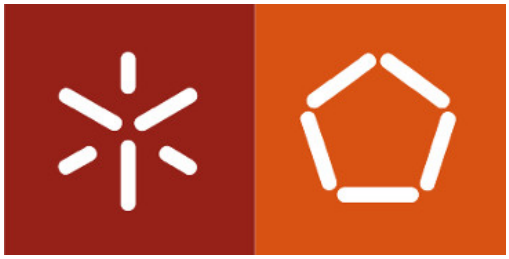

Universidade do Minho

Mestrado integrado em Engenharia Informática

Trabalho Prático

Grupo 5

Diogo Pinto Ribeiro, A84442

João Nuno Cardoso Gonçalves de Abreu, A84802

José Diogo Xavier Monteiro, A83638

Vasco António Lopes Ramos, PG42852

Infraestrutura de Centro de Dados
4º Ano, 1º Semestre
Departmento de Informática

28 de abril de 2021

Índice

1 Introdução **1**

2 Arquitetura e Componentes do Wiki.js **2**
2.1 Servidor Wiki.js 2
2.2 Servidor PostgreSQL 3

3 Arquitetura Inicial **4**
3.1 Pontos Críticos do Sistema 4
3.2 Testes de Carga 5
3.2.1 Teste 1 5
3.2.2 Teste 2 7
3.2.3 Teste 3 8
3.2.4 Teste 4 10
3.3 Análise de Desempenho 12

4 Arquitetura Implementada **14**
4.1 Camada de Persistência 14
4.2 Camada Aplicacional 15
4.3 Camada Web (*Load Balancers*) 16
4.4 Representação Geral da Arquitetura 18
4.5 Testes de Carga 19
4.5.1 Teste 1 19
4.5.2 Teste 2 20
4.5.3 Teste 3 22
4.5.4 Teste 4 24
4.6 Análise de Desempenho 27
4.7 Mitigação dos Pontos Críticos do Sistema 27

5 Conclusões **29**

A Wiki.js - configuração simples com Docker Compose **30**

Lista de Figuras

1 Diagrama da arquitetura do sistema 2
2 Percentil Tempo de Resposta para 100 e 1000 Threads 6
3 Percentil Tempo de Resposta para 5000 e 10000 Threads 7
4 Percentil Tempo de Resposta para 1000 e 5000 Threads 8
5 Percentil Tempo de Resposta para 100 e 250 Threads 10
6 Percentil Tempo de Resposta para 500 e 1000 Threads 10
7 Percentil Tempo de Resposta para 5000 Threads 10
8 Percentil Tempo de Resposta para 100 Threads 12
9 Percentil Tempo de Resposta para 250 Threads 12
10 Percentil Tempo de Resposta para 500 Threads 13
11 Arquitetura Implementada 18
12 Percentil Tempo de Resposta para 100 e 1000 Threads 20
13 Percentil Tempo de Resposta para 5000 e 10000 Threads 20
14 Percentil Tempo de Resposta para 1000 e 5000 Threads 21
15 Percentil Tempo de Resposta para 100 e 250 Threads 23
16 Percentil Tempo de Resposta para 500 e 1000 Threads 23
17 Percentil Tempo de Resposta para 5000 Threads 24
18 Percentil Tempo de Resposta para 100 Threads 26
19 Percentil Tempo de Resposta para 250 Threads 26
20 Percentil Tempo de Resposta para 500 Threads 26

Lista de Tabelas

1 Instalação Simples - Sumário do Teste 1 6
2 Instalação Simples - Sumário do Teste 2 7
3 Instalação Simples - Sumário do Teste 3 9
4 Instalação Simples - Sumário do Teste 4 11
5 Deployment - Sumário do Teste 1 19
6 Deployment - Sumário do Teste 2 21
7 Deployment - Sumário do Teste 3 22
8 Deployment - Sumário do Teste 4 25

1 Introdução

Cada vez mais o ser humano encontra-se conectado e dependente da tecnologia, sendo que esta tem de ser capaz de se adaptar e responder com sucesso às necessidades exigentes de qualquer tipo de cliente. Para isso, possuir uma infraestrutura segura e confiável capaz de garantir o fluxo de informação sem erros e em tempo útil é essencial para um serviço de qualidade. Sendo assim, é fundamental que as infraestruturas estejam preparadas para responder a desafios relativos à tolerância a falhas, escalabilidade, alocação de recursos lidando com grandes volumes de dados, elevada disponibilidade, eficiência energética, entre outros.

Nesse sentido, este trabalho tem como objectivo não só consolidar os conhecimentos obtidos na unidade curricular Infraestruturas de Centro de Dados, nomeadamente no planeamento, configuração, análise de desempenho e operação de infraestruturas de elevada disponibilidade e desempenho, como também implementar um serviço escalável e de elevada disponibilidade de infraestruturas computacionais para a plataforma *Wiki.js*.

Ao longo dos próximos capítulos será apresentada a abordagem feita pelo grupo para cumprir com os requisitos acima descritos.

2 Arquitetura e Componentes do Wiki.js

O projeto, que nos foi apresentado, consiste no planeio e operacionalização da instalação de uma plataforma **Wiki.js**, tal como toda a manutenção necessária ao seu bom funcionamento. Para esse efeito, começamos por desenvolver a arquitetura que servirá como base de todo o trabalho.

A aplicação **Wiki.js** encontra-se dividida entre *frontend*, que apresentará a interface ao *web client*, e *backend* e utilizará uma base de dados para o armazenamento de dados:

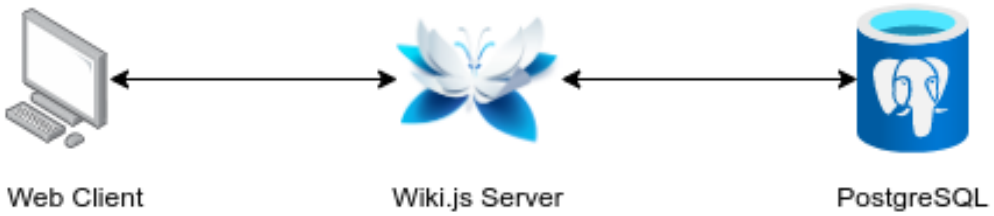

Figura 1: Diagrama da arquitetura do sistema

2.1 Servidor Wiki.js

Wiki.js é uma aplicação *open source*, escrita em JavaScript e que corre em Node.js, oferecendo a funcionalidade de gerar páginas wiki completamente customizáveis e modulares.

Para além de ser uma aplicação web extremamente rápida e ter um design simplista e elegante, possui um vasto conjunto de ferramentas *dev friendly* e oferece suporte para a base de dados **PostgreSQL** [1].

Numa primeira intalação (simplificada), esta plataforma, ficará instalada num servidor individual (servidor aplicacional). Aquando da comunicação pelos utilizadores, *web clients*, terá de fornecer o *frontend* ou camada de apresentação. Para além disso, o *backend* irá comunicar com o servidor de base de dados, onde será armazenada toda a informação.

2.2 Servidor PostgreSQL

Tal como referido previamente, o nosso servidor aplicacional comunica (escrita e leitura de dados) diretamente com uma base de dados **PostgreSQL**, que estará instalada no seu próprio servidor.

Decidimos recorrer a este sistema de gestão de base de dados por vários motivos. Primeiramente, a própria documentação da plataforma **Wiki.js** recomenda o seu uso, [2]. Depois, é o sistema mais popular do mercado e ganhou a sua extensa reputação devido à sua arquitetura, confiabilidade, integridade de dados, vasta gama de *features* e dedicação à comunidade *open source*. Além disso, está presente em todos os maiores sistemas operativos [3]. Por fim, achamos que o facto de já estarmos acostumados à sua utilização, se mostraria uma mais valia na realização deste projeto.

3 Arquitetura Inicial

De forma a perceber-se qual o comportamento do sistema como um todo, sem qualquer tipo de replicação ou otimização, decidiu-se fazer uma primeira instalação distribuída básica, utilizando *Docker* e *Docker Compose*, tendo por a configuração disponível na documentação do *Wiki.js*, em [4].

