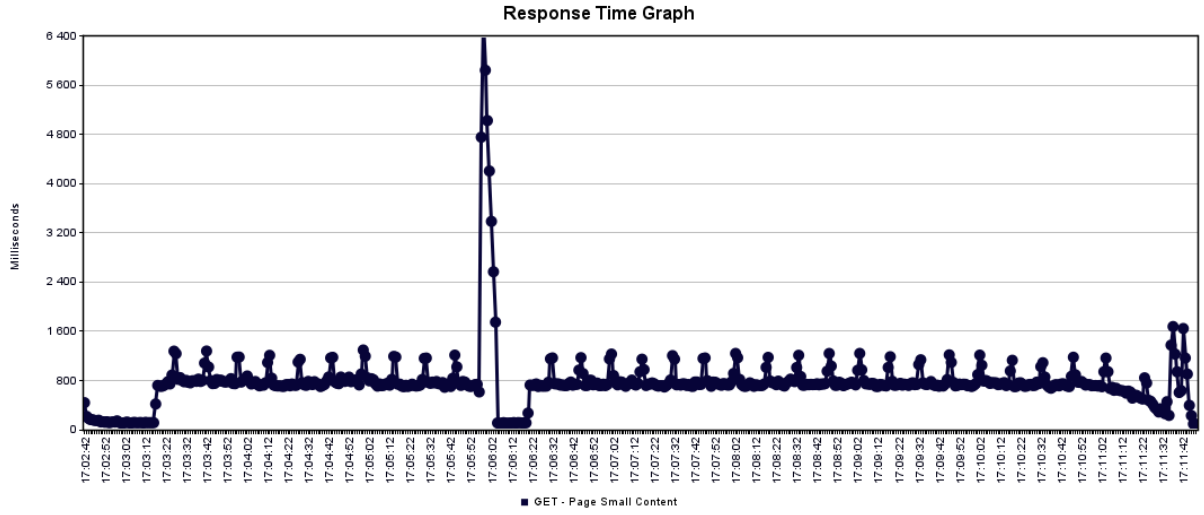

Figura 22: Evolução do *response-time*.

Com o gráfico apresentado acima conseguimos perceber a evolução do *response-time* ao longo do teste efetuado. É possível perceber que no instante de tempo 17:05:52, o tempo de resposta sofre um aumento considerável, correspondente à falha na máquina diretor. Passado alguns segundos os serviços voltam a estar completamente funcionais, e o *response-time* volta a apresentar valores constantes ao longo do tempo, até o teste terminar.

É possível perceber, pela análise do gráfico que, apesar da falha, correspondente ao desligar do diretor, a máquina diretor *backup* assume os pedidos e permite manter um constante fluxo de pedidos.

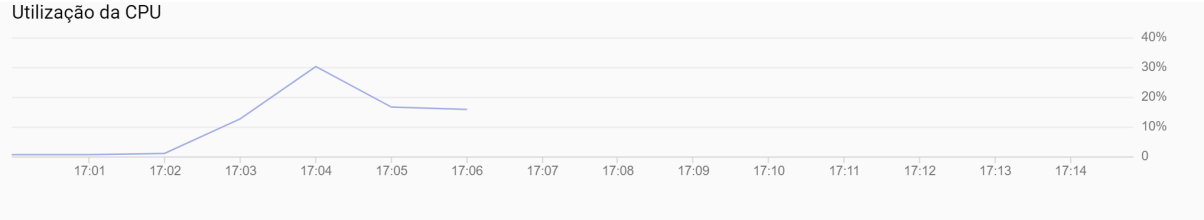

Figura 23: Utilização do CPU no diretor

No gráfico acima conseguimos ver que a máquina diretor faz o processamento dos pedidos até ao instante 17:06, em que é desligada.

Figura 24: Utilização do CPU no diretor *backup*

Já na máquina diretor *backup* podemos ver que se encontrou sempre ligada, e a partir do instante 17:06 a utilização de CPU aumenta, significando que nesse momento, esta assumiu os pedidos e passou a realizar o balanceamento dos pedidos pelos *web servers*.

