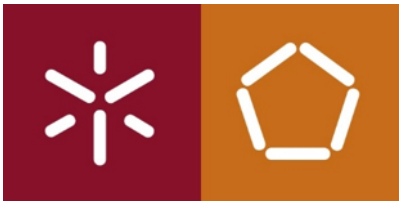


Universidade do Minho

Infraestrutura de Centros de Dados

(Perfil de Engenharia de Aplicações)

Trabalho Prático

Relatório de Desenvolvimento

Mestrado Integrado em Engenharia Informática

Desenvolvido por G10:
Filipa Alves dos Santos, a83631
Hugo André Coelho Cardoso, a85006
João da Cunha e Costa, a84775
Luís Miguel Arieira Ramos, a83930
Válter Ferreira Picas Carvalho, a84464

14 de janeiro de 2021

Resumo

O trabalho prático é essencialmente o *deployment* da plataforma Wiki.js de forma escalável e de elevada disponibilidade e desempenho, utilizando para esse efeito a **GCP** (*Google Cloud Platform*).

São abordados os **SPOFs**, *"Single Point of Failure"*, e são expostas possíveis soluções para estes problemas, que transformam uma instância da **Wiki.js** em um *cluster* de máquinas com funções específicas.

Para a correção destes pontos de falha e de modo a permitir elevada escalabilidade, são utilizadas ferramentas expostas nas aulas ao longo do semestre, nomeadamente **ISCSI**, **DRBD**, **Heartbeat**, **Corosync** e **Pacemaker**. No entanto, não foi possível utilizar as ferramentas **IPVS** e **Keepalive** abordadas nas aulas, pelo que foi necessário proceder à utilização das ferramentas da *cloud* para substituir as suas funções.

Por fim, são testados os resultados do *deployment* utilizando o **JMeter**, de modo a estabelecer conclusões finais.

Conteúdo

Listagens

Lista de Figuras

1 Introdução

O principal objetivo da unidade curricular *Infraestrutura de Centros de Dados* é expor as abordagens necessárias para a elevada disponibilidade, escalabilidade e desempenho de uma aplicação.

Para isso ser possível, é primeiro necessário analisar a aplicação em detalhe e entender onde é que ela pode falhar e, com essa informação, construir um plano de salvaguarda para não a tornar indisponível caso um desses recursos falhe (identificação dos **SPOFs**, *Single Point of Failure*). É necessário, também, a utilização de balanceadores de carga para não se utilizar em demasia um recurso.

Após todo este processo, é necessário ainda realizar testes à plataforma para garantir que, de facto, tudo funciona com a disponibilidade pretendida, há tolerância a faltas e as cargas são distribuidas pelas várias máquinas no *cluster*.

Para este projeto em específico, é utilizada a plataforma Wiki.js, uma plataforma *open-source* que atua como modelo para a criação de páginas *wiki* para um projeto. As suas funcionalidades são, principalmente, a criação de páginas, autenticação de utilizadores e comentários.

Tendo estes conceitos cimentados, é dividido o projeto nas seguintes fases:

1. Análise da arquitetura da Wiki.js;
2. Identificação de **SPOFs**;
3. Implementação da solução de elevado desempenho e disponibilidade, assim como balanceamentos de carga;
4. Testes de desempenho à plataforma.

2 Análise Inicial

2.1 Arquitetura da Aplicação

A plataforma Wiki.js segue a arquitetura típica de um *web server*, pelo que é composta por 3 grandes componentes:

1. **Back-End**: contém um número enorme de serviços, desde autenticação do cliente até incorporação de motores de pesquisa da **AWS**. No entanto, para o propósito do projeto, serão apenas relevantes três deles: o servidor em **Node.js**, a API de acesso à camada de dados (**PostgreSQL**) e o *end-point* **GraphQL**, a ser analisado mais à frente.

2. **Front-End**: contém toda a interação com o utilizador, é a parte em **HTML** e **JavaScript**, gerados pelo servidor e renderizados no lado do cliente, que envia os pedidos **Restful** (neste caso, utiliza o **GraphQL**) ao servidor no *back-end*.