O objetivo desta instalação é conseguir avaliar o desempenho do sistema através da execução de um *workload* com múltiplos testes de carga e, com base nos respetivos resultados, identificar e explorar os potenciais pontos críticos onde o sistema pode falhar ou perder desempenho.

Para tal, usou-se uma configuração de *Docker Compose* com dois serviços (***db*** e ***wiki***), tal como se pode ver no anexo A.

3.1 Pontos Críticos do Sistema

Para que seja possível garantir que um serviço possua uma elevada disponibilidade mantendo-se com um bom desempenho no atendimento de pedidos, torna-se necessário perceber quais são os **pontos críticos de falha e desempenho** do nosso sistema.

O principal cenário em que um sistema pode falhar é aquele em que um dos componentes que o constitui (ou mais) falhar, não existindo mais do que uma instância de cada um destes elementos. Isto pode acontecer quer por falhas de software/hardware, quer por falhas de energia. No nosso caso em concreto, com a nossa arquitetura inicial, possuímos 2 elementos: a **Base de Dados** e o servidor **Wiki.js**. Estes elementos não possuem qualquer réplica, o que significa que na eventualidade de um ou mais falhar, todo o sistema irá ficar inoperacional.

Outro dos principais pontos de falha e desempenho de uma aplicação são os ***bottlenecks***. Os *bottlenecks* são situações em que um componente da aplicação limita todas as outras, diminuindo a performance do sistema. No nosso caso, o principal *bottleneck* que identificámos é a **Base de Dados** visto que tanto serve pedidos do *frontend* como do *backend*. Dado que a nossa base de dados não está preparada com mecanismos de alta disponibilidade, prevemos que esta poderá a

vir ser um ponto crítico do sistema.

Com estes pontos, ficámos com uma melhor noção do que poderá vir a ser um problema no nosso sistema, e quais serão algumas das soluções possíveis.

3.2 Testes de Carga

De modo a avaliar o desempenho da nossa instalação recorremos a **Testes de Carga** utilizando o **JMeter**. Utilizando GUI do *JMeter* criámos 4 Testes que tentam simular um pouco do que será o funcionamento de uma *Wiki* de um determinado produto ou serviço. Considerámos que no nosso cenário de teste utilizadores normais apenas consultam páginas enquanto que administradores têm a possibilidade de efetuar login e adicionar, editar ou remover páginas. Efetuámos essa segregação devido à diferença nos números de utilizadores e administradores, sendo os primeiros em número consideravelmente superior.

Os testes foram corridos no ambiente de linha de comandos de modo a poupar recursos das nossas máquinas, sendo que as limitações de hardware e as limitações relativas à **JVM** foram os principais obstáculos ao aumento da carga.

3.2.1 Teste 1

O Teste 1 visa representar o cenário mais básico: um utilizador aceder à página inicial da wiki. Para este teste irão ser utilizadas mais *threads* do que nos restantes dado que num cenário real, o acesso à página inicial da wiki é o mais comum. Ao correr este teste é testado o seguinte método:

1. GET /

O teste foi corrido com 100, 1000, 5000 e 10000 *threads* (utilizadores), sendo que recorremos aos relatórios gerados pelo *JMeter* para apresentar os seguintes resultados.

Página Inicial			
Threads	Erro	Tempo Resposta Médio (ms)	*Throughput (Trans./s)*
100	0.00%	4129.90	17.51
1000	0.00%	11572.32	47.10
5000	47.20%	23539.35	98.41
10000	67.87%	24610.09	157.72

Tabela 1: Instalação Simples - Sumário do Teste 1

Analisando os dados da tabela, podemos verificar que com 100 e com 1000 utilizadores não obtemos nenhuma percentagem de erro. Ao subir para um valor na casa dos 5000 já começámos a obter uma percentagem significativa de erros, sendo que com 10000 essa percentagem é ainda superior. Esta percentagem de erros permite-nos ter uma perceção do limite da disponibilidade da instalação atual da aplicação.

Ao aumentar o número de threads, verificámos também o aumento do *Throughput* da aplicação, bem como do tempo de resposta médio aos pedidos. Este último valor é aquele que nos preocupa mais acerca do teste dado que também pode ser encarado como um indicador de performance do sistema. Mesmo em casos em que todos os pedidos são atendidos, o tempo de resposta é bastante elevado, como por exemplo, no caso de 1000 *threads*, em que precisámos em média de 11.6 segundos para abrir a página inicial, algo que é inaceitável nos padrões atuais. Os tempos de resposta elevados em conjunto com a elevada taxa de erro significam que o nosso sistema não está a responder corretamente aos pedidos efetuados.

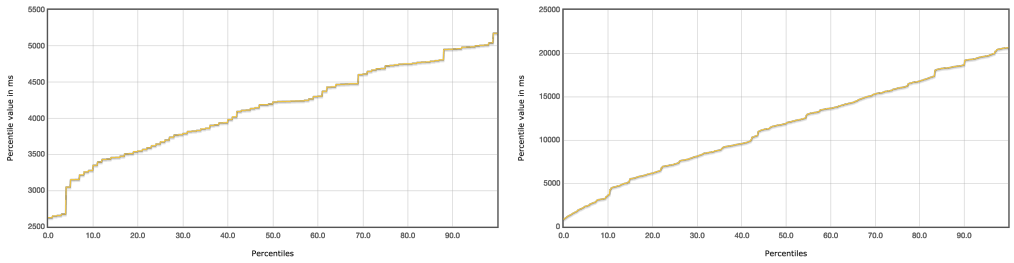

Figura 2: Percentil Tempo de Resposta para 100 e 1000 Threads

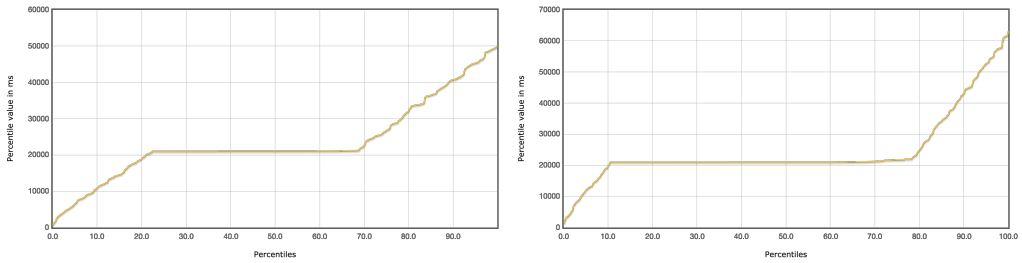

Figura 3: Percentil Tempo de Resposta para 5000 e 10000 Threads

Verificando os gráficos apresentados, fica ainda mais evidente que os tempos de resposta da aplicação são bastante elevados.

3.2.2 Teste 2

O Teste 2 visa representar o cenário em que um utilizador entra na página inicial, navegando de seguida para uma outra página da wiki. Neste caso em específico, todas as *threads* abrem a mesma página após pedirem a página inicial e esperarem 1000 milissegundos. Ao correr este teste são testados os seguintes métodos:

1. GET /
2. GET /en/ICD

O teste foi corrido com 1000 e 5000 *threads*, sendo que recorremos aos relatórios gerados pelo *JMeter* para apresentar os seguintes resultados. Estes valores foram escolhidos pela sua relevância tendo em conta o teste anterior.

Página Inicial			
Threads	Erro	Tempo Resposta Médio (ms)	*Throughput (Trans./s)*
1000	0.00%	9328.98	52.63
5000	49.56%	25202.86	109.55

Tabela 2: Instalação Simples - Sumário do Teste 2

Analisando os dados da tabela, podemos verificar que com 1000 utilizadores não obtemos nenhuma percentagem de erro. Quando aumentamos para valores na casa dos 5000 começámos a obter uma percentagem significativa de erros, que neste

caso já se demonstra inaceitável para uma plataforma funcional. Esta percentagem de erros mostra-nos um pouco qual o limite da disponibilidade da aplicação de momento.

