

UNIVERSIDADE DO MINHO
MESTRADO INTEGRADO EM ENGENHARIA INFORMÁTICA
ENGENHARIA DE APLICAÇÕES

Infraestruturas de Centros de Dados

Bruno Teixeira, a84430
Carolina Cunha, A80142
João Diogo Mota, A80791
Valeriy Apostolyuk, PG42647

Grupo 3

29 de junho de 2021

Conteúdo

1	Introdução	3
2	Arquitetura da <i>Wiki.js</i>	4
2.1	Descrição da Arquitetura	4
2.2	Identificação de Pontos Críticos de Falha	5
2.3	Desempenho da Aplicação	5
2.3.1	Login	5
2.3.2	Login + Consulta das páginas criadas	8
2.3.3	Login + Registo de utilizador + Consulta de utilizadores	10
2.4	Disponibilidade da Aplicação	12
3	Arquitetura Proposta	13
3.1	Desenho da Arquitetura	13
3.2	Descrição da Arquitetura	14
3.2.1	DRBD	14
3.2.2	Cluster	14
3.2.3	Web Servers	14
3.2.4	Balanceadores de Carga	14
3.3	Resolução dos Pontos Críticos	14
3.3.1	Web Servers	14
3.3.2	Cluster	14
3.3.3	DRBD	15
3.4	Política de Balanceamento	15
3.5	Desempenho da Aplicação: Testes de Carga	15
3.5.1	Login	15
3.5.2	Login + Consulta das páginas criadas	17
3.5.3	Login + Registo de utilizador + Consulta de utilizadores	18
3.6	Disponibilidade da Aplicação	21
4	Discussão dos Resultados Obtidos	22

Lista de Figuras

2.1	Arquitetura padrão da aplicação	4
2.2	Resultados Gráficos - <i>login</i> (100 threads)	6
2.3	Resultados Gráficos - <i>login</i> (500 threads)	7
2.4	Resultados Gráficos - <i>login</i> (1000 threads)	7
2.5	Resultados Gráficos - <i>login</i> (1500 threads)	7
2.6	Resultados Gráficos - <i>login</i> (2000 threads)	7
2.7	Erro de tentativa de autenticação	7
2.8	<i>Token</i> de Acesso	8
2.10	Resultados Gráficos - <i>login + consulta</i> (500 threads)	9
2.11	Resultados Gráficos - <i>login + consulta</i> (1000 threads)	9
2.12	Resultados Gráficos - <i>login + consulta</i> (1500 threads)	9
2.13	Erro de tentativa de registo de utilizador	11
2.14	Resultados Gráficos - <i>login + registo + consulta</i> (100 threads)	11
2.15	Resultados Gráficos - <i>login + registo + consulta</i> (500 threads)	11
2.16	Resultados Gráficos - <i>login + registo + consulta</i> (1000 threads)	12
2.17	Resultados Gráficos - <i>login + registo + consulta</i> (1500 threads)	12
2.18	Resultados Gráficos - <i>login + registo + consulta</i> (2000 threads)	12
3.1	Desenho da Arquitetura Implementada	13
3.2	Resultados Gráficos com Arquitetura - <i>login</i> (100 threads)	16
3.3	Resultados Gráficos com Arquitetura - <i>login</i> (500 threads)	16
3.4	Resultados Gráficos com Arquitetura - <i>login</i> (1000 threads)	17
3.5	Resultados Gráficos com Arquitetura - <i>login</i> (1500 threads)	17
3.6	Resultados Gráficos com Arquitetura - <i>login</i> (falha de nodo CLI)	17
3.7	Resultados Gráficos com Arquitetura - <i>login</i> (2000 threads)	17
3.8	Erro de tentativa de registo de utilizadores	20
3.9	Resultados Gráficos com Arquitetura - <i>login + registo + consulta</i> (100 threads)	20
3.10	Resultados Gráficos - <i>login + registo + consulta</i> (200 threads)	20
3.11	Resultados Gráficos - <i>login + registo + consulta</i> (300 threads)	20
3.12	Resultados Gráficos - <i>login + registo + consulta</i> (400 threads)	21
3.13	Resultados Gráficos - <i>login + registo + consulta</i> (500 threads)	21

Lista de Tabelas

2.1	Resultados dos Testes da Aplicação - <i>Login</i> (Loop-count = 1)	6
2.2	Resultados dos Testes da Aplicação - <i>Login</i> (Loop-count = 10)	6
2.3	Resultados dos Testes da Aplicação - <i>Login + Consulta</i>	8
2.4	Resultados dos Testes da Aplicação - <i>Login + Registo + Consulta</i>	11
3.1	Resultados dos Testes com Arquitetura Implementada - <i>Login</i>	16
3.2	Resultados dos Testes com Arquitetura Implementada - <i>Login + Consulta</i>	18
3.3	Resultados dos Testes com Arquitetura Implementada - <i>Login + Registo + Consulta</i>	19

Capítulo 1

Introdução

Com a constante evolução da tecnologia e, naturalmente, do *software*, o programador assume uma responsabilidade crescente de planejar minuciosamente a arquitetura de uma infraestrutura em desenvolvimento.

Prevê-se, com este trabalho, o desenvolvimento e posterior avaliação de desempenho de uma arquitetura para um sistema da plataforma *Wiki.js*, utilizada para construir páginas *wiki*.

Este projeto tem como objetivo consolidar os conteúdos lecionados nas aulas da unidade curricular de Infraestruturas de Centros de Dados, concretizando o planeamento, *deployment*, análise de desempenho e operação de infraestruturas de elevada disponibilidade e desempenho.

Para este efeito, a realização do projeto seguiu a seguinte estruturação:

1. Planeamento e operacionalização da instalação da plataforma *Wiki.js*, garantindo alta disponibilidade e balanceamento de carga;
2. Avaliação do impacto dos mecanismos de balanceamento e alta disponibilidade introduzidos no desempenho da plataforma, realizando testes de carga.

No presente relatório, encontram-se descritos os processos de implementação do sistema, sendo abordadas sequencialmente: a análise e desenvolvimento da arquitetura, descrição e disponibilidade da arquitetura, possíveis pontos críticos de falha, testes de carga sobre a arquitetura e análise dos resultados obtidos em cada fase.

Capítulo 2

Arquitetura da *Wiki.js*

2.1 Descrição da Arquitetura

O *software* da plataforma *Wiki.js* encontra-se dividido em *frontend* e *backend* e utiliza uma base de dados *PostgreSQL* para armazenamento de dados.

Tanto o *frontend* como o *backend* estão escritos em *javascript*. O primeiro recorre à *framework* *Vue.js*. Ambos comunicam entre si através de uma API *GraphQL*. A aplicação executa com recurso ao *software* *Node.js*.

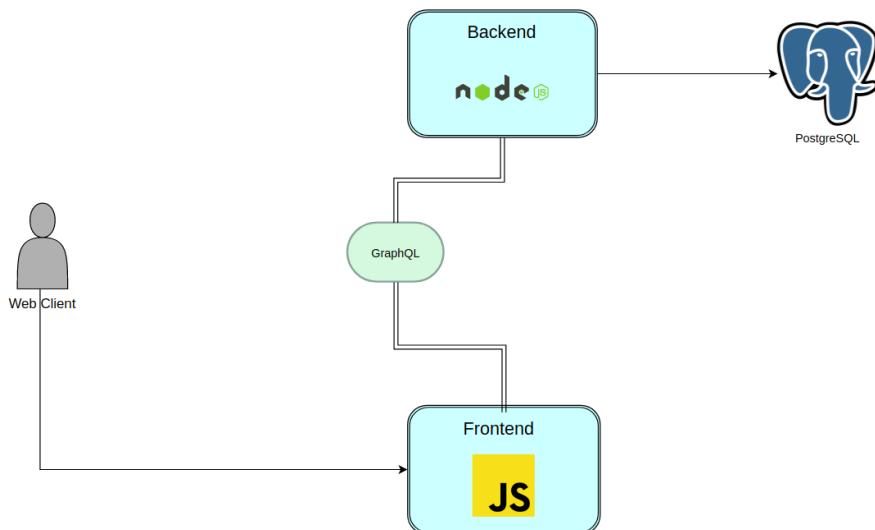


Figura 2.1: Arquitetura padrão da aplicação.

2.2 Identificação de Pontos Críticos de Falha

Um ponto crítico de falha, ou ponto único de falha (*Single Point of Failure*), é qualquer componente de um sistema que, perante uma falha, provoca a falência do sistema. Durante a projeção de um sistema, os pontos únicos de falha podem ser evitados implementando componentes redundantes e replicando partes críticas do sistema [1]. Isto porque, quando na presença de servidores únicos (isto é, sem qualquer réplica), a falha destes pode vulnerabilizar o sistema.

Posto isto, foi identificado como principal ponto crítico de falha da aplicação o seu *backend*, uma vez que a falha da camada de negócios provoca a falência de todos os seus serviços.