3. **Base de Dados**: a aplicação permite utilizar uma variedade de bases de dados, no entanto, será utilizado o **PostgreSQL**, uma vez que é o recomendado pelos desenvolvedores originais da aplicação, de modo a ser o mais rápida possível e confiável, para benefício do utilizador.

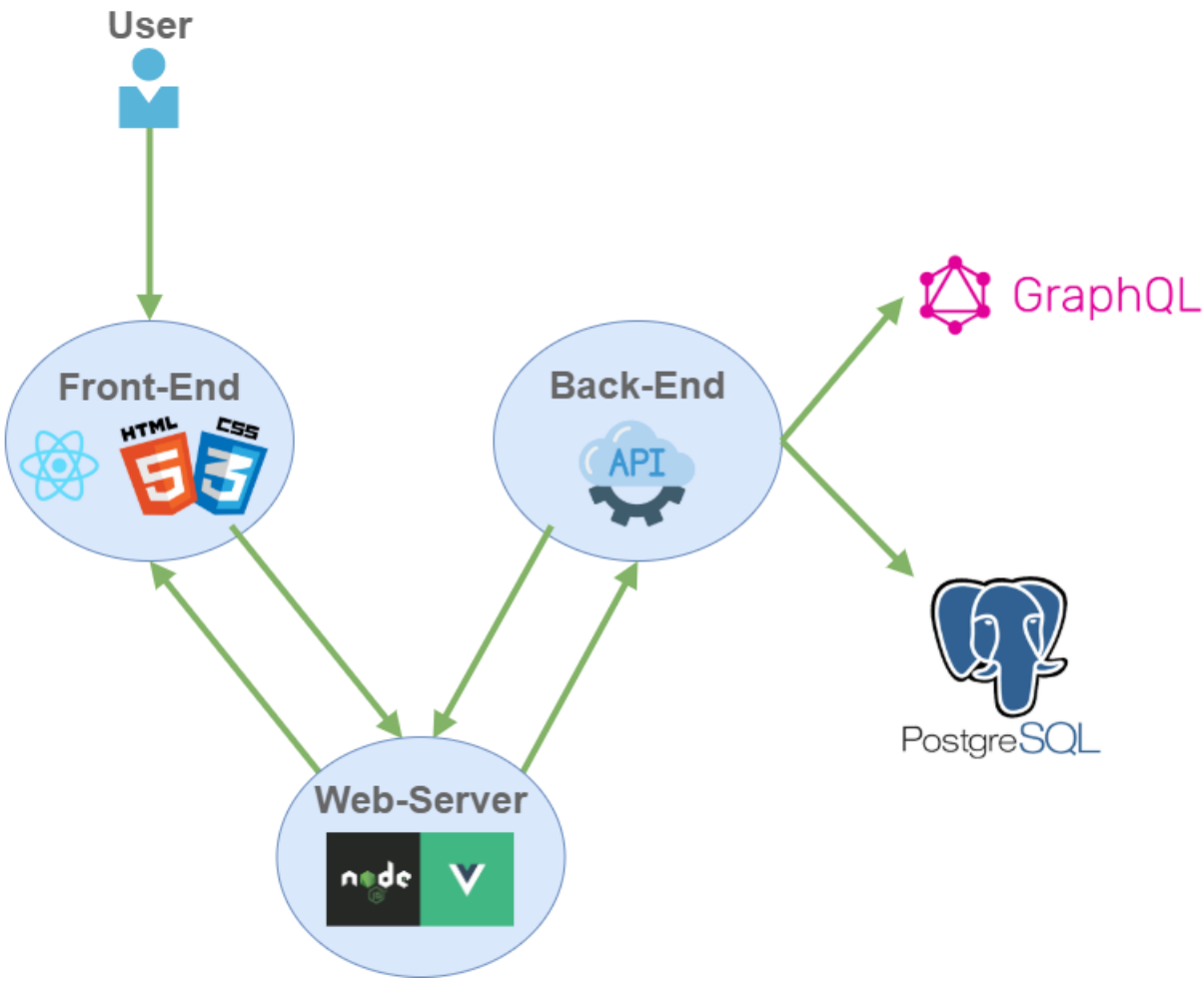


Figura 1: Arquitetura da plataforma Wiki.js (simplificada)

2.2 Virtual Machine (GCP)

O *deployment* inicial da Wiki.js foi realizada numa *Instância de Máquina Virtual da Google Cloud*, para testar todos os gargalos do sistema.