Ao aumentar o número de threads, verificámos também o aumento do *Throughput* da aplicação, bem como do tempo de resposta médio aos pedidos. Este último valor, dado que também serve de medida de performance do sistema, mostra-nos que nos casos em que assegurámos que todos os pedidos são atendidos, estes esperam uma quantidade de tempo considerável.

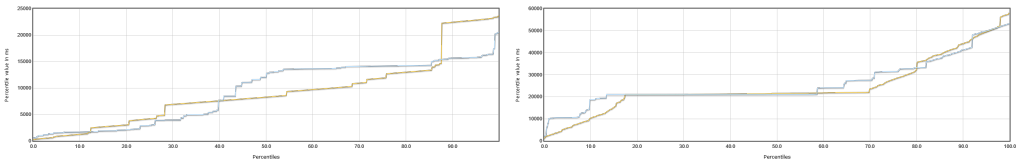

Figura 4: Percentil Tempo de Resposta para 1000 e 5000 Threads

Verificando os gráficos apresentados, em que a linha azul representa os pedidos referentes à página de teste e a linha laranja os pedidos referentes à página inicial, fica ainda mais evidente que os tempos de resposta da aplicação são bastante elevados.

3.2.3 Teste 3

O Teste 3 visa representar o cenário em que um administrador efetua login no sistema, sendo redirecionado para a página principal. Ao correr este teste são testados os seguintes métodos:

1. GET /login
2. POST /graphql
3. GET /

O teste foi corrido com 100, 250, 500, 1000 e 5000 *threads*, sendo que recorremos aos relatórios gerados pelo JMeter para apresentar os seguintes resultados. O processo de login é constituido pelos 3 pedidos apresentados, sendo que o conteúdo do POST que é feito possui o endereço de email e a password da conta utilizada

no teste. Estes 3 pedidos foram encapsulados num processo mais generalizado que representa o login como um todo. Optámos por esta visão de modo a representar de modo mais fidedigno o ato de efetuar login, dado que não é possível efetuar login se uma das operações falhar. Outra atenção que tivemos, foi o cancelamento de uma *thread* no caso de um dos pedidos falhar.

Página Inicial			
Threads	Erro	Tempo Resposta Médio (ms)	*Throughput (Trans./s)*
100	0.00%	6205.00	14.13
250	0.00%	7104.55	28.19
500	0.00%	16982.52	25.65
1000	1.09%	30312.79	22.80
5000	64.74%	90830.34	25.57

Tabela 3: Instalação Simples - Sumário do Teste 3

Analisando os dados da tabela, podemos verificar que com 100, 250 e 500 não obtemos nenhuma percentagem de erro. Ao subir para valor na casa dos 1000 já começámos a obter uma percentagem de erros, embora insignificativa. Nos 5000 utilizadores, a maioria deles já não consegue efetuar login. Esta percentagem de erros mostra-nos um pouco qual o limite da disponibilidade da aplicação de momento.

Ao aumentar o número de threads, verificámos também o aumento do *Throughput* da aplicação até um certo ponto, sendo que estagnou um pouco ao chegar às 500 threads. O tempo de resposta médio aos pedidos por sua vez não estagnou, subindo para valores elevados, mesmo sem a ocorrência de erros.

Nos gráficos apresentados, as **linhas vermelhas** representam o pedido **GET /login**, as **linhas azuis** representam o pedido **POST /graphql**, as **linhas amarelas** representam o pedido **GET /** e as **linhas verdes** representam o **Login** como um todo. Através destes gráficos, fica ainda mais evidente que os tempos de resposta da aplicação são bastante elevados.

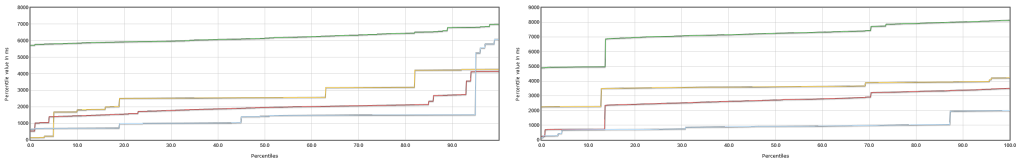

Figura 5: Percentil Tempo de Resposta para 100 e 250 Threads

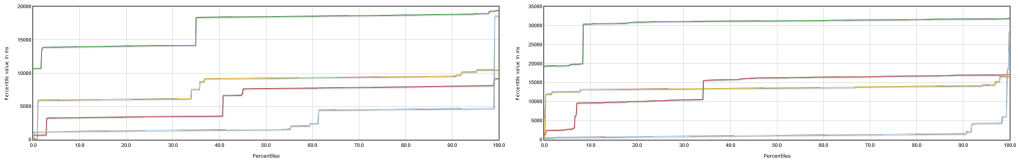

Figura 6: Percentil Tempo de Resposta para 500 e 1000 Threads

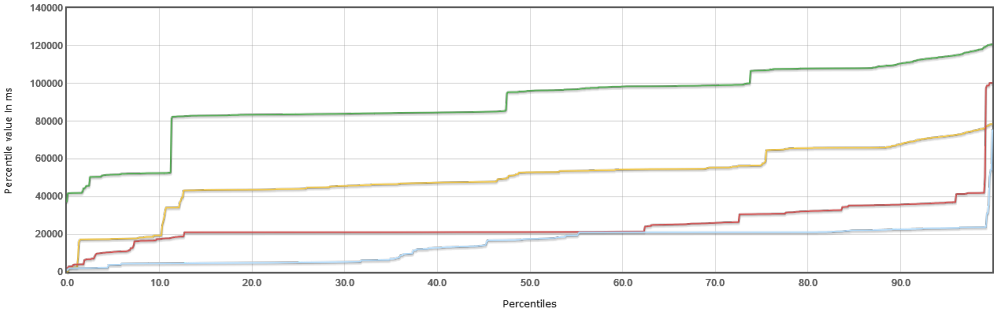

Figura 7: Percentil Tempo de Resposta para 5000 Threads

3.2.4 Teste 4

O Teste 4 visa representar o cenário em que um administrador efetua login no sistema, sendo redireccionado para a página principal, seguido da criação de uma nova página wiki. Ao correr este teste são testados os seguintes métodos:

1. GET /login
2. POST /graphql
3. GET /
4. GET /e/en/ThreadNum-Pagename
5. POST /graphql
6. GET /e/en/ThreadNum-Pagename

Nos métodos podemos encontrar as palavras *ThreadNum* e *Pagename*, que na verdade são variáveis. Cada *thread* onde corremos os testes terá um *ThreadNum* único, de maneira a conseguirmos saber que páginas cada *thread* criou, e optamos por dar o nome à página a data do momento em que é criada em milissegundos para sabermos quando e por que ordem foram criadas as novas páginas. Isto é possível tomando partido das funções `__time()` e `__threadNum` do *JMeter*.

O teste foi corrido com 100, 250 e 500 *threads*, sendo que recorremos aos relatórios gerados pelo *JMeter* para apresentar os seguintes resultados. A única diferença entre este teste e o anterior é que agora criamos uma página nova depois de fazermos o login, enquanto que no outro o teste acabaria precisamente depois de realizarmos o login. O processo de criação duma nova página é constituído pelos 3 últimos pedidos apresentados, sendo que o conteúdo do POST possui o conteúdo que a nova página irá apresentar. Estes 3 pedidos foram encapsulados num processo mais generalizado que representa a criação duma nova página como um todo. Optámos por esta visão de modo a representar de modo mais fidedigno o ato de criar uma nova página, dado que não é possível criá-la se uma das operações falhar. Sendo assim, preferimos separar a operação de efetuar login da criação da nova página para podermos avaliar individualmente cada um dos processos.

Threads	Operação	Erro	Tempo Resposta Médio (ms)	*Throughput (Trans./s)*
100	Login	0.00%	4686.52	17.05
	New Page	18.00%	244610.95	0.40
250	Login	0.00%	12209.70	18.49
	New Page	37.14%	369123.91	0.46
500	Login	0.00%	23461.58	19.98
	New Page	84.25%	345850.68	1.30

Tabela 4: Instalação Simples - Sumário do Teste 4

Analisando os dados da tabela, podemos verificar que as operações de login não possuem qualquer erro, enquanto que as da criação de páginas apresentam sempre erros, aumentando com o número de *threads*. Podemos dizer que os tempos

de resposta também aumentam proporcionalmente com o número de *threads* mas o *throughput* fica estagnado, seja qual for a operação e também o número de *threads*!