A base de dados dos *softwares Wiki* é um pilar fundamental neste sistema, uma vez que a maioria das operações são realizadas sobre a mesma, quer por escritores, editores e administradores. Consequentemente, a inacessibilidade aos dados em caso de falha incapacitaria toda a aplicação. Por último, o sustento que a aplicação tem na sua vertente *web* sofrerá um grande impacto em caso de falha, pelo que o *frontend* da aplicação é, também, um ponto de falha crítico.

2.3 Desempenho da Aplicação

Para avaliar o desempenho da aplicação, foram realizados testes de carga, recorrendo ao *Apache JMeter*.

O *Apache JMeter* é uma ferramenta que permite testar o desempenho em recursos estáticos e dinâmicos, simulando várias situações de pressão (uma carga pesada num servidor, grupo de servidores, rede ou objeto) para analisar o desempenho geral [2].

Para a realização dos testes de carga, é relevante o manuseamento das seguintes propriedades:

- *Number of threads* - Número de utilizadores concorrentes num grupo de *threads*;
- *Ramp-up period* - Período de *warm-up* de cache do sistema. O *JMteter* não irá contabilizar este período para o desempenho calculado;
- *Loop count*: Número de iterações que serão executadas por *thread*.

Em seguida, serão apresentados os diferentes testes de carga realizados sobre a plataforma.

2.3.1 Login

O primeiro teste consiste em entrar na página inicial do *Wiki.js* e, posteriormente, efetuar o *login* na aplicação. Para este efeito, são testados dos seguintes pedidos:

1. GET /login
2. POST /graphql

Body data:

{

```
"query": "mutation($username: String!, $password: String!, $strategy: String!) {\\n authentication {\\n login(username: $username,\\n password: $password,\\n strategy: \\$strategy) {\\n responseResult {\\n succeeded \\n errorCode \\n slug \\n message \\n } jwt\\n mustChangePwd \\n mustProvideTFA \\n mustSetupTFA \\n continuationToken \\n redirect\\n tfaQRImage \\n } \\n}}",\\n \"variables\": {\\n \"username\" : \"icd2021@uminho.pt\",\\n \"password\" : \"icdwikijs\",\\n \"strategy\" : \"local\"\\n }
```

```

    }
}

```

Este teste foi realizado para diferentes números de *threads*, simulando acessos concorrentes à aplicação por parte de vários utilizadores. Na Tabela 2.1, estão representados os valores de **tempo de resposta**, **throughput** e percentagem de **erro** dos respetivos pedidos.

Threads	Throughput (min)	GET		POST	
		Tempo de Resposta (ms)	Erro (%)	Tempo de Resposta (ms)	Erro (%)
100	19.82	286.51	0	222.00	0
500	99.04	221.58	0	114.99	0
1000	107.26	5915.72	0	3921.42	0
1500	131.19	6249.51	0	4323.31	0
2000	124.59	12452.38	0	7805.00	0

Tabela 2.1: Resultados dos Testes da Aplicação - *Login* (Loop-count = 1)

Assumindo valores de 10 para o *ramp-up period* e 1 para *loop count*, verifica-se que a aplicação é capaz de responder à totalidade dos pedidos realizados, sem qualquer percentagem de erro. No entanto, conclui-se que esta atinge a sua capacidade máxima entre os 1500 e os 2000 utilizadores, uma vez que o tempo de resposta aumenta significativamente com o aumento do número de *threads*, mas o débito (*throughput*) diminui, indicando que foi ultrapassado o limite máximo da aplicação.

De forma a testar a disponibilidade da aplicação, realizaram-se os mesmos testes para um *loop count* = 10.

Threads	Throughput (min)	GET		POST	
		Tempo de Resposta (ms)	Erro (%)	Tempo de Resposta (ms)	Erro (%)
100	111.98	450.16	0	458.15	0
500	117.62	3931.04	0	2748.21	0
1000	175.36	5662.21	27.68	4644.15	0.11
1500	155.59	8212.46	35.01	8591.28	0.15
2000	93.44	16901.24	21.82	22344.63	0.03

Tabela 2.2: Resultados dos Testes da Aplicação - *Login* (Loop-count = 10)

A análise dos dados permite verificar que, até 500 utilizadores, todos os pedidos são respondidos com sucesso (percentagem nula de erro). A partir de 1000 utilizadores verifica-se um acréscimo do erro, revelando uma redução da disponibilidade da aplicação. Constata-se que foi ultrapassado o limite máximo da aplicação entre os 1000 e 1500 utilizadores.

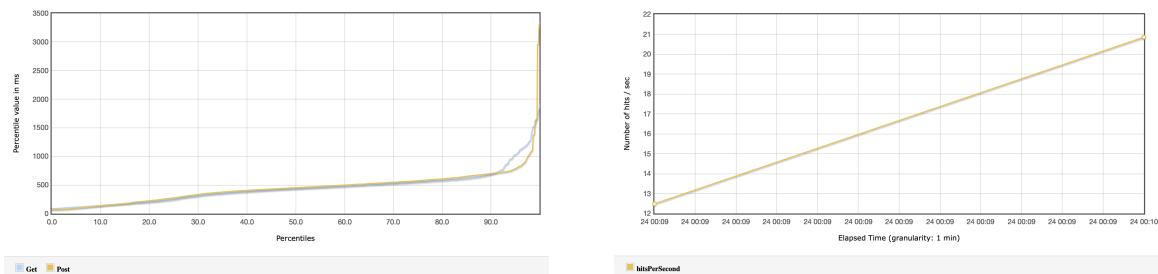


Figura 2.2: Resultados Gráficos - *login* (100 threads)

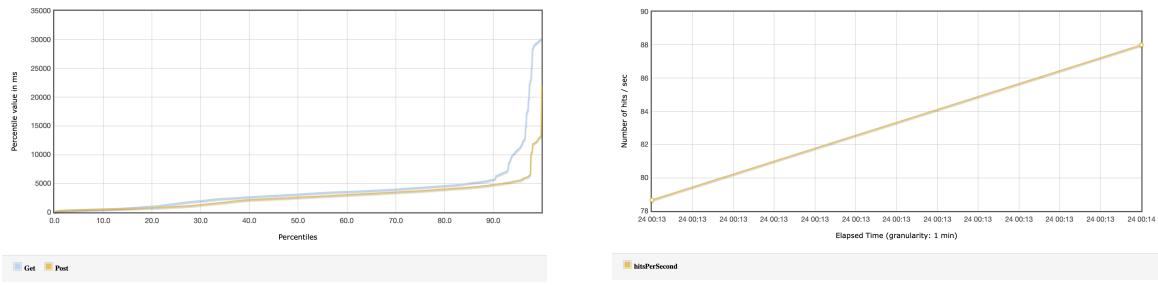


Figura 2.3: Resultados Gráficos - *login* (500 threads)

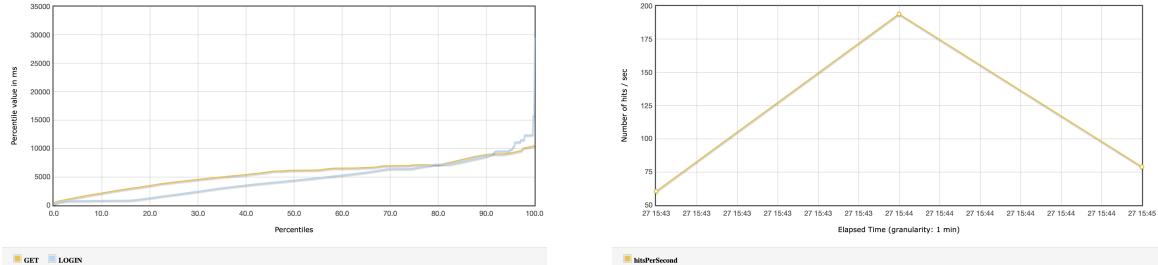


Figura 2.4: Resultados Gráficos - *login* (1000 threads)

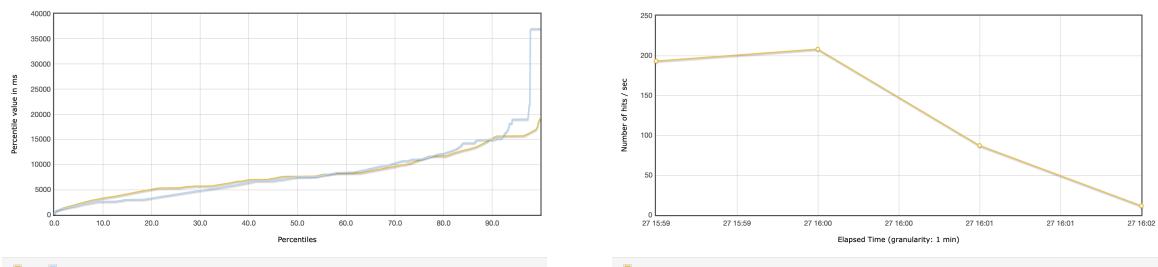


Figura 2.5: Resultados Gráficos - *login* (1500 threads)