Durante o projeto, foi atribuído ao grupo um cupão de 50 dólares para utilizar ao critério do mesmo na GCP. Deste modo, o grupo decidiu ser moderadamente conservador na medida em que tentou ao máximo evitar desperdiçar o crédito porque uma vez usado, não teria mais. Portanto, acabou por utilizar uma máquina **e2-small** para todas as instâncias no projeto, uma vez que era o compromisso perfeito de perfomance e preço.

Para além disso, foram utilizados os servidores americanos devido ao seu baixo custo e, uma vez que era uma plataforma não orientada a consumidores reais, não faria sentido colocar multi-zonas ou colocar na Europa com preço acrescido.

As suas características são as seguintes:

- **Localização**: us-central1-a
- **Tipo**: e2-small
- **CPU**: 2 vCPU @ 2.0 GHz, 2.7 GHz Turbo
- **Memória**: 2 GB
- **Armazenamento**: HDD 20 GB
- **OS**: CentOS 8

2.2.1 Instalação

Para a instalação da aplicação numa só instância, o processo, de forma geral, é relativamente simples.

Inicialmente, é necessário configurar a base de dados, porque é um serviço externo à aplicação e tem de ser feito pelo administrador do sistema, que é o grupo que realizou o projeto, neste caso. Foi criado, no PostgreSQL, uma base de dados única ("*wiki*"), com um utilizador específico para aceder a essas tabelas na base de dados ("*wikijs*") com uma palavra-chave de acesso ("*wiki1234*").

Para o servidor Node.js conseguir conectar-se à base de dados, é necessário mudar o ficheiro "*config.yml*"e colocar lá a informação criada no passo anterior, nomeadamente o nome de utilizador, palavra-chave e a base de dados.

Após isso basta correr o servidor numa porta arbitrária que, no caso do grupo, foi a porta **3000**, que é a que vem por defeito, através do Node.js.

Apesar de já estar operacional, isso implicava colocar a porta explicitamente (xx.xx.xx.xx:3000) nos pedidos HTTP do *browser* do utilizador, o que não facilita tanto para testes como facilidade de acesso. Para isso, como a porta **80** não está a ser usada na máquina virtual, podemos usá-la como **proxy** de pedidos HTTP por browser (porta utilizada por defeito em HTTP). Para esse efeito, foi utilizado o **NGINX**, que atua por defeito na porta 80. Para funcionar como *proxy*, é mudado apenas um único campo nas suas definições de roteamento ("*nginx.conf*"):

```
location / {
     proxy_pass http://127.0.0.1:3000/;
}
```

Ou seja, realiza o roteamento de todo o tráfego na porta 80 para a porta 3000, que é o desejado. Agora já é possível aceder ao servidor sem indicar explicitamente a porta a utilizar, através do IPv4 da máquina virtual.

2.3 SPOFs

Um **SPOF**, *Single Point of Failure*, é essencialmente uma parte do sistema que, caso falhe, leva a que todo o sistema colapse e pare de funcionar.

No caso da Wiki.js, são detetados SPOFs nos vários níveis da plataforma, nomeadamente:

1. **Web-Server**: a Wiki.js expõe ao cliente serviços através duma interface gráfica (*front-end*). Se este serviço pára de funcionar, não há comunicação com todos os restantes serviços na *back-end* e, portanto, o cliente fica sem interação possível.

2. **Serviços Aplicacionais**: entre muitos outros serviços na *back-end*, a plataforma acede à API de persistência de dados usando Postgres, pelo que é necessário ter o seu *daemon* sempre ativo (*postgresql*), uma vez que é uma aplicação externa à Wiki.js. Caso esta aplicação, por algum motivo, deixe de funcionar, a Wiki.js deixa de ter persistência de dados, levando a que o sistema pare por completo.

3. **Armazenamento**: para além do *daemon* de Postgres, se um dos setores de armazenamento (no caso do grupo, será um disco rígido) tem uma avaria, leva a que a camada de acesso à persistência, apesar de funcionar corretamente, não consiga efetivamente escrever as mudanças de estado em memória não-volátil, levando a que todo o sistema se torne inconsistente.