Nos gráficos apresentados, as linhas **vermelho-escuro, roxo e amarelo-escuro** são as mais visíveis e ao mesmo tempo preocupantes, representando o **login, New Page e GET Page**, respetivamente.

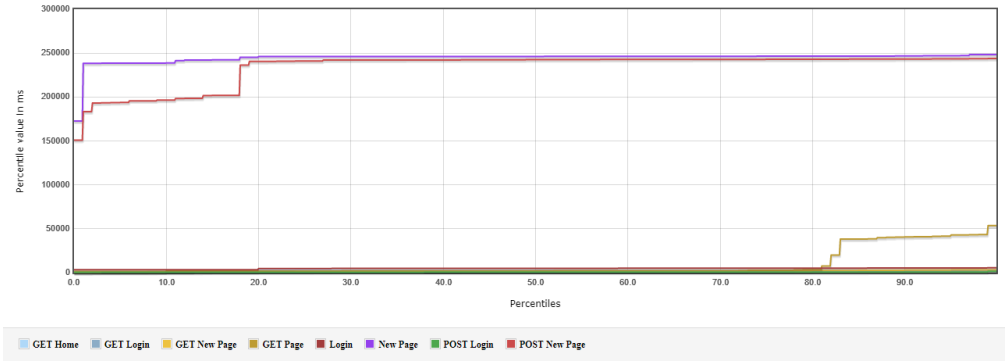

Figura 8: Percentil Tempo de Resposta para 100 Threads

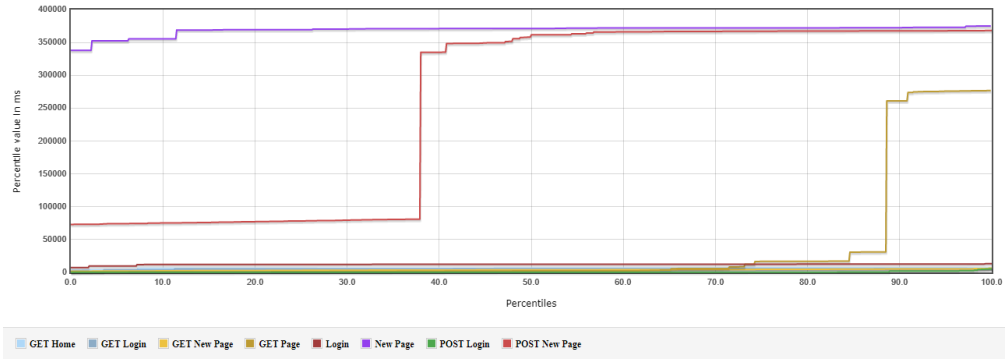

Figura 9: Percentil Tempo de Resposta para 250 Threads

3.3 Análise de Desempenho

Após analisarmos os resultados do **Testes de Carga** listados na secção anterior, somos agora capazes de efetuar uma análise face ao desempenho desta versão inicial.

A primeira conclusão a que chegámos é que apesar de um pedido ser atendido, não significa que o sistema esteja a operar de modo aceitável, isto é, possua

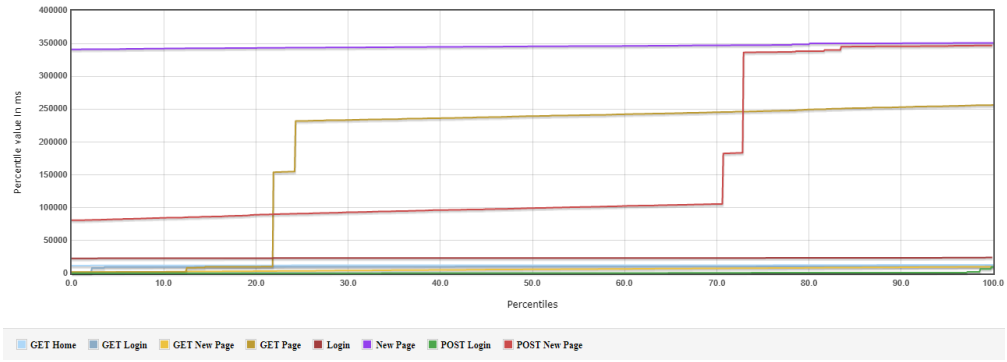

Figura 10: Percentil Tempo de Resposta para 500 Threads

tempos de resposta aceitáveis ou as respostas representem o estado esperado do sistema. Nos nossos testes, mesmo com percentagens de erro nulas, os tempos de resposta médios foram bastante elevados. Isto confirma a nossa suspeita acerca de ***bottlenecks*** no nosso sistema.

Outra conclusão que tirámos prende-se nas percentagens de erro relativas a situações em que temos números de utilizadores mais elevados. Quando o número de utilizadores sobe, a nossa plataforma baixa consideravelmente a sua disponibilidade, chegando a ponto em que fica inoperacional.

As conclusões obtidas através dos testes de carga em conjunto com a possibilidade de um dos nossos elementos falhar, tal como visto na Secção 3.1, levam-nos a crer que a nossa implementação do Wiki.js é de **baixa disponibilidade** e possui uma **performance baixa**. Isto leva-nos à necessidade de implementar o Wiki.js num ambiente diferente, tal como iremos ver no capítulo a seguir.

4 Arquitetura Implementada

Nesta Secção iremos descrever a arquitetura que implementámos para obter uma infraestrutura de elevada disponibilidade e desempenho, explicando quais as ferramentas utilizadas e explicando como estas resolvem os problemas identificados anteriormente. De modo a comparar a arquitetura implementada com a anterior, iremos efetuar os mesmos testes de carga de modo a comparar as duas soluções para poder efetuar comparações.

4.1 Camada de Persistência

Para a camada de persistência do sistema, iremos apresentar as componentes que a constituem pela ordem de implementação.

A primeira componente desta camada consiste num grupo de VM's que constituem um **DRBD** (*Distributed Replicated Block Device*) que são acedidas utilizando o protocolo **iSCSI** (*Internet Small Computer Systems Interface*). Para esta componente criámos **duas instâncias de VM** na GCP com **2 vCPU** e **4GB de memória**. Cada uma das máquinas possui um disco **SSD de 20GB** onde foi instalado o sistema operativo (CentOS 8). A utilização de DRBD deve-se ao facto de este permitir garantir a replicação dos nossos dados. Para o armazenamento dos dados, cada uma das instâncias possui um **disco SSD de 50 GB** que irá ser atribuída a um recurso DRBD, sendo que o denominámos de **d1**. Ambas as instâncias foram sinalizadas como primárias para permitir que estas sejam utilizadas sem qualquer tipo de restrição. Para finalizar, numa das instâncias foi colocado um sistema de ficheiros no formato xfs, sendo estas alterações sincronizadas com a segunda instância. Após finalizada o setup do DRBD, avançámos para o setup do iSCSI *target*.

A segunda componente desta camada consiste num grupo de VM's que constituem um cluster de elevada disponibilidade. Este cluster é constituído por **3 nodos**, tendo cada um **2 vCPU** e **8GB de memória**. Cada uma das máquinas possui um disco **SSD de 20GB** onde foi instalado o sistema operativo (CentOS 8). Cada uma das instâncias foi configurada com o cliente **iSCSI**, ligando-se a cada um

dos *targets*. Após esta configuração do iSCSI foi colocado um serviço **multipath**. De seguida foi feito o setup do ***High-Availability Cluster*** da RedHat em cada uma das três instâncias, sendo este cluster composto por três nodos: **cluster1**, **cluster2** e **cluster3**. Neste cluster foram definidos dois serviços: o serviço **fs**, que consiste no sistema de ficheiros replicado anteriormente criado, e o serviço **bd**, que consiste na configuração da base de dados postgres que pretendemos manter.