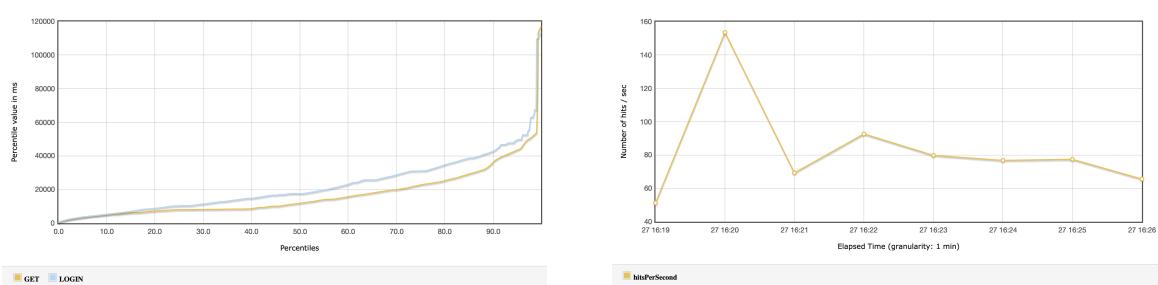


Figura 2.6: Resultados Gráficos - *login* (2000 threads)

Apesar da baixa percentagem de erro observada, a análise das mensagens *HTTP* recebidas revelam a presença de erros não identificados pelo *JMeter* aquando da autenticação do administrador, visível na Figura 2.7, impossibilitando a autenticação do utilizador na plataforma grande parte das vezes.

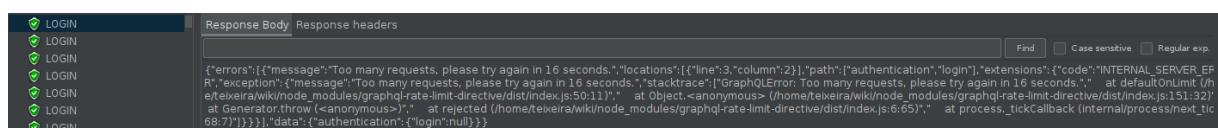


Figura 2.7: Erro de tentativa de autenticação

2.3.2 Login + Consulta das páginas criadas

Num segundo teste, foi realizado o *login* do administrador, onde este visualiza, posteriormente, a lista de todas as páginas *wiki* criadas.

1. GET /login
2. POST /graphql

Body data:

```
{
  "query": "mutation($username: String!, $password: String!, $strategy: String!) {\\n authentication {\\n login(username: $username,\\n password: $password,\\n strategy: $strategy) {\\n responseResult {\\n succeeded \\n errorCode \\n slug \\n message \\n } jwt \\n mustChangePwd \\n mustProvideTFA \\n mustSetupTFA \\n continuationToken \\n redirect \\n tfaQRImage } }}",
  "variables": {
    "username" : "icd2021@uminho.pt",
    "password" : "icdwikijs",
    "strategy" : "local"
  }
}
```

3. GET /p/pages

Este teste foi realizado sob as mesmas condições que o teste anterior, com um *loop count* de 1. Verifica-se, na Tabela 2.3, os resultados obtidos neste teste.

Para testes onde foi necessário memorizar a autenticação do utilizador, recorreu-se a um *token* de acesso. Os *tokens* de acesso são utilizados na autenticação para permitir que uma aplicação aceda a uma API. Este *token* atua como uma credencial quando chama a API de destino, informando a autorização do utilizador para aceder à API e executar ações concedidas durante a sua autenticação [3]. Este *token* é adicionado nos parâmetros de *Header Manager* (Figura 2.8).

Name:	Value
Authorization	Bearer eyJhbGciOiSJUzI1NiIsInR5cCI6IkpXVCJ9.eyJhcGkiOiJEsImdyCi6MSwiaWF0IjoxNjA4NzE
Content-Type	application/json

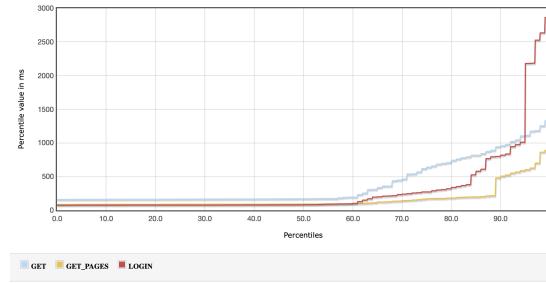
Figura 2.8: *Token* de Acesso

Threads	Throughput (min)	GET		POST		GET	
		Tempo de Resposta (ms)	Erro (%)	Tempo de Resposta (ms)	Erro (%)	Tempo de Resposta (ms)	Erro (%)
100	29.54	387.95	0	318.06	0	168.71	0
500	91.17	1354.90	0	1030.37	0	1051.56	0
1000	97.15	7274.45	0	5409.86	0.2	5364.70	9.90
1500	127.06	9680.37	0	6369.05	0.40	5109.64	45.73
2000	133.65	14132.03	0	8522.96	0.3	6821.88	51.40

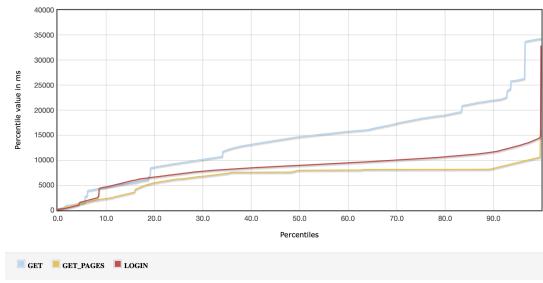
Tabela 2.3: Resultados dos Testes da Aplicação - *Login + Consulta*

É evidente a discrepância entre os tempos de respostas dos pedidos para autenticação na plataforma e do pedido de consulta das páginas criadas. Esta deve-se à latência proveniente da verificação e validação das credenciais de autenticação.

Analogamente ao teste anterior, a aplicação demonstra ser capaz de responder corretamente a todos os pedidos provenientes de 500 utilizadores. No entanto, são também apresentadas as respostas de erro *HTTP*, sendo a percentagem de erro real superior à indicada (Figura 2.7).



(a) Resultados Gráficos - *login + consulta*
(100 threads)



(b) Resultados Gráficos - *login + consulta*
(2000 threads)

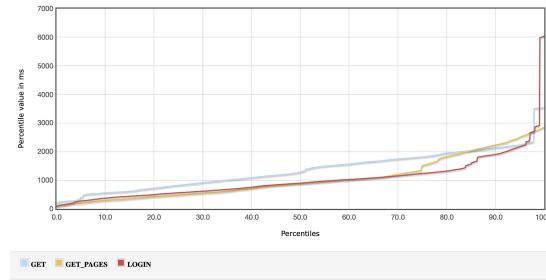


Figura 2.10: Resultados Gráficos - *login + consulta* (500 threads)

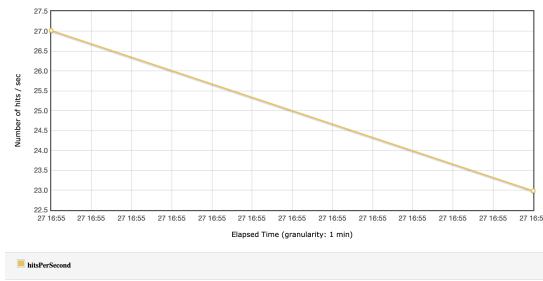
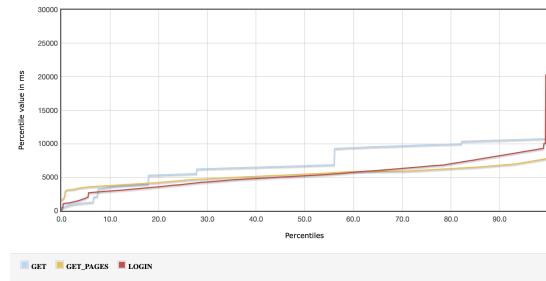
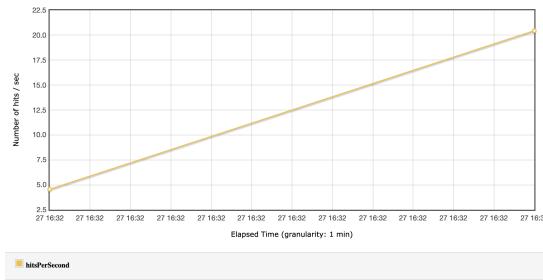


Figura 2.11: Resultados Gráficos - *login + consulta* (1000 threads)

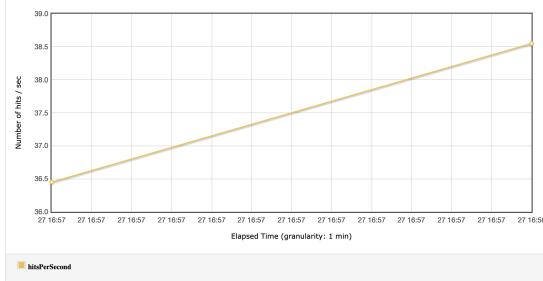
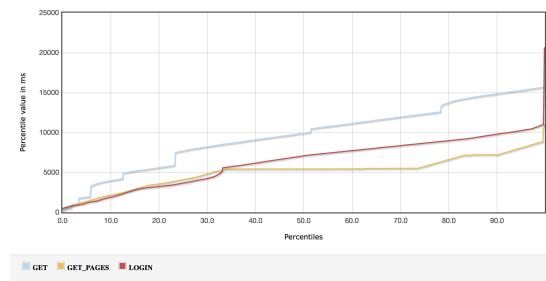


Figura 2.12: Resultados Gráficos - *login + consulta* (1500 threads)

2.3.3 Login + Registo de utilizador + Consulta de utilizadores

Neste teste, o administrador irá realizar a sua autenticação na plataforma e, em seguida, inserir um novo utilizador. Finalmente, vai consultar todos os utilizadores.