2.4 Testes de Desempenho

2.4.1 GraphQL

A plataforma expõe ao exterior uma API de dados em GraphQL, pelo que será utilizada para a realização de todos os testes que necessitem de operações CRUD no *back-end* da aplicação.

Por defeito, a própria aplicação evita utilizar o **Restful** tradicional, de modo a dar prioridade a operações de mutação ou de interrogação através do método **POST**.

Define-se como uma interrogação um pedido iniciado por *query*, que é utilizada essencialmente para realizar operações de busca de dados na API. As mutações, iniciadas por *mutation* no corpo da mensagem, são todas as operações que envolvam mudanças na base de dados, mais concretamente operações de escrita.

Todos os pacotes HTTP que tenham como propósito aceder a este *endpoint*, têm de ter no cabeçalho o seguinte campo e respetivo valor:

- **Authorization**: Bearer API_KEY

Em que a API_KEY é gerada pelo administrador nas definições do servidor (ver `https://docs.requarks.io/dev/api`).

2.4.2 Método de Testes

Para proceder à execução dos testes, foi utilizada a aplicação Apache JMeter, que é uma ferramenta muito poderosa e permite manipular várias vertentes para executar testes, nomeadamente criação de *threads* para simular vários utilizadores, pedidos HTTP, geração automática de resultados, *benchmarking*, entre outras.

No entanto, os seus resultados dependem do computador utilizado, uma vez que o tempo para completar uma *thread* depende se o CPU suporta *Hyper-Threading* (paralelismo de execução simultânea de processos em um único núcleo) e do *clock speed*, para além de todas as outras características do processador e memória RAM, nomeadamente o total disponível para a **Heap da JVM** do JMeter.

Tendo este pequeno percalço em mente, foi usado um único computador para efeitos de testes com as seguintes características relevantes:

- **CPU**: i5-6600k, QuadCore @ 3.5-3.9 GHz (Sem Hyper-Threading)
- **RAM**: 16 GB DDR4 @ 3200 Mhz
- **JVM Heap**: 8 GB

Com o ambiente de testes preparado e com estes conceitos cimentados, foi dado início à fase de testes inicial à plataforma.

2.4.3 Teste 1: Página Inicial

Este teste serve essencialmente para descobrir quantos utilizadores, em simultâneo, a aplicação permite com uma taxa de erro baixa (inferior a 1%), num cenário de carga baixa. Será utilizada a requisição da página inicial, uma vez que é o teste mais simples que é possível fazer, que envolve apenas o envio do seguinte pacote HTTP:

- **GET** /

Os resultados obtidos foram os seguintes:

Threads	Average(ms)	Min(ms)	Max(ms)	Std. Dev.	Error %	Throughput
1000	1137	523	1612	259.5	0.00%	392.92731
5000	977	268	2824	492.42	0.36%	628.77264
10000	2465	268	21275	1700.85	10.02%	313.29302
25000	2552	256	25811	3333.37	21.08%	847.68751
50000	5706	1233	42184	7028.54	41.48%	830.63216

Figura 2: Resultados do Teste 1

O **Throughput** tende a aumentar com o aumento de *threads*, como seria de esperar, porque o computador envia mais pedidos num dado intervalo de tempo.

Analisando a coluna de **Error %**, vemos que para um número de clientes igual a **1000**, o sistema é totalmente responsivo (0.00 %). Passando para os **5000** clientes, notamos que o sistema começa a apresentar erros, apesar de bastante baixos (taxa inferior a 1%).

Quando à latência de resposta, para um número de clientes igual a **5000** ou menor, será de 1 segundo, que está dentro do expectável, tendo em conta que o servidor está nos Estados Unidos.

Porém, é quando se atinge o patamar de **10000** clientes que o sistema se torna bastante irresponsivo, tanto em latência como em taxa de erros, sendo 10 % demasiado insustentável para o sistema funcionar corretamente (o requisito é ter no mínimo 99% de disponibilidade, ou seja, 1% de taxa de erro no máximo).

A tabela seguinte indica, para uma determinada percentagem de disponinilidade do sistema, o *downtime* (tempo em que o servidor não responde) respetivo.