A terceira e última componente que compõe a camada de persistência é um ***(tcp) load balancer*** interno, que tem como principal função encaminhar os pedidos para o nó *director* e, também, verificar se cada um dos nós do cluster se encontra operacional. Na eventualidade de o nó ativo ficar inoperacional, o *load balancer* automaticamente comunica com o nó seguinte. Para conseguir isto agrupámos os nós do cluster em dois grupos: um grupo primário e um grupo secundário. O cluster1 foi adicionado ao grupo primário, sendo este o principal nodo do cluster. O cluster2 e o cluster3 foram adicionados ao grupo secundário, sendo que no caso do nodo principal falhar, o *failover* é feito para um destes nodos.

O acesso a esta camada de persistência é então feito a partir do *load balancer*. Deste modo, possuímos uma camada de persistência que nos garante elevada disponibilidade bem como replicação do disco com os dados, isto é, no caso de um dos elementos que compõe esta camada falhar, o serviço não é interrompido, e no caso de um dos discos de armazenamento falhar, possuímos uma cópia de todos os dados, mantendo-se o serviço sem interrupções. Com a utilização de discos SSD somos capazes de implementar melhorias de *performance*, tornando o nosso sistema mais rápido no geral.

4.2 Camada Aplicacional

A camada aplicacional tem a responsabilidade de tratar os pedidos e responder-lhes devidamente. Tal como se pode ver na figura 11, para esta camada decidiu-se ter três servidores semelhantes, cada um com a aplicação *Wiki.js* instalada. Tanto para a camada aplicacional como para a camada web, decidiu-se ter VMs semelhantes com:

- 2 CPUs;
- 8 GB de Memória RAM;
- 10 GB de disco;
- Sistema operativo Ubuntu 20.04 LTS.

A razão para ter 3 servidores semelhantes na camada aplicacional serve o propósito de uma maior paralelização no atendimento de pedidos, o que de um modo geral permite diminuir os tempos de resposta, isto é, tornar as operações mais rápidas e distribuir a carga para várias instâncias da aplicação, ao invés de sobrecarregar uma única instância da aplicação.

A razão anterior foca-se no desempenho, outra razão tem que ver com elevada disponibilidade, pois, se um dos servidores entrar em falta e não for capaz de responder a pedidos, temos os outros dois servidores a responder. No limite, podem entrar em falta 2 dos 3 servidores aplicacionais. No caso dos 3 servidores falharem, o serviço fica indisponível.

Como nota final relativamente ao *deployment* desta camada, para facilitar a manutenção das aplicações, em cada VM criou-se um serviço associado à aplicação, que foi utilizado para pôr as aplicações a correr e que facilita qualquer tipo de manutenção das mesmas através das premissas de *enable*, *disable*, *start*, *stop*, *restart*, etc dos serviços *Linux*.

4.3 Camada Web (*Load Balancers*)

A camada mais superficial (mais próxima da *Internet*) é a camada *web*. Esta tem a responsabilidade de receber os pedidos e distribuí-los pelos vários servidores aplicacionais existentes (*load balancer* e *reverse proxy*), bem como controlar o tráfego, através de regras de *firewall*.

Esta camada dividi-se em duas sub-camadas:

- Um load balancer externo que serve como ponto de acesso ao sistema (é o componente mais superficial de toda a estrutura) e distribui os pedidos pelos servidores *web*;

- Um conjunto de 3 servidores *web* que distribuem os pedidos pelos vários servidores aplicacionais.

Olhando primeiro para os servidores *web*, em si, cada um destes corre uma instância de ***NGINX***, configurada para receber pedidos e distribuí-los pelos 3 servidores aplicacionais. Com o objetivo de melhorarmos o tipo de distribuição de carga decidimos usar a opção ***least_conn*** que o NGINX fornece e que distribui a carga pelos vários servidores aplicacionais tendo em conta o seu número de conexões ativas.

Novamente, a decisão de ter 3 servidores *web* segue a mesma lógica dos servidores aplicacionais: por um lado, dar alto desempenho ao aumentar o número de servidores *web*, ao invés de ter apenas um, diminuindo a carga que cada um recebe e, por outro lado, ter alto disponibilidade pois, o sistema permite que, no limite, falhem 2 dos 3 servidores *web*, sem que o sistema fique comprometido e/ou indisponível.

De notar que, tanto nos servidores *web* como nos servidores aplicacionais a infraestrutura escolhida permite remover o problema do *Single Point of Failure* (SPOF).

Por fim, temos o *load balancer* externo para servir como ponto único de acesso ao sistema pela Internet/ *Web*. Este *load balancer* distribui os pedidos que recebe pelos 3 servidores *web* e, apesar de trazer a vantagem de permitir um ponto único de acesso e de abstrair o utilizador final de toda a estrutura que está por trás do sistema, tem a desvantagem de que se este serviço falha, todo o sistema fica indisponível para os utilizadores. Tendo isto em conta e de forma a limitar esta desvantagem, utilizámos um serviço da *Google Cloud Platform*: um *HTTP External Load Balancer* que, segundo a própria *GCP*, tem elevada disponibilidade e uma probabilidade de falha muito baixa. Este serviço permite, então, distribuir os pedidos que recebe pelas 3 instâncias do servidor *web* e tem, também, *healthchecks* de 30 em 30 segundos para os servidores *web*, de forma a conseguir perceber quais deles é que estão disponíveis para receber pedidos e assim adaptar o seu *routing* do tráfego.

4.4 Representação Geral da Arquitetura

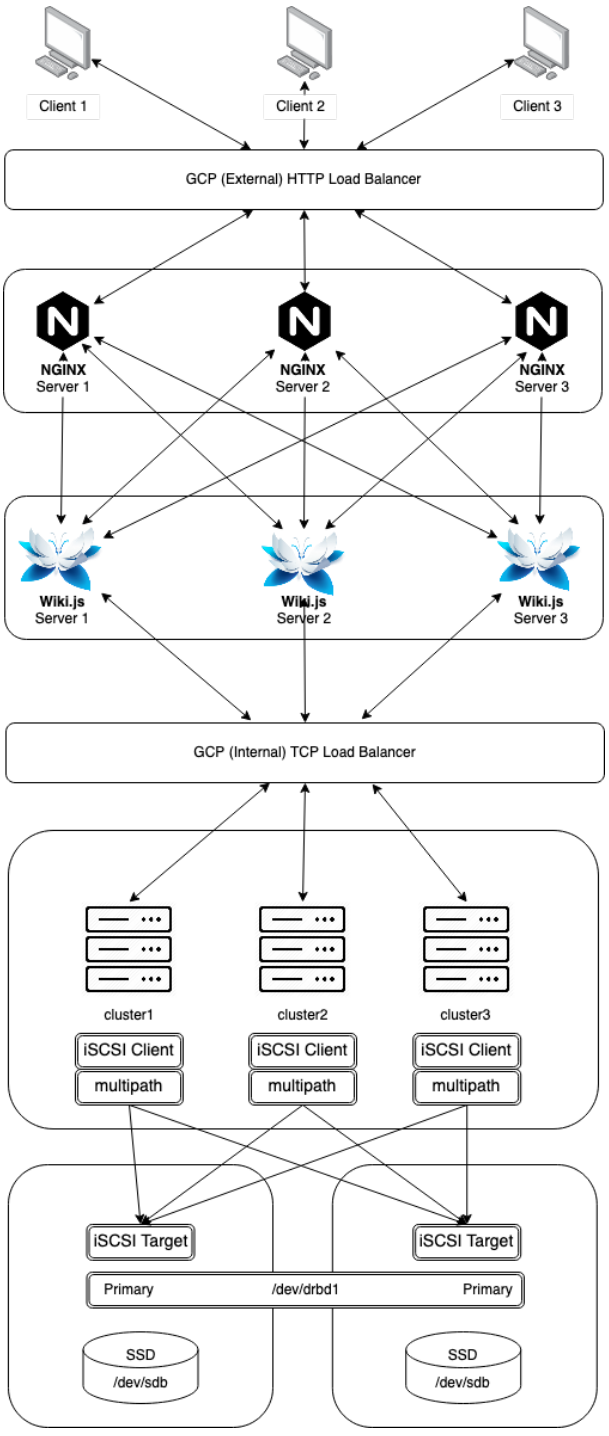

Figura 11: Arquitetura Implementada

4.5 Testes de Carga

Uma vez implementada a topologia apresentada na Figura 11, voltámos a realizar os mesmos teste de carga já referidos, mas desta vez nesta nova arquitectura de alta disponibilidade e desempenho.