De modo a registar um utilizador, é indispensável que cada *thread* insira credenciais diferentes. Para este efeito, e recorrendo às funcionalidades do *JMeter*, o e-mail e nome de um utilizador será composto pela data atual e o número da *thread* em execução.

1. GET /login

2. POST /graphql

Body data:

```
{  
    "query": "mutation($username: String!, $password: String!, $strategy: String!) {\\n authentication {\\n login(username: $username,\\n password: $password,\\n strategy: $strategy) {\\n responseResult {\\n succeeded \\n errorCode \\n slug \\n message \\n } jwt \\n mustChangePwd \\n mustProvideTFA \\n mustSetupTFA \\n continuationToken \\n redirect \\n tfaQRImage } }}",  
    "variables": {  
        "username" : "icd2021@uminho.pt",  
        "password" : "icdwikijs",  
        "strategy" : "local"  
    }  
}
```

3. POST /graphql

Body data:

```
{  
    "query": "mutation ($providerKey: String!, $email: String!, $name: String!, $passwordRaw: String, $groups: [Int]!, $mustChangePassword: Boolean, $sendWelcomeEmail: Boolean) {\\n users {\\n create(providerKey: $providerKey, email: $email, name: $name, passwordRaw: $passwordRaw, groups: $groups, mustChangePassword: $mustChangePassword, sendWelcomeEmail: $sendWelcomeEmail) {\\n responseResult {\\n succeeded \\n errorCode \\n slug \\n message \\n } }}",  
    "variables": {  
        "providerKey" : "local",  
        "email" : "JM${__time()}_${__threadNum()}@teste.pt",  
        "name" : "JM${__time()}_${__threadNum()}",  
        "passwordRaw" : "testeteste",  
        "groups" : [3],  
        "mustChangePassword" : false,  
        "sendWelcomeEmail" : false  
    }  
}
```

4. GET /a/users

Uma vez que este teste é computacionalmente mais pesado que os anteriormente realizados, foi reduzido o número de *threads* utilizadas para simular os acessos concorrentes de utilizadores à aplicação. Foram mantidos os parâmetros de *ramp-up* = 10 e *loop-count* = 1.

Threads	Throughput (min)	GET		POST		POST		GET	
		Tempo de Resposta (ms)	Erro (%)						
100	9.13	3457.94	0	2384.91	0	26997.50	0	1597.33	0
200	9.51	3548.79	0	2657.67	0	70103.10	0	702.53	0
300	11.85	4656.68	0	2980.62	0	84302.91	0	2489.43	0
400	15.61	5498.11	0	3250.74	0	79324.66	0	5538.59	0
500	16.35	3954.29	0	3372.22	0.4	76532.03	0	6772.96	0

Tabela 2.4: Resultados dos Testes da Aplicação - *Login + Registo + Consulta*

Com base nos tempos de resposta representados, é possível comprovar que o peso computacional de uma inserção de um utilizador é muito superior ao peso de realizar a autenticação na plataforma. A partir dos 500 utilizadores, a plataforma entra em sobrecarga, pelo que a diminuição do tempo de resposta da obtenção da página inicial e do registo de utilizadores traduz-se na apresentação da página de erro, ao invés das páginas pretendidas (Figuras 2.7 e 2.13).



Figura 2.13: Erro de tentativa de registo de utilizador

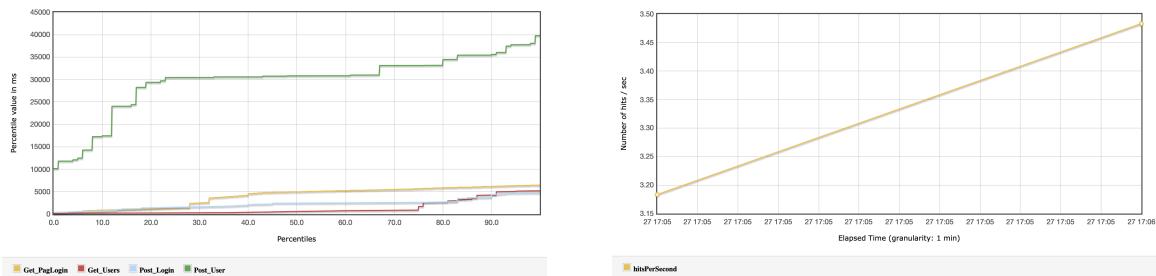


Figura 2.14: Resultados Gráficos - *login + registo + consulta* (100 threads)

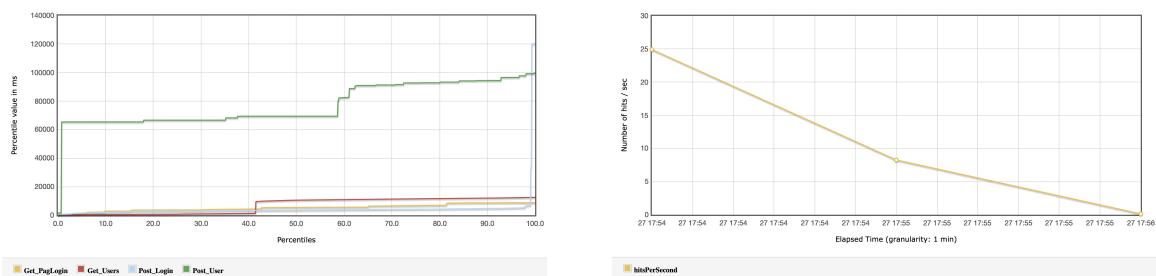


Figura 2.15: Resultados Gráficos - *login + registo + consulta* (500 threads)

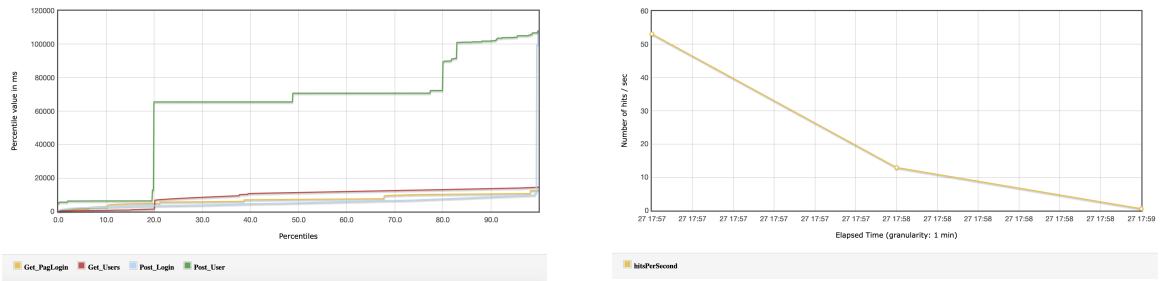


Figura 2.16: Resultados Gráficos - *login + registo + consulta* (1000 threads)

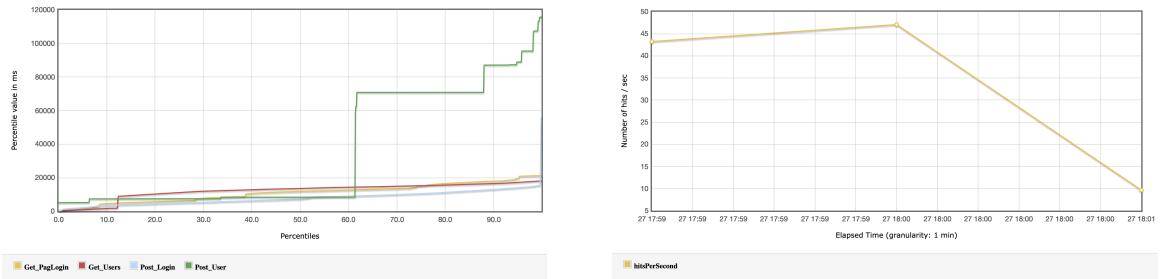


Figura 2.17: Resultados Gráficos - *login + registo + consulta* (1500 threads)