Availability %	Downtime per year[note 1]	Downtime per month	Downtime per week	Downtime per day
90% ("one nine")	**36.53 days**	**73.05 hours**	**16.80 hours**	**2.40 hours**
95% ("one and a half nines")	18.26 days	36.53 hours	8.40 hours	1.20 hours
97%	10.96 days	21.92 hours	5.04 hours	43.20 minutes
98%	7.31 days	14.61 hours	3.36 hours	28.80 minutes
99% ("two nines")	**3.65 days**	**7.31 hours**	**1.68 hours**	**14.40 minutes**
99.5% ("two and a half nines")	1.83 days	3.65 hours	50.40 minutes	7.20 minutes
99.8%	17.53 hours	87.66 minutes	20.16 minutes	2.88 minutes
99.9% ("three nines")	**8.77 hours**	**43.83 minutes**	**10.08 minutes**	**1.44 minutes**
99.95% ("three and a half nines")	4.38 hours	21.92 minutes	5.04 minutes	43.20 seconds
99.99% ("four nines")	**52.60 minutes**	**4.38 minutes**	**1.01 minutes**	**8.64 seconds**
99.995% ("four and a half nines")	26.30 minutes	2.19 minutes	30.24 seconds	4.32 seconds
99.999% ("five nines")	**5.26 minutes**	**26.30 seconds**	**6.05 seconds**	**864.00 milliseconds**
99.9999% ("six nines")	**31.56 seconds**	**2.63 seconds**	**604.80 milliseconds**	**86.40 milliseconds**
99.99999% ("seven nines")	**3.16 seconds**	**262.98 milliseconds**	**60.48 milliseconds**	**8.64 milliseconds**
99.999999% ("eight nines")	**315.58 milliseconds**	**26.30 milliseconds**	**6.05 milliseconds**	**864.00 microseconds**
99.9999999% ("nine nines")	**31.56 milliseconds**	**2.63 milliseconds**	**604.80 microseconds**	**86.40 microseconds**

Figura 3: Tabela com a percentagem de disponibilidade e o respetivo *downtime*[1]

Utilizando esta tabela e os resultados do teste, conclui-se então que o sistema só consegue responder, de forma a ter um grau de disponibilidade de 99%, para um número de clientes aproximadamente igual a **5000**.

2.4.4 Teste 2: Login

Este teste baseia-se, essencialmente, em simular um processo de login.

Neste teste já é necessário aceder à camada API exposta pelo GraphQL, pelo que é utilizado o *endpoint* por defeito, em */graphql*.

É composto pelos seguintes passos:

- **GET** /
- **POST** /graphql
  - Body Data (JSON): presente no anexo A.1

Os resultados obtidos foram os seguintes:

Threads	Average(ms)	Min(ms)	Max(ms)	Std. Dev.	Error %	Throughput
1000	2336	267	7495	1898.18	0.05%	230.89356
5000	4351	299	17309	3835.6	1.44%	468.84523
10000	7770	467	31683	7727.05	17.40%	603.17269
25000	8166	963	46467	8496.24	38.09%	903.54908
50000	5045	1201	90618	8055.85	49.18%	890.57504

Figura 4: Resultados do Teste 2

Novamente, o **Throughput** aumenta de forma expectável, pelo mesmo motivo do teste anterior.

Nota-se agora que, para um número de clientes igual a **5000**, a aplicação já se torna um pouco mais irresponsiva, comparada ao 1º teste, tanto em **tempo de resposta** (subiu aproximadamente 3 segundos) como para o **Error %** (subiu aproximadamente 1%).

Quanto mais se aumenta o número de clientes, mais irresponsiva se torna a aplicação.