Nos gráficos apresentados abaixo conseguimos perceber que durante o todo o teste, o balanceamento de pedidos pelos 2 *web servers* se mantém idêntico em ambos, mostrando que apesar da falha os serviços continuam funcionais e disponíveis.

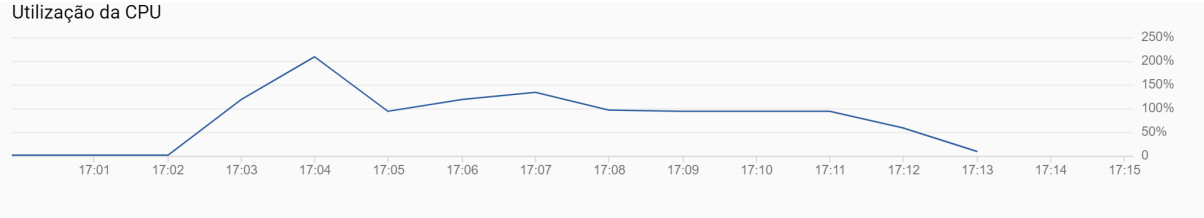

Figura 25: Utilização do CPU no ws1

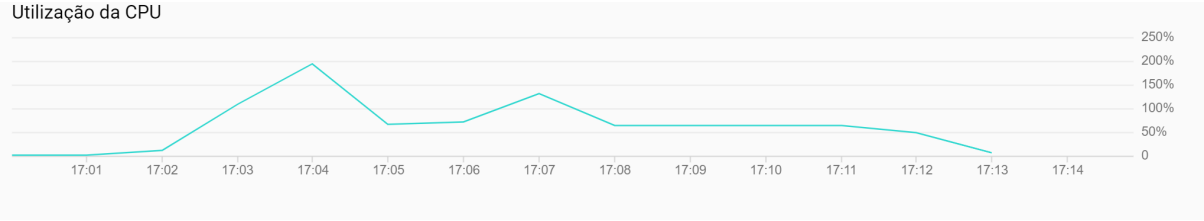

Figura 26: Utilização do CPU no ws2

4.7.3 *drbd*

Para finalizar os testes de tolerância a faltas iremos agora, usando o mesmo método dos testes anteriores, avaliar como o sistema se comporta quando existe uma falha num dos *drbd*. Para a simulação decidimos desligar a máquina *drbd2* e, posteriormente, voltar a liga-la .

```
Test Log :
[21:59] Teste Iniciado
  ...   Pedidos são processados
[22:01] drbd2 é desligado
  ...   Pedidos continuam a ser processados
[22:10] drbd2 é ligado
  ...   Continuação da execução de pedidos
[22:12] Teste Finalizado
```