Nota: de referir que para facilitar a leitura e suprimir a necessidade de voltar atrás para se perceber qual o objetivo dos demais testes, as descrições apresentadas acima voltam a ser apresentadas nesta secção.

4.5.1 Teste 1

Tal como previamente descrito, o teste 1 visa representar o cenário mais básico: um utilizador aceder à página inicial da wiki. Para a realização deste teste, irão ser utilizadas mais *threads* do que nos restantes, dado que num cenário real, o acesso à página inicial da wiki será a ação mais comum. O método que será testado é o seguinte:

1. GET /

O teste foi corrido com 100, 1000, 5000 e 10000 *threads* (utilizadores), sendo que recorremos aos relatórios gerados pelo *JMeter* para apresentar os seguintes resultados:

Página Inicial			
Threads	Erro	Tempo Resposta Médio (ms)	*Throughput (Trans./s)*
100	0.00%	89.32	98.81
1000	0.00%	2929.41	124.64
5000	35.20%	14183.61	132.07
10000	53.29%	18069.55	199.77

Tabela 5: Deployment - Sumário do Teste 1

Realizando uma análise aos dados da tabela, podemos, novamente, verificar que nas situações em que utilizamos 100 e 1000 utilizadores não obtemos nenhuma percentagem de erro. À medida que esses valores sobem para a casa dos 5000 já começámos a obter uma percentagem significativa de erros, sendo que com 10000 essa percentagem é ainda maior. Esta situação também se repetia na tipologia

anterior, porém, em comparação, a percentagem de erros desta versão sofreu uma redução considerável em ambos os casos.

Para além disso e tal como esperado, o aumento do número de threads provocou tanto o aumento do *throughput* como dos tempos médios de resposta da aplicação. Tal como no caso da percentagem de erros, comparativamente à versão anterior, ambos os valores sofreram uma **enorme** melhoria.

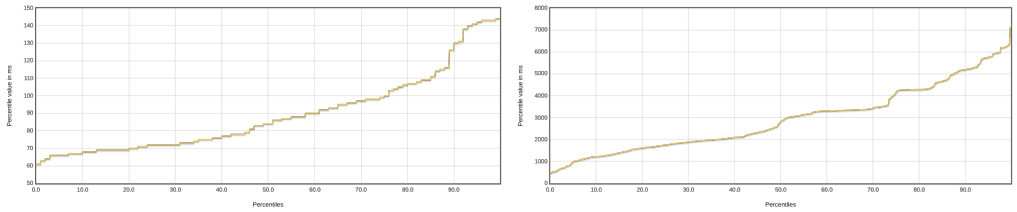

Figura 12: Percentil Tempo de Resposta para 100 e 1000 Threads

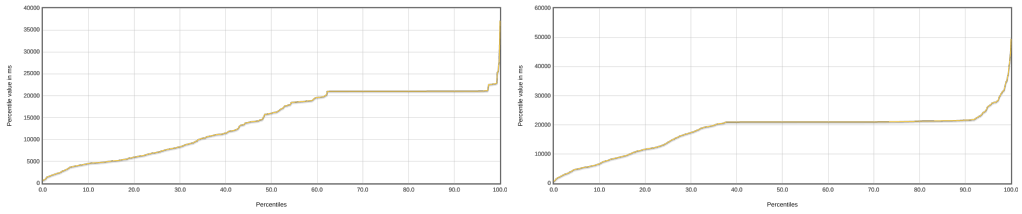

Figura 13: Percentil Tempo de Resposta para 5000 e 10000 Threads

Por fim, este primeiro teste veio demonstrar que contrariamente à implementação anterior, a nova arquitectura, que implementámos, é capaz de fornecer tempos de resposta perfeitamente aceitáveis, mas com uma percentagem de erro que, apesar de ter sofrido uma melhoria considerável, permanece alta.

4.5.2 Teste 2

O Teste 2 visa representar o cenário em que um utilizador entra na página inicial, navegando de seguida para uma outra página da wiki. Neste caso em específico, todas as *threads* abrem a mesma página após pedirem a página inicial e esperarem 1000 milissegundos. Ao correr este teste são testados os seguintes métodos:

1. GET /
2. GET /en/ICD

O teste foi corrido com 1000 e 5000 *threads*, sendo que recorremos aos relatórios gerados pelo *JMeter* para apresentar os seguintes resultados. Estes valores foram escolhidos pela sua relevância tendo em conta o teste anterior.

Página Inicial			
Threads	Erro	Tempo Resposta Médio (ms)	*Throughput (Trans./s)*
1000	0.00%	1439.57	201.03
5000	24.26%	9570.59	226.88

Tabela 6: Deployment - Sumário do Teste 2

Analisando os dados da tabela, podemos verificar que com 1000 utilizadores não obtemos nenhuma percentagem de erro. Quando aumentamos para valores na casa dos 5000 começámos a obter uma percentagem significativa de erros.

Ao aumentar o número de threads, verificámos também o aumento do *Throughput* da aplicação, bem como do tempo de resposta médio aos pedidos. Este último valor, dado que também serve de medida de performance do sistema, mostra-nos que nos casos em que assegurámos que todos os pedidos são atendidos, estes esperam uma quantidade de tempo considerável.

Em relação à arquitectura inicial, podemos ver que o tempo de resposta médio diminui drasticamente para ambos os números de threads, bem como o aumento do *throughput*. No entanto, este último teve valores semelhantes para número de threads diferentes, o que não acontecia anteriormente. A percentagem de erros também diminui significativamente, para metade no caso das 5000 threads.

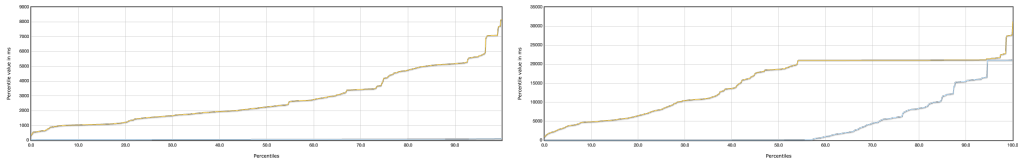

Figura 14: Percentil Tempo de Resposta para 1000 e 5000 Threads

Verificando os gráficos apresentados, em que a linha azul representa os pedi-

dos referentes à página de teste e a linha laranja os pedidos referentes à página inicial. Ao contrário da arquitectura inicial, podemos ver que há agora uma grande diferença de tempos de resposta entre as duas operações.

4.5.3 Teste 3

O Teste 3 visa representar o cenário em que um administrador efetua login no sistema, sendo redirecionado para a página principal. Ao correr este teste são testados os seguintes métodos:

1. GET /login
2. POST /graphql
3. GET /

O teste foi corrido com 100, 250, 500, 1000 e 5000 *threads*, sendo que recorremos aos relatórios gerados pelo JMeter para apresentar os seguintes resultados. O processo de login é constituido pelos 3 pedidos apresentados, sendo que o conteúdo do POST que é feito possui o endereço de email e a password da conta utilizada no teste. Estes 3 pedidos foram encapsulados num processo mais generalizado que representa o login como um todo. Optámos por esta visão de modo a representar de modo mais fidedigno o ato de efetuar login, dado que não é possível efetuar login se uma das operações falhar. Outra atenção que tivemos, foi o cancelamento de uma *thread* no caso de um dos pedidos falhar.