[1]Imagem retirada de https://en.wikipedia.org/wiki/High_availability

2.4.5 Teste 3: Login + Criar Página

Para simular um utilizador a criar uma página após a sua autenticação, este teste executa as seguintes operações:

- **GET** /
- **POST** /graphql
  - Body Data (JSON): presente no anexo A.1
- **GET** /en/new-page
- **POST** /graphql
  - Body Data (JSON): presente no anexo A.2

Os resultados obtidos para este teste estão presentes na seguinte tabela:

Threads	Average(ms)	Min(ms)	Max(ms)	Std. Dev.	Error %	Throughput
1000	2888	268	8832	1941.6	2.30%	300.4582
5000	6674	264	27985	6493.15	22.13%	447.87818
10000	10776	521	74613	9317.66	35.86%	446.64465
25000	8492	1003	103098	11414.95	66.45%	877.54068
50000	5131	2201	120284	9192.86	82.07%	1401.58935

Figura 5: Resultados do Teste 3

Novamente, o **Throughput** aumenta à medida que o número de utilizadores cresce, pelo mesmo motivo do primeiro teste.

Agora, mesmo **1000** utilizadores levam a que o sistema se torne irresponsivo, tanto em **latência** como **disponibilidade**.

Comparativamente ao anterior, começa-se a verificar que quanto mais se aumenta a complexidade do pedido, mais rapidamente irresponsivo o servidor fica, uma vez que em **5000** utilizadores já ultrapassam a capacidade do servidor, caso todos realizem este processo de criação de página.

Agora, cada pedido demora, em média, 3 segundos a ser respondido, o que é demasiado tempo para um *web-server*, mesmo estando alojado nos Estados Unidos (já foi possível verificar que é possível obter latências de cerca de 1 segundo).

2.4.6 Teste 4: Login + Criar Página + Pesquisar Página

Este teste simula o utilizador ir pesquisar a página que acabou de criar, que permite testar um cenário realista de criação de páginas. Realiza os seguintes pedidos HTTP:

- **GET** /
- **POST** /graphql
  - Body Data (JSON): presente no anexo A.1
- **GET** /en/new-page
- **POST** /graphql
  - Body Data (JSON): presente no anexo A.2

- **GET** /
- **POST** /graphql
  - Body Data (JSON): presente no anexo A.3

Threads	Average(ms)	Min(ms)	Max(ms)	Std. Dev.	Error %	Throughput
1000	2573	266	8468	2149.64	1.40%	330.99796
5000	6730	265	34733	6889.21	19.56%	511.61363
10000	7917	761	135718	9758.86	46.60%	384.83741
25000	7964	2283	165826	11481.35	73.75%	702.53533
50000	5074	2747	113084	9462.82	85.85%	1578.34975

Figura 6: Resultados do Teste 4

De igual modo, o **Throughput** aumenta de forma expectável, pelo mesmo motivo do teste número 1.

Agora, acontece algo interessante relativamente à **latência**: não aumenta tanto como no teste anterior. Isto deve-se ao facto da **Error %** ser tão elevada que os pedidos levam mais *timeouts*, uma vez que o servidor não os processa.

Para um número de clientes igual a **1000**, vemos que é um cenário relativamente aceitável, para a disponibilidade pretendida tanto em **latência** (apesar de relativamente alta) como em **Error %**.

Servir um total de **5000** clientes nestas condições é totalmente insustentável, leva a terrível **performance**, **latência** e é altamente **indisponível**. A latência é, agora, cerca de 7 segundos caso este número de utilizadores seja atingido, o que é sinal de uma aplicação com disponibilidade baixa.

Cada vez mais se pode tirar a conclusão que ter **5000** utilizadores em simultâneo não é um cenário realista, pois apenas toma em consideração o caso mais simples de utilização do servidor: servir páginas iniciais ao utilizador.

2.4.7 Teste 5: Login + Criar Página + Pesquisar Página + Criar Comentário

Para este último teste, o grupo decidiu criar uma carga acrescida no servidor, que simula a criação de um comentário logo após o utilizador criar uma página.

São, então, realizados os seguintes pedidos HTTP:

- **GET** /
- **POST** /graphql
  - Body Data (JSON): presente no anexo A.1
- **GET** /en/new-page
- **POST** /graphql
  - Body Data (JSON): presente no anexo A.2
- **GET** /
- **POST** /graphql
  - Body Data (JSON): presente no anexo A.3

- **GET** /en/new-page
- **POST** /graphql
  - Body Data (JSON): presente no anexo A.4

Estes foram os resultados obtidos:

Threads	Average(ms)	Min(ms)	Max(ms)	Std. Dev.	Error %	Throughput
1000	2642	266	9174	2210.62	0.61%	348.38653
5000	5058	311	48478	6896.08	41.80%	564.66868
10000	8393	667	109195	10315.66	54.63%	442.8698
25000	7912	1941	144106	10724.96	77.96%	796.85085
50000	7413	2344	162407	11354.11	86.50%	1132.64473

Figura 7: Resultados do Teste 5

Tal como em todos os outros testes, o **Throughput** aumenta de forma expectável, pelo mesmo motivo do teste número 1.