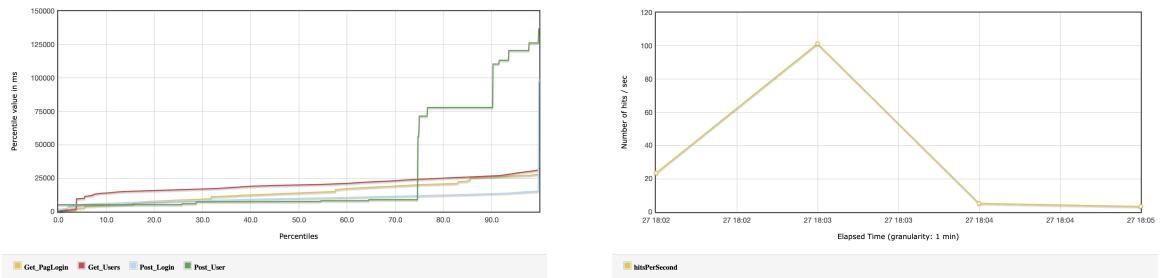


Figura 2.18: Resultados Gráficos - *login + registo + consulta* (2000 threads)

2.4 Disponibilidade da Aplicação

Entende-se por **Sistema de Alta Disponibilidade** (*High Availability*), o sistema informático que é resistente a falhas de *hardware*, *software* e energia. Para reduzir a probabilidade de interrupções no serviço, é necessária a introdução de redundância, assegurando a redução de pontos críticos de falha no sistema.

No que diz respeito à plataforma *Wiki.js*, verifica-se a existência de diversos pontos críticos de falha, que diminuem consideravelmente a disponibilidade da mesma (como comprovado pelos testes indicados no capítulo prévio).

Para ultrapassar esta dificuldade, decidiu-se implementar um sistema com componentes de alta disponibilidade (*cluster*), que permite a agregação de diversos nós independentes, como se apenas de um nó se tratasse. Esta implementação permite a introdução de balanceamento de carga e alta disponibilidade na aplicação [4].

Capítulo 3

Arquitetura Proposta

3.1 Desenho da Arquitetura

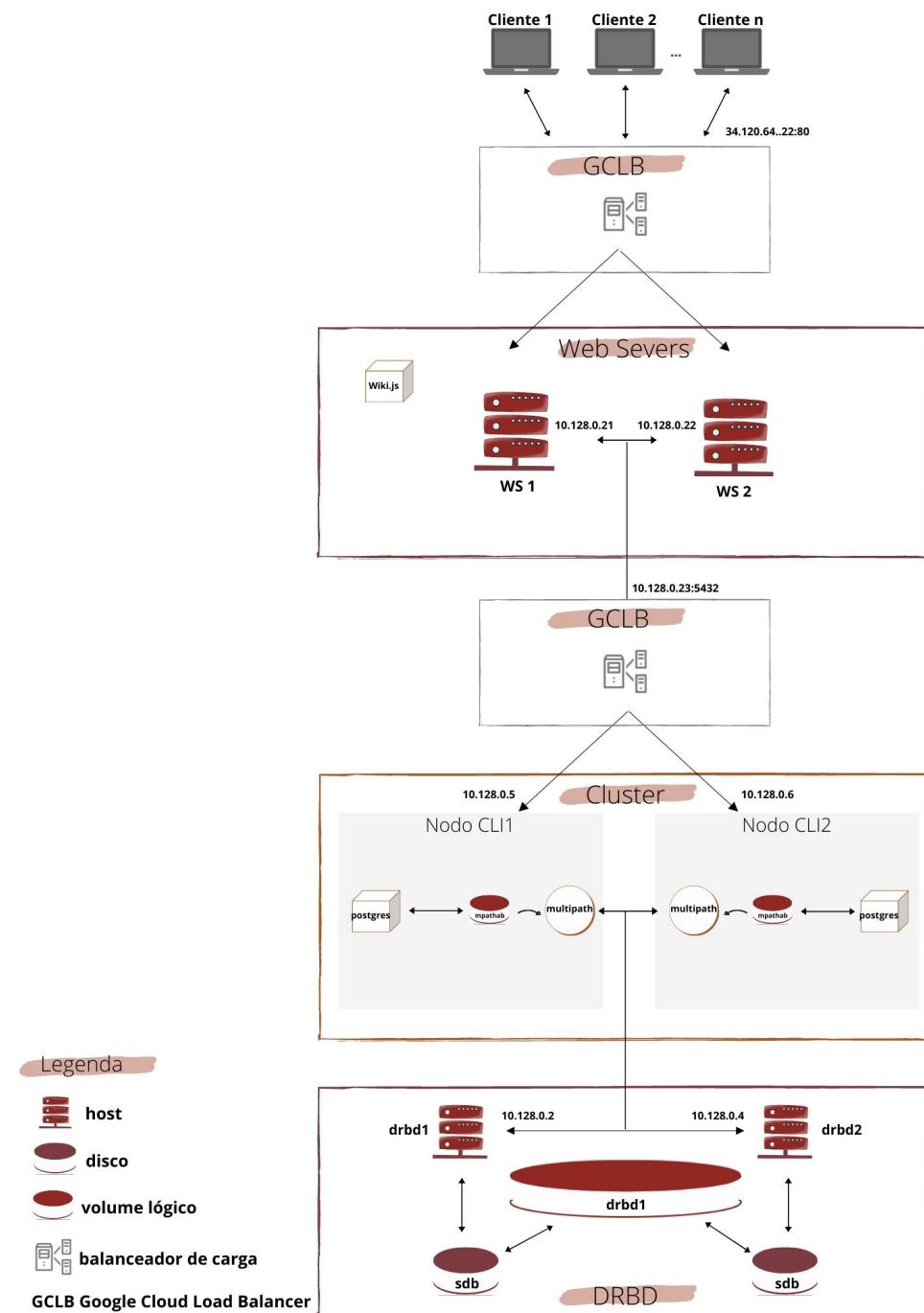


Figura 3.1: Desenho da Arquitetura Implementada

3.2 Descrição da Arquitetura

A arquitetura implementada encontra-se dividida em quatro grupos: o **DRBD**, o *Cluster*, os *Web Servers* e os平衡adores de carga.

3.2.1 DRBD

O primeiro passo para a concretização da arquitetura proposta esteve na implementação de um DRBD (*Distributed Replicated Block Device*). Foi utilizado o protocolo **iSCSI** (*Internet Small Computer System Interface*) para a sua configuração. Trata-se de um protocolo da camada de transporte que funciona sobre o protocolo de *TCP* e permite que o comando *SCSI* seja enviado ponta-a-ponta através de redes locais (LANs), redes de longa distância (WANs) ou da Internet. Permite aceder a recursos remotos como se se tratassem de recursos locais [5]. Para esta arquitetura, foi definido como *iSCSI Client* os nodos do *cluster*, e como *iSCSI Target* os servidores drbd1 e drbd2. Foi, ainda, utilizada a característica *multipath* do *iSCSI*, que configura várias rotas entre um servidor e os seus dispositivos de armazenamento, de forma a manter uma conexão constante e equilibrar a carga de tráfego [6].

3.2.2 Cluster

Um **cluster de alta disponibilidade**, ou *cluster de failover*, usa vários sistemas que já estão instalados, configurados e conectados, de modo que, se um sistema falhar, outro possa ser perfeitamente aproveitado para manter a disponibilidade do serviço ou aplicação [7]. Nesta arquitetura, o *cluster* é constituído por duas máquinas que distribuem o serviço da base de dados *PostgreSQL*.

3.2.3 Web Servers

Os *web servers* correspondem às máquinas onde se encontra a plataforma *Wiki.js*. Foram implementadas duas máquinas para estes servidores.

3.2.4 Balanceadores de Carga

De forma a balancear o tráfego pela rede e tolerar uma falta dos *web servers* e do *cluster*, foram configurados dois balanceadores de carga.

3.3 Resolução dos Pontos Críticos

A resolução dos pontos críticos é crucial para garantir ultrapassar erros num sistema de alta disponibilidade. Consequentemente, foram criadas soluções que permitissem a eliminação dos pontos críticos do sistema.

3.3.1 Web Servers

De modo a reduzir a possibilidade de falha, foram implementados dois servidores com os serviços *web*, permitindo o funcionamento de, pelo menos, uma máquina em cada momento. Desta forma, é ainda garantida a distribuição de carga.

3.3.2 Cluster

No *cluster* encontram-se os serviços de base de dados, imprescindíveis para o correto funcionamento do sistema, tendo sido adicionados dois nodos. Recorrendo à funcionalidade *multipath* do protocolo **iSCSI**, foi possível garantir o correto funcionamento do sistema, mesmo que um dos nodos do *cluster* falhe. Isto deve-se à configuração de *failover*, fornecida pelo *multipath*. Assim, no caso de falha de um caminho ou de qualquer um dos seus componentes, o servidor seleciona outro caminho disponível. Para

além disso, o balanceamento de carga entre os vários nodos distribui as cargas de armazenamento por vários caminhos, reduzindo potenciais *bottlenecks* [6].