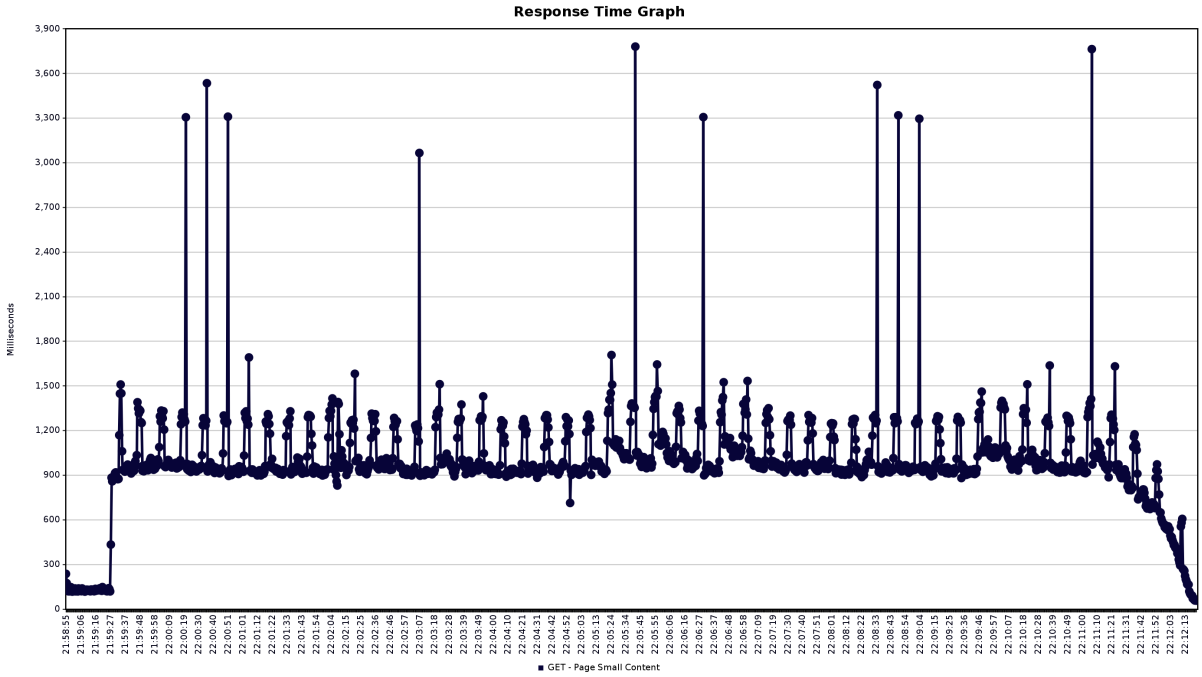

Figura 27: Evolução do *Response-time* durante a falha e recuperação de um *drbd*.

Analisando o gráfico acima podemos ver que o tempo de resposta segue um padrão mais ou menos constante, existindo alguns picos esporádicos que não se relacionam com o teste em si. Conseguimos perceber que uma falha numa das *drbd* não terá interferência no desempenho global da aplicação. Estes resultados encontram-se de acordo com o propósito da criação das mesmas, já que têm como principal função a replicação dos dados, de modo a garantir a persistência dos mesmos. Tal podemos confirmar com o gráfico acima, já que não existe indisponibilidade do serviço, havendo sempre os dados necessários na base de dados.

5 Conclusão

Alta disponibilidade e escalabilidade são dois pontos chaves numa aplicação e, para alcançá-los, devemos recorrer a um bom planeamento de toda a estrutura da aplicação de modo a garantir um bom funcionamento para os utilizadores.

Neste projeto, em ambiente *Cloud*, conseguimos analisar todas as diferentes componentes de uma aplicação, de modo a perceber as suas limitações, e saber distribuir a sua configuração numa infraestrutura que suporta aplicações, em grande escala, de modo a que o utilizador encontre uma aplicação sempre disponível e com alto desempenho.

Se pensarmos em termos práticos, uma indisponibilidade, por mais pequena que seja, em grande parte dos casos pode levar a grandes perdas e custos na manutenção dos recursos, tanto a nível monetário como a nível de reputação e tempo de resolução dos problemas. A importância desta análise pode não ser visível numa fase inicial do projeto, mas com o decorrer do tempo e a consequente necessidade de uma alteração da configuração da infraestrutura, do *software* ou do número de máquinas existentes, isto torna-se muito relevante de ter como pré-requisito.

A principal dificuldade, num momento inicial, concentrou-se em conseguir implementar o *Cluster* de alta disponibilidade no *Backend*, através de servidores *iSCSI* (dois clientes), visto que a sua configuração não seguiu mais o padrão que utilizava um recurso para um IP virtual (*ipaddr2*), no entanto, a sua resolução foi simplificada visto que recorremos a mecanismos, já explicados, de *failover* na *GCP*.

Em segundo lugar, destacamos as próprias limitações da *Wiki.js* que tiveram impacto na diversidade dos testes relacionados com inserção/atualização (mutações) de dados, não sendo, a nosso ver, o ponto mais importante num sistema aplicacional como este, cujo tráfego mais importante e mais frequente não se concentra em inserção de utilizadores ou páginas, mas sim a visualização das mesmas, que podem, deste modo, ter bastante conteúdo.