Página Inicial			
Threads	Erro	Tempo Resposta Médio (ms)	*Throughput (Trans./s)*
100	0.00%	1987.23	39.48
250	0.00%	2397.48	83.31
500	0.00%	2951.27	124.53
1000	0.00%	3185.77	120.18
5000	25.78%	12737.36	95.51

Tabela 7: Deployment - Sumário do Teste 3

Analisando os dados da tabela, podemos verificar que com 100, 250, 500 e 1000 não obtemos nenhuma percentagem de erro. Ao aumentar o número de threads,

verificámos também o aumento do *Throughput* da aplicação até um certo ponto, sendo que começou a diminuir a partir das 1000 threads. O tempo de resposta médio aos pedidos por sua vez não estagnou, subindo para valores elevados, mesmo sem a ocorrência de erros.

Em relação à arquitectura inicial, podemos ver que o tempo de resposta médio diminui drasticamente para todas as threads, bem como o aumento do *throughput*, apesar deste começar a diminuir a certo ponto. Isto significa que a partir de um dado ponto a performance se começa a degradar acentuadamente. Quanto à percentagem de erros conseguimos eliminar por completo no caso das 1000 threads enquanto que na de 5000 obtemos resultados 2.5 vezes melhores.

Nos gráficos apresentados, as **linhas vermelhas** representam o pedido **GET /login**, as **linhas azuis** representam o pedido **POST /graphql**, as **linhas amarelas** representam o pedido **GET /** e as **linhas verdes** representam o **Login** como um todo.

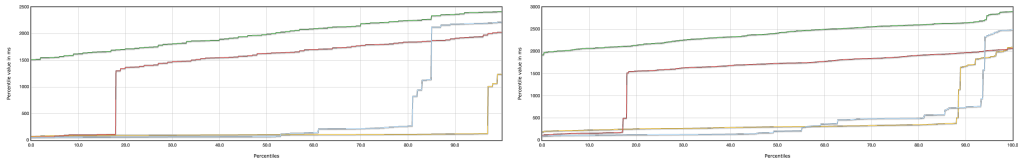

Figura 15: Percentil Tempo de Resposta para 100 e 250 Threads

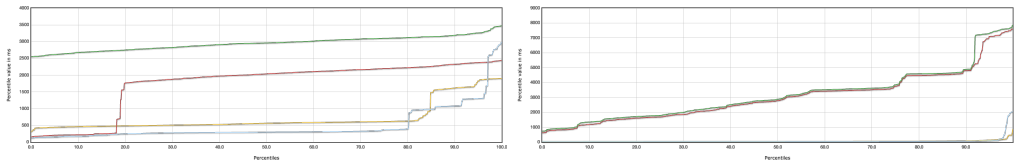

Figura 16: Percentil Tempo de Resposta para 500 e 1000 Threads

Ao contrário da arquitectura inicial, podemos ver que há agora uma grande diferença de tempos de resposta entre as operações. Anteriormente todas elas tinham a sua relevância enquanto que agora apenas as operações de LOGIN e GET LOGIN são comparativamente mais demoradas.

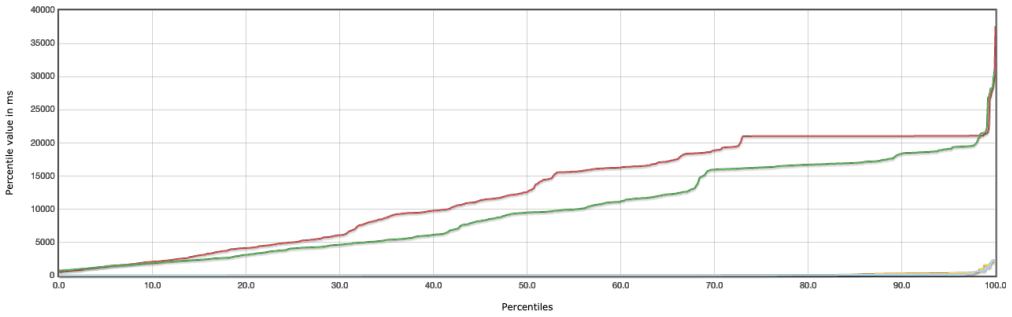

Figura 17: Percentil Tempo de Resposta para 5000 Threads

4.5.4 Teste 4

O Teste 4 visa representar o cenário em que um administrador efetua login no sistema, sendo redireccionado para a página principal, seguido da criação de uma nova página wiki. Este teste irá testar os seguintes métodos:

1. GET /login
2. POST /graphql
3. GET /
4. GET /e/en/ThreadNum-Pagename
5. POST /graphql
6. GET /e/en/ThreadNum-Pagename

Tal como explicado acima, nos métodos podemos encontrar as palavras *ThreadNum* e *Pagename*, que na verdade são variáveis. Cada *thread* onde corremos os testes terá um *ThreadNum* único, de maneira a conseguirmos saber que páginas cada *thread* criou, e optamos por dar o nome à página a data do momento em que é criada em milissegundos para sabermos quando e por que ordem foram criadas as novas páginas. Isto é possível tomando partido das funções `__time()` e `__threadNum` do *JMeter*.

O teste foi corrido com 100, 250 e 500 *threads*, sendo que recorremos aos relatórios gerados pelo *JMeter* para apresentar os seguintes resultados. A única diferença entre este teste e o anterior é que agora criamos uma página nova depois

de fazermos o login, enquanto que no outro o teste acabaria precisamente depois de realizarmos o login. O processo de criação duma nova página é constituído pelos 3 últimos pedidos apresentados, sendo que o conteúdo do POST possui o conteúdo que a nova página irá apresentar. Estes 3 pedidos foram encapsulados num processo mais generalizado que representa a criação duma nova página como um todo. Optámos por esta visão de modo a representar de modo mais fidedigno o ato de criar uma nova página, dado que não é possível criá-la se uma das operações falhar. Sendo assim, preferimos separar a operação de efetuar login da criação da nova página para podermos avaliar individualmente cada um dos processos.

Threads	Operação	Erro	Tempo Resposta Médio (ms)	*Throughput (Trans./s)*
100	Login	0.00%	3705.34	37.645
	New Page	0.00%	84141.29	4,31
250	Login	0.00%	3997.54	46.75
	New Page	0.00%	161938.23	2.32
500	Login	0.00%	6516.45	17.14
	New Page	0.00%	306854.25	1.7

Tabela 8: Deployment - Sumário do Teste 4

Realizando uma análise aos dados recolhidos, podemos verificar que as operações de login continuam a ser realizadas sem qualquer erro. Porém, contrariamente à percentagem de erros que a versão anterior evidenciava, esta nova topologia reduz essa percentagem para 0 em todas as situações. Para além disto, esta nova versão apresenta uma estagnação quando o número de utilizadores cresce, começando a haver uma degradação nos tempos de resposta e começando o *throughput* a cair. Isto leva-nos a crer que estamos a lidar com um número de utilizadores já acima da capacidade do sistema, começando os valores a ser pouco fidedignos.

Nos gráficos apresentados, as linhas **vermelho-escuro, roxo e amarelo-escuro** são as mais visíveis e ao mesmo tempo preocupantes, representando o **login, New Page e GET Page**, respetivamente.