3.3.3 DRBD

Por fim, foi também replicado o servidor do DRBD, responsável pelos dados dos serviços do *cluster*, de modo a garantir o funcionamento dos serviços, na eventualidade de ocorrência de falta de uma das máquinas. Em caso de *failover* de um DRBD, o sistema mantém o seu funcionamento, uma vez que ambas as máquinas são DRBD **primários**.

A replicação de todos os serviços do sistema permite garantir o correto funcionamento do mesmo, partindo do princípio que não ocorrem falhas no balanceador principal. Por conseguinte, todos os serviços estarão operacionais, mesmo que uma máquina de cada serviço falhe, visto que as respetivas réplicas serão capazes de responder aos pedidos. Desta forma, a plataforma deixa de funcionar unicamente quando as duas máquinas de um serviço falharem, ou quando o balanceador principal não estiver disponível.

3.4 Política de Balanceamento

De forma a otimizar a utilização de recursos, maximizar o desempenho, minimizar o tempo de resposta e evitar sobrecarga na rede, foi necessário recorrer a métodos de balanceamento de carga disponibilizados pelo *Google Cloud Platform*. Para este fim, foram criados dois serviços de rede de balanceamento de carga:

1. Balanceador de carga HTTP - Aplicado para distribuir os pedidos pelos dois servidores *web*;
2. Balanceador de carga TCP - Aplicado para distribuir a carga entre os dois nodos do *cluster*.

3.5 Desempenho da Aplicação: Testes de Carga

Foram realizados os mesmos testes de carga para avaliar o desempenho da aplicação após implementação da arquitetura proposta.

3.5.1 Login

O primeiro teste consiste em entrar na página inicial do *Wiki.js* e, posteriormente, efetuar o *login* na aplicação. Para este efeito, são testados dos seguintes pedidos:

1. GET /login
2. POST /graphql

Body data:

```
{  
    "query": "mutation($username: String!, $password: String!, $strategy: String!) {\\n        authentication {\\n            login(username: $username, password: $password, strategy: $strategy) {\\n                responseResult {\\n                    succeeded \\n                    errorCode \\n                    slug \\n                    message \\n                }\\n                jwt \\n                mustChangePwd \\n                mustProvideTFA \\n                mustSetupTFA \\n                continuationToken \\n                redirect \\n                tfaQRImage \\n            }\\n        }\\n    }\\n}  
    \"variables\": {  
        \"username\" : \"icd2021@uminho.pt\",  
        \"password\" : \"icdwikijs\",  
        \"strategy\" : \"local\"  
    }  
}
```

Este teste foi realizado para o mesmo número de *threads* que o respetivo teste na aplicação base, permitindo uma comparação mais consistente. No entanto, de forma a testar as capacidades da arquitetura implementada, considerou-se *loop-count* = 100. Na Tabela 3.1, estão representados os valores de **tempo de resposta**, **throughput** e percentagem de **erro** dos respetivos pedidos.

Threads	Throughput (min)	GET		POST	
		Tempo de Resposta (ms)	Erro (%)	Tempo de Resposta (ms)	Erro (%)
100	502.31	150.34	0	151.47	0.1
500	982.18	439.37	0	438.91	0.4
1000	841.00	1085.53	0.01	1062.37	1.32
1500	798.62	1767.64	0.41	1682.42	1.97
2000	909.00	1975.06	18.81	1876.58	20.42
2000	742.57	2544.39	1.15	2472.01	3.26

Tabela 3.1: Resultados dos Testes com Arquitetura Implementada - *Login*

Nesta avaliação, verifica-se que o primeiro teste realizado com 2000 *threads* obteve valores muito elevados de percentagens de erro. Estes valores devem-se à falha de um nodo CLI do *cluster*, pelo que foi necessária a migração dos serviços. O segundo teste resulta do correto funcionamento de toda a infraestrutura.

Comparando os resultados deste teste com o realizado na aplicação base, observa-se que os valores do débito são muito superiores, e os tempos de resposta menores, **com resultados mais de duas vezes melhores**.

Similarmente ao sucedido na aplicação base, vários pedidos têm como resposta HTTP, a mensagem de erro "*Too many requests*" (Figura 2.7). Estes erros devem-se a mecanismos de segurança por parte da aplicação base, não permitindo muitos pedidos por segundo.

O aumento da percentagem de erro na implementação da arquitetura, comparativamente com a aplicação base, deve-se ao *load balancer* a que estão ligados os *webservers*. Quando este deteta um erro vindo dos *webservers*, interpreta o erro como uma falha.

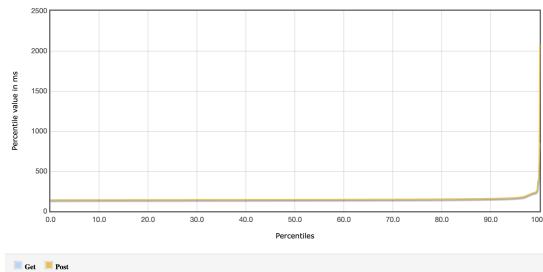


Figura 3.2: Resultados Gráficos com Arquitetura - *login* (100 threads)

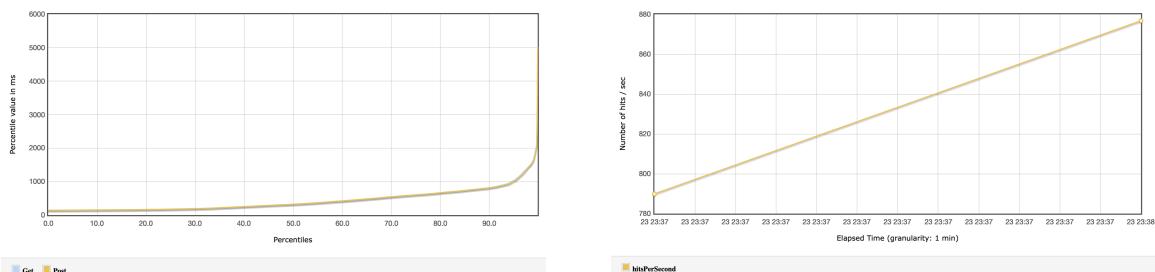


Figura 3.3: Resultados Gráficos com Arquitetura - *login* (500 threads)

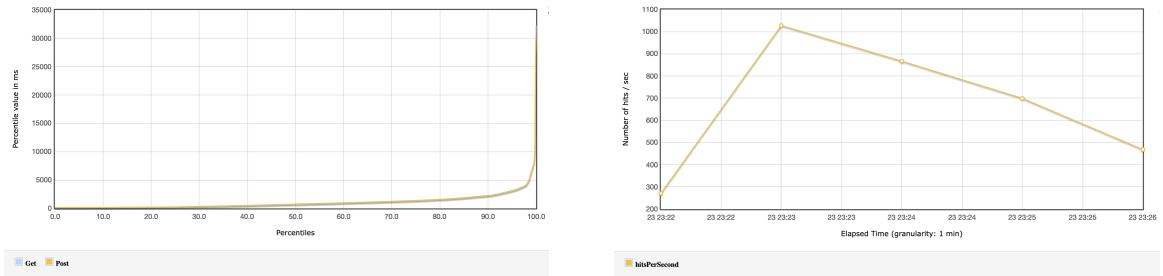


Figura 3.4: Resultados Gráficos com Arquitetura - *login* (1000 threads)

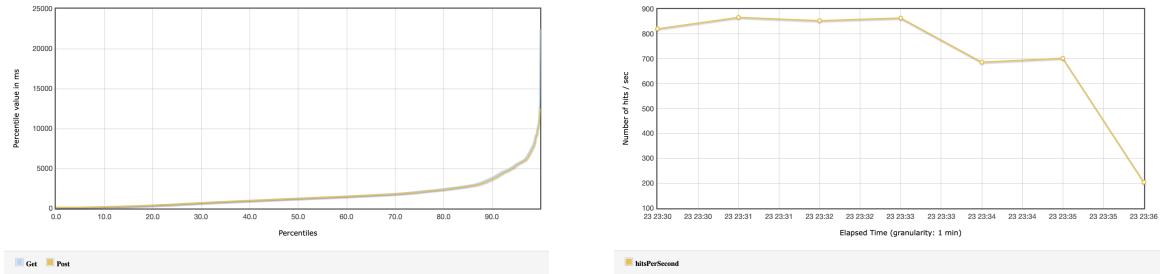


Figura 3.5: Resultados Gráficos com Arquitetura - *login* (1500 threads)