Em terceiro lugar, achamos importante referir que, como não fomos nós a desenvolver a aplicação, não temos a possibilidade de separar os componentes da mesma *frontend* e *backend* pois, por um lado, pode não ser efetivamente possível, e por outro pode não ser seguro pois não sabemos como é que são geridos os pedidos ao *backend* e, portanto, não nos foi possível testar, em separado, por mais que achemos que seria importante em qualquer aplicação com esta estrutura.

Por fim, balanceando todos os prós e contras detetados ao longo do desenvolvimento deste trabalho, concluímos que foi bastante positiva toda esta análise, em grupo, que nos preparou para ambientes futuros de trabalho de equipa onde aprendemos a ter espírito crítico em todas as questões que relatamos anteriormente, importantes para pensar em termos da escalabilidade global de uma aplicação.

6 Scripts

6.1 Código Selenium

```
public class Login{

    private String email;
    private String password;

    public Login(String email, String password) {
        this.email = email;
        this.password = password;
    }

    public void executeLogin(WebDriver browser, String ipadd){
        //Access login page wikijs
        browser.get("http://" + ipadd + "/login");
        //Find input email
        WebElement input_email = browser.findElement(By.id("input-20"));
        input_email.sendKeys(this.email);
        //Find input password
        WebElement input_password = browser.findElement(By.id("input-22"));
        input_password.sendKeys(this.password);
        //Enter LogIn
        input_password.sendKeys(Keys.ENTER);

        WebDriverWait wait = new WebDriverWait(browser, 5);
        wait.until(ExpectedConditions.elementToBeClickable(By.id("discussion")));
    }
}

public class Test extends Thread{

    public WebDriver browser;
    public String email;
    public String pass;
    public String ipaddress;

    public Test (WebDriver browser){
        this.browser = browser;
        this.email = "admingrupo10wikijs@icd.pt";
        this.pass = "12345grupo10";
        this.ipaddress = "35.228.23.28:3000";
    }

    public void run(){
        //double real_time = login();
        //double duration = addComment("ISTO  UM TESTE") + real_time;
        double duration = reloadPage("");
        System.out.println(" Interao  durou " + duration + " s");
```

```
    }

    public double login(){
        long initial = System.currentTimeMillis();
        Login log = new Login(this.email, this.pass);
        log.executeLogin(this.browser, this.ipaddress);
        long fim = System.currentTimeMillis();
        return (fim - initial)/1000.00 ;

    }

    public double addComment(String comment){
        long initial = System.currentTimeMillis();
        browser.findElement(By.id("discussion-new")).sendKeys(comment);
        browser.findElement(By.cssSelector("button[type='button'][aria-label='Post
            Comment']")).click();

        //Esperar que o meu comentrio seja inserido
        WebDriverWait wait = new WebDriverWait(browser, 5);
        wait.until(ExpectedConditions.elementToBeClickable(
        By.xpath("//*[contains(text(),'a few seconds ago')]")));

        browser.quit();
        long fim = System.currentTimeMillis();
        return (fim - initial)/1000.00;
    }

    public double reloadPage(String path){
        long initial = System.currentTimeMillis();
        browser.get("http://" + this.ipaddress + path);
        WebDriverWait wait = new WebDriverWait(browser, 5);
        wait.until(ExpectedConditions.elementToBeClickable(
        By.cssSelector("div[style='background-image:
        url(\"https://static.requarks.io/logo/wikijs-butterfly.svg\");
            background-position: center center;']")));

        long fim = System.currentTimeMillis();
        return (fim - initial)/1000.00 ;
    }

    public static void main(String[] args){
        System.setProperty("webdriver.chrome.driver",
            "C:/Users/Admin/Downloads/chromedriver.exe");
        //Create a chromeDriver
        WebDriver webD = new ChromeDriver();
        Test t = new Test(webD);
        t.start();
    }
}
```

6.2 Scripts inicialização

6.2.1 drbds

```
#!/bin/sh
sudo drbdadm up d1
sudo drbdadm --force primary d1
sudo systemctl restart target
```

6.2.2 Clientes -*iSCSI*

```
#!/bin/sh
sudo iscsiadm --mode node --logoutall=all
sudo iscsiadm --mode node -l
```