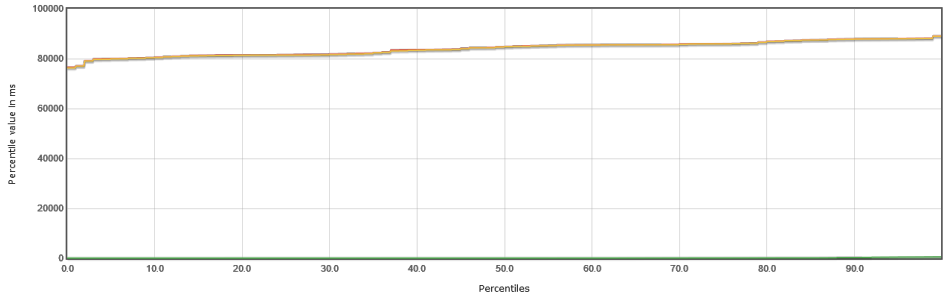

Figura 18: Percentil Tempo de Resposta para 100 Threads

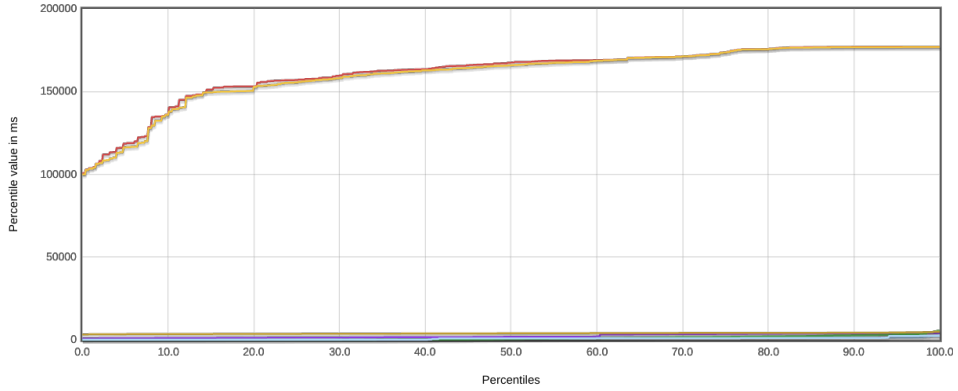

Figura 19: Percentil Tempo de Resposta para 250 Threads

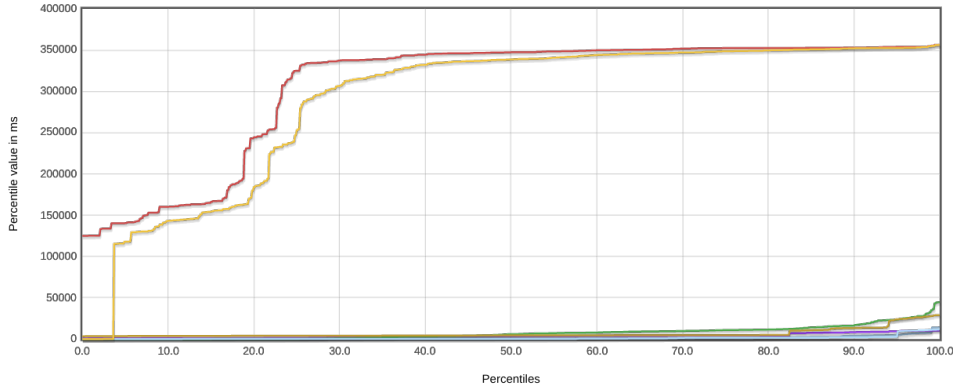

Figura 20: Percentil Tempo de Resposta para 500 Threads

Em conclusão, no que diz respeito ao teste 4, esta nova topologia demonstra um excelente redução da percentagem de erro para a criação de uma nova página, mostrando assim uma melhoria na disponibilidade da plataforma. No entanto

existem alguns aspetos a melhorar, nomeadamente a nível de performance, dado que gostaríamos de melhorar um pouco os tempos de resposta.

4.6 Análise de Desempenho

Neste ponto, tendo uma arquitectura distribuída, é de esperar um aumento de disponibilidade e performance da aplicação, como foi mostrado anteriormente através da comparação dos resultados dos testes de carga das arquiteturas Inicial e Final. No entanto, apenas as camadas aplicacionais e web correm paralelamente, o que explica que as melhorias mais significativas sejam em pedidos menos dependentes da base de dados. Dado que o principal foco foi a disponibilidade do sistema, a principal alteração na camada de persistência foi a replicação do armazenamento. Isto apenas nos fornece redundância dos dados, não garantindo melhorias de performance em operações na base de dados.

Olhando para o nosso principal objetivo, garantir elevada disponibilidade, podemos dizer que obtivemos sucesso, dado que todas as operações com uma percentagem de erro não nula diminuíram consideravelmente, e dado que o sistema é capaz de resistir a falhas de componentes. A nível de performance, os nossos testes de carga revelam que também obtivemos melhorias significativas, especialmente devido ao balanceamento de carga por parte das camadas *web* e aplicacional.

4.7 Mitigação dos Pontos Críticos do Sistema

Após a implementação da plataforma e execução de testes de carga, podemos avaliar os pontos críticos identificados na secção 3.1.

Um dos pontos críticos identificados foi a possibilidade de uma ou mais componentes do sistema falharem, causando uma interrupção do serviço. Esse problema foi mitigado, dado que os dados se encontram replicados e nenhuma componente do sistema possui uma só instância.

Foram também mencionados pontos críticos no sistema sob a forma de *bottlenecks*, tendo estes tendo sido maioritariamente mitigados. A nível dos servidores aplicacionais e *web*, com o uso de um balanceador de carga, eliminámos o *bottle-*

neck por completo. No entanto, na nossa base de dados, apesar das melhorias de performance, esta continua em parte a ser um *bottleneck* do sistema, dado que as operações mais lentas estão diretamente relacionadas com pedidos feitos à mesma. A conclusão a que chegámos é que para melhorar a performance do sistema como um todo, a implementação da base de dados terá de ser diferente. Como o nosso principal foco é a disponibilidade, optámos por manter a nossa arquitetura dado que satisfaz os nossos requisitos.

Apesar de resolver uma boa parte dos problemas existentes, existem alguns quer irão sempre persistir, e que por muito improváveis que possam ou não ser, devem ser considerados. Apesar de termos todas as componentes replicadas, se falharem os 3 nós do cluster de armazenamento em simultâneo, o nosso sistema irá falhar. O mesmo se aplica aos servidores NGINX e aos servidores Wiki.js, bem como às duas instâncias que compõem o drbd. Outro problema que poderá existir é o caso de um dos *load balancers* falhar, tornando o nosso sistema inoperacional.

Resumindo, no caso de todas as instâncias de uma das camadas falharem ou de um dos *load balancers* falhar, o nosso sistema ficará indisponível, apesar de se tratarem de situações improváveis.

Visto isto, podemos afirmar que a arquitetura implementada garante elevada disponibilidade.

5 Conclusões

No paradigma da evolução das nossas capacidades como estudantes de engenharia informática, a realização deste projeto permitiu-nos consolidar a aprendizagem da UC de Infraestutura de Centro de Dados, mais concretamente, os métodos e ferramentas que devem ser aplicados no que toca à resolução do problema de planear, analisar e operar uma infraestrutura de elevada disponibilidade e desempenho.

O planeamento deste tipo de infraestruturas é uma das fases essenciais para que a aplicação seja capaz de corresponder a todas as exigências capazes de surgirem. Sendo ainda mais importante, quando estamos a planear uma tipologia que servirá como base de um serviço de grande escala, devido ao requisito de escalabilidade.

Apesar de considerarmos que tivemos sucesso na implementação deste tipo de arquitetura, consideramos que existem aspetos nos quais poderíamos melhorar em iterações futuras, nomeadamente a diversificação de testes realizados, de modo a obter mais cenários de uso, e, a execução de uma maior quantidade de testes e modo a obter valores mais precisos.

A Wiki.js - configuração simples com Docker Compose

```
version: "3.8"
services:
  db:
    image: postgres:alpine
    environment:
      POSTGRES_DB: wiki
      POSTGRES_PASSWORD: wikijsrocks
      POSTGRES_USER: wikijs
    logging:
      driver: "none"
    restart: unless-stopped
    volumes:
      - db-data:/var/lib/postgresql/data

  wiki:
    image: requarks/wiki:2
    depends_on:
      - db
    environment:
      DB_TYPE: postgres
      DB_HOST: db
      DB_PORT: 5432
      DB_USER: wikijs
      DB_PASS: wikijsrocks
      DB_NAME: wiki
    restart: unless-stopped
    ports:
      - "80:3000"

volumes:
  db-data:
```

Referências

[1] *Wiki.js: A Modern Open-source Wiki Engine for the Enterprise*, "https://medevel.com/wikijs/", Acedido: 09-12-2020.

[2] *Wiki.js requirements*, "https://docs.requarks.io/install/requirements#database", Acedido: 11-12-2020.

[3] *What is PostrgeSQL?* "https://www.postgresql.org/about/", Acedido: 11-12-2020.

[4] *Install Wiki.js using Docker and Docker Compose*, "https://docs.requarks.io/install/docker", Acedido: 01-12-2020.