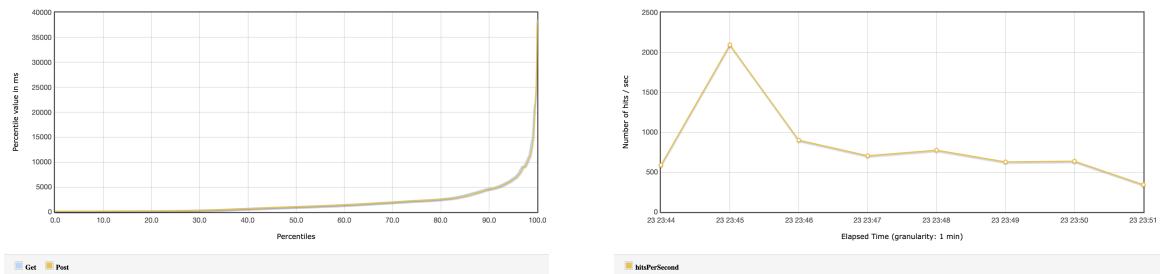


Figura 3.6: Resultados Gráficos com Arquitetura - *login* (falha de nodo CLI)

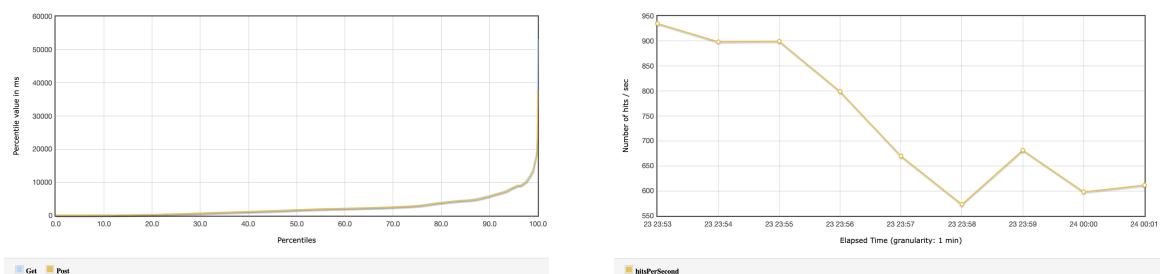


Figura 3.7: Resultados Gráficos com Arquitetura - *login* (2000 threads)

3.5.2 Login + Consulta das páginas criadas

No segundo teste, foi efetuado o *login* do administrador, onde este visualiza, posteriormente, a lista de todas as páginas *wiki* criadas.

1. GET /login
2. POST /graphql

Body data:

{

```

"query": "mutation($username: String!, $password: String!, $strategy: String!) {\n  authentication {\n    login(username: $username, password: $password, strategy: $strategy) {\n      responseResult {\n        succeeded\n        errorCode\n        slug\n        message\n      }\n      jwt\n      mustChangePwd\n      mustProvideTFA\n      mustSetupTFA\n      continuationToken\n      redirect\n      tfaQRImage\n    }\n  }\n}\n\nvariables: {\n  \"username\" : \"icd2021@uminho.pt\", \n  \"password\" : \"icdwikijs\", \n  \"strategy\" : \"local\"\n}\n"
}

```

3. GET /p/pages

Este teste foi realizado sob as mesmas condições que o teste anterior, com um *loop count* de 100. Verifica-se, na Tabela 3.2, os resultados obtidos neste teste.

Threads	Throughput (min)	GET		POST		GET	
		Tempo de Resposta (ms)	Erro (%)	Tempo de Resposta (ms)	Erro (%)	Tempo de Resposta (ms)	Erro (%)
100	170.84	164.09	0	165.17	0	162.74	0
500	995.91	436.83	0	438.86	0.3	444.02	0
1000	775.32	1171.53	0.02	1147.59	1.51	1196.77	0.02
1500	741.69	1555.08	0.18	967.51	1.15	968.89	0.01
2000	726.16	2624.89	1.59	2574.22	3.73	2643.63	1.63

Tabela 3.2: Resultados dos Testes com Arquitetura Implementada - *Login + Consulta*

Com base nos resultados obtidos, e realizando uma comparação com os resultados obtidos na Figura 2.3, é evidente a melhoria considerável do débito e tempos de resposta. Tal como nos testes anteriores, o aumento da percentagem de erro nos testes deve-se às limitações, por questões de segurança, impostas pela *Wiki.js* para o número de pedidos enviados por unidade de tempo, e à identificação do erro por parte dos *load balancers*.

3.5.3 Login + Registo de utilizador + Consulta de utilizadores

Para este teste, o administrador efetua a sua autenticação na plataforma e, em seguida, insere um novo utilizador. Finalmente, consulta todos os utilizadores.

1. GET /login
2. POST /graphql

Body data:

{

```

"query": "mutation($username: String!, $password: String!, $strategy: String!) {\n  authentication {\n    login(username: $username, password: $password, strategy: $strategy) {\n      responseResult {\n        succeeded\n        errorCode\n        slug\n        message\n      }\n      jwt\n      mustChangePwd\n      mustProvideTFA\n      mustSetupTFA\n      continuationToken\n      redirect\n      tfaQRImage\n    }\n  }\n}\n\nvariables: {\n  \"username\" : \"icd2021@uminho.pt\", \n  \"password\" : \"icdwikijs\", \n  \"strategy\" : \"local\"\n}"
}

```

```

    }
}

```

3. POST /graphql

Body data:

```

{
  "query": "mutation ($providerKey: String!, $email: String!, $name: String!, $passwordRaw: String,
$groups: [Int]!, $mustChangePassword: Boolean, $sendWelcomeEmail: Boolean) { \n users { \n
create(providerKey: $providerKey, email: $email, name: $name, passwordRaw: $passwordRaw, groups:
$groups, mustChangePassword: $mustChangePassword, sendWelcomeEmail: $sendWelcomeEmail) { \n
responseResult { \n succeeded \n errorCode \n slug \n message \n } } } }",
  "variables": {
    "providerKey" : "local",
    "email" : "JM${__time()}_${__threadNum()}@teste.pt",
    "name" : "JM${__time()}_${__threadNum()}",
    "passwordRaw" : "testeteste",
    "groups" : [3],
    "mustChangePassword" : false,
    "sendWelcomeEmail" : false
  }
}

```

4. GET /a/users

Devido à grande exigência computacional deste teste, foi alterado o número de *threads* utilizado para simular os acessos concorrentes de utilizadores à aplicação. Foi, ainda, ajustado o *loop-count* para 1.

Threads	Throughput (min)	GET		POST		POST		GET	
		Tempo de Resposta (ms)	Erro (%)						
100	16.73	1405.17	0	1890.76	1	8106.74	3	1744.25	0
200	17.80	2745.71	0	3045.62	2	22621.04	5.50	2695.77	0.5
300	20.11	3308.73	0	4115.98	2.33	37124.05	7	1983.55	0
400	20.21	4678.65	0.25	5331.11	5	51211.98	8	2796.86	0.25
500	20.85	4135.82	0.8	5846.02	7	70193.02	16.20	1625.93	0

Tabela 3.3: Resultados dos Testes com Arquitetura Implementada - *Login + Registo + Consulta*

Como expectável, os resultados obtidos apontam para melhores valores de débito e, em geral, melhores tempos de resposta. Verifica-se um aumento do tempo de resposta na autenticação dos utilizadores e um acréscimo dos valores de percentagem de erro devido à sobrecarga dos *web servers*, provocando a sua falha. Foram realizados testes com a introdução de um novo *web server*, verificando-se uma ligeira melhoria, visto que só falha um *web server* a partir de 700 utilizadores. Para 750 utilizadores verificou-se a falha de todos os *web servers*.

A elevada percentagem de erro deve-se, ainda, ao reconhecimento do erro por parte do *load balancer* (Figura 3.8), contrariamente ao verificado na aplicação base (Figura 2.13).

```

<html><head>
<meta http-equiv="content-type" content="text/html; charset=utf-8">
<title>502 Server Error</title>
</head>
<body style="background-color: #000000; color: #ffffff;">
<h1>Error: Server Error</h1>
<h2>The server encountered a temporary error and could not complete your request.<p>Please try again in 30 seconds.</h2>
</body></html>

```

Figura 3.8: Erro de tentativa de registo de utilizadores

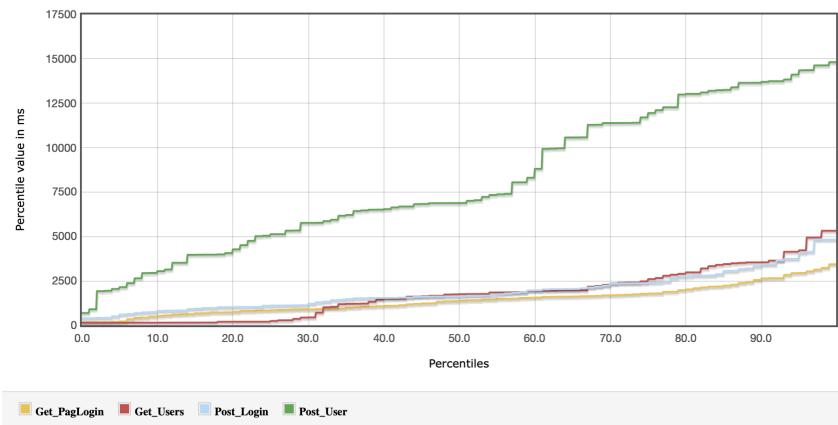


Figura 3.9: Resultados Gráficos com Arquitetura - *login + registo + consulta* (100 threads)

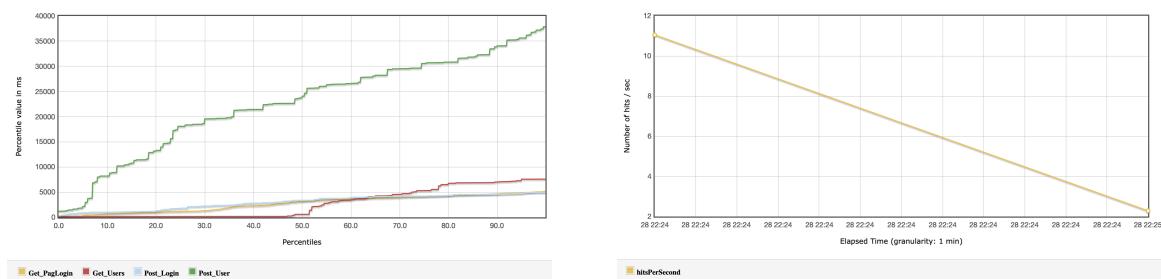


Figura 3.10: Resultados Gráficos - *login + registo + consulta* (200 threads)

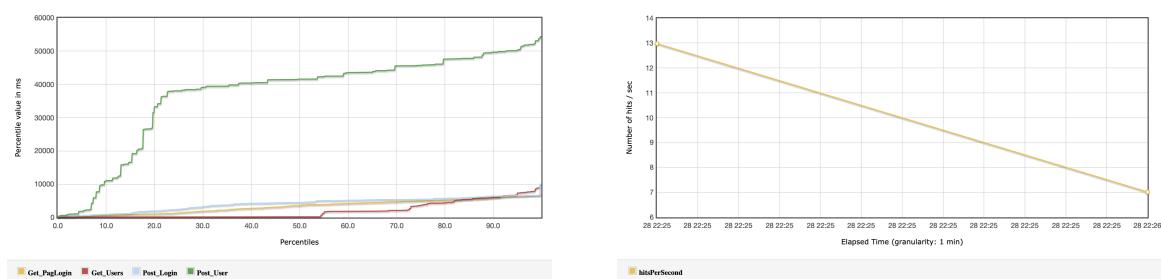


Figura 3.11: Resultados Gráficos - *login + registo + consulta* (300 threads)

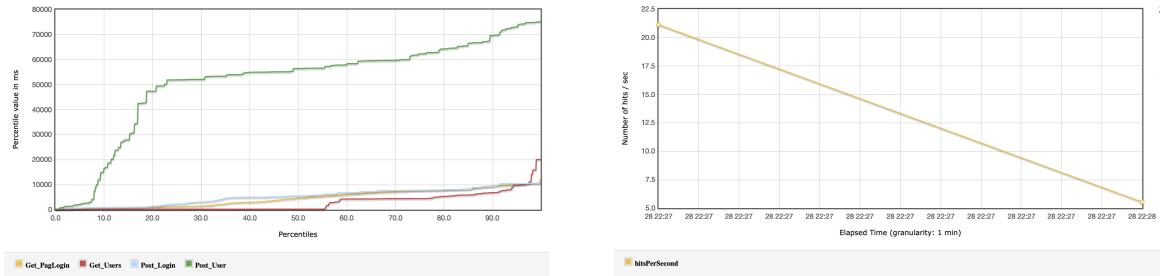


Figura 3.12: Resultados Gráficos - *login + registo + consulta* (400 threads)

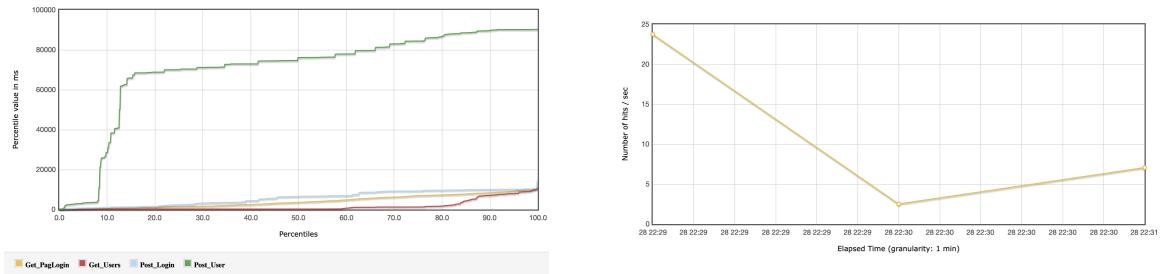


Figura 3.13: Resultados Gráficos - *login + registo + consulta* (500 threads)

3.6 Disponibilidade da Aplicação

Após a implementação da arquitetura, é expectável que haja acréscimo da disponibilidade da aplicação (comprovado pelos testes realizados), bem como uma melhoria no desempenho da mesma. Com esta arquitectura é possível garantir a disponibilidade da aplicação em todas as ocasiões, exceto:

- Se o balanceador de carga inicial falhar. Neste caso, toda a aplicação falha, uma vez que não será possível distribuir a carga pelos diferentes *web servers*;
- Se ambos os *web servers* falharem. Neste caso, perde-se a ligação à aplicação;
- Se falhar o segundo balanceador. A falha do segundo balanceador impede o reencaminhamento de tráfego para os nodos do *cluster*;
- Se falharem todos os nodos do *cluster*. No caso de ambos os nodos que constituem o *cluster* falharem, é perdido o acesso aos serviços de bases de dados.

Desta forma, pode-se afirmar que a arquitetura implementada garante alta disponibilidade.

Capítulo 4

Discussão dos Resultados Obtidos

Com a realização deste trabalho, foi possível consolidar os conhecimentos obtidos na Unidade Curricular de Infraestrutura de Centros de Dados, aplicando-os para ultrapassar obstáculos e desafios que foram sendo apresentados ao longo do percurso.

Numa fase inicial, foi feita a instalação do *Wiki.js* nos servidores da *Google Cloud* e analisada a arquitetura da aplicação. Foram identificados os pontos críticos mais suscetíveis a falhas durante o seu funcionamento, sendo eles o *backend*, a base de dados e o *frontend*.

Posteriormente, foi testado o desempenho da aplicação com a sua arquitetura base. Nesta fase, foram determinados os limites da aplicação quando submetida a testes de carga, quantificando os resultados com a ferramenta *Apache JMeter*. Foram realizados testes com diferentes tipos de pedidos e páginas a serem solicitadas, com o intuito de analisar o comportamento do sistema em diferentes situações.

Numa segunda fase, o próximo passo foi melhorar a aplicação. Para tal, criaram-se soluções que permitissem eliminar os pontos críticos do sistema; foram implementados mecanismos para aumentar o desempenho da aplicação através do balanceador de carga usando ferramentas disponíveis na *Google Cloud*; foi também aplicado um mecanismo de *failover*, recorrendo ao *iSCSI*, para salvaguardar de eventuais falhas de alguns dos componentes da arquitetura, garantindo assim a alta disponibilidade da aplicação.

Por fim, foram realizados novamente os testes de carga para testar os efeitos das alterações. Verificou-se que a aplicação, de facto, obteve um melhoramento no seu desempenho.

Tendo em conta os aspetos mencionados e os resultados obtidos na realização deste trabalho, consideramos que conseguimos atingir os objetivos propostos.

Bibliografia

- [1] *What is a Single Point of Failure?* Jan. de 2019. URL: https://www.computerhope.com/jargon/s_spof.htm (acedido em 21/12/2020) (ver p. 5).
- [2] *Apache JMeter.* 2020. URL: <https://jmeter.apache.org/> (acedido em 22/12/2020) (ver p. 5).
- [3] *Access Tokens.* 2021. URL: <https://auth0.com/docs/tokens/access-tokens> (acedido em 31/12/2020) (ver p. 8).
- [4] *Sistema de Alta Disponibilidade.* 2020. URL: https://pt.wikipedia.org/wiki/Sistema_de_alta_disponibilidade (acedido em 22/12/2020) (ver p. 12).
- [5] *What is iSCSI and How Does it Work?* 2020. URL: <https://searchstorage.techtarget.com/definition/iSCSI> (acedido em 31/12/2020) (ver p. 14).
- [6] *Cisco Nexus 1000V System Management Configuration Guide, Release 4.0(4)SV1(3) - Configuring iSCSI Multipath [Cisco Nexus 1000V Switch for VMware vSphere].* Jun. de 2016. URL: https://www.cisco.com/c/en/us/td/docs/switches/datacenter/nexus1000/sw/4_0_4_s_v_1_3/system_management/configuration/guide/n1000v_system/n1000v_system_13iscsi.html (acedido em 21/12/2020) (ver pp. 14, 15).
- [7] *High Availability Cluster.* 2020. URL: <https://www.sciencedirect.com/topics/computer-science/high-availability-cluster> (acedido em 31/12/2020) (ver p. 14).