Acontece exatamente a mesma coisa à **latência** que no teste anterior: não aumenta tanto como seria esperado. Isto deve-se ao facto da **Error %** ser tão elevada que os pedidos levam mais *timeouts*, uma vez que o servidor não os processa.

Tira-se a conclusão que para **1000** utilizadores o servidor é responsivo, disponível (com **Error %** inferior a 1%) e responde relativamente rápido, apesar de não tão rápido quanto seria desejável, porém é um resultado aceitável.

2.5 Limite de Clientes

Tendo em conta todos os testes realizados anteriormente, conclui-se, então, que o sistema tem elevado desempenho, disponibilidade e tempo de resposta aceitável para os seguintes números de clientes:

- **5000**, quando recebe muitos pedidos de baixa carga, que é o melhor cenário possível, mas também o mais irrealista;
- **1000**, num cenário mais realista, quando são executados vários pedidos de carga relativamente acrescida.

O objetivo deste projeto é, para além de melhorar o **número simultâneo** de clientes que o sistema suporta, permitir o elevado **desempenho**, **alta disponibilidade**, **balanceamento de carga** e **escalabilidade**, uma vez que este é um número demasiado baixo de utilizadores que podem utilizar a plataforma, especialmente para um servidor *web*.

3 Proposta de Solução

3.1 Virtual Machines (GCP)

Todas as máquinas virtuais na nova solução seguem uma arquitetura exatamente igual à utilizada para os testes base, também pelos mesmos motivos já enunciados na secção 2.2. Evidentemente, máquinas com melhores características levariam a melhor desempenho, mas com maior custo financeiro.

Dependendo da solução necessária para garantir alta disponilidade, desempenho e balanceamento de carga, foram acrescentadas às máquinas virtuais funcionalidades **extra** usando a *cloud*, que será visto em detalhe nas secções seguintes.

3.2 Eliminação de SPOFs

3.2.1 Armazenamento

Começando pela camada mais abaixo da *stack* da arquitetura da aplicação, foi necessário criar um sistema de alta disponibilidade e eficiência de armazenamento de informação, que é acedido pela base de dados para guardar as várias tabelas com um número variável de linhas.

Se o disco original da máquina se avariasse ou ficasse indisponível, por algum motivo, perdia-se toda a informação na base de dados, pelo que é boa ideia ter sempre um *backup* algures, para evitar esta situação.

A solução utilizada foi o **DRBD**, *Distributed Replicated Block Device*, uma ferramenta utilizada nas aulas práticas que replica um volume de armazenamento numa rede de nós interligados por ela. Se um dos volumes se avariar, existe sempre o *backup* nos nós e, caso se conecte um novo nó, terá também a informação copiada para ele.

É possível utilizar um número arbitrário de nós, limitado apenas pela capacidade financeira de quem está a organizar o sistema. Com a limitação financeira imposta, o grupo decidiu apenas utilizar **2 nós**, porém é perfeitamente escalável caso seja necessário no futuro.

Cada um destes nós é uma instância da máquina virtual base, cada um com um **disco rígido extra com 10 GB**, que alojará toda a informação dos blocos replicados. É necessário o disco extra porque caso fosse utilizado o mesmo disco onde se encontra o sistema operativo, caso ele por algum motivo se avarie, seja por causa do *kernel* ou por culpa do utilizador, perder-se-à toda a informação da base de dados e do sistema operativo, **simultaneamente**. Ter um disco extra dá sempre uma folga a estes acontecimentos, permitindo trocar o disco criado unicamente para servir de bloco de replicação por um mais recente, dando a possibilidade, então, de isolar problemas de armazenamento com problemas externos, como o do sistema operativo.

Foi selecionado este tipo de disco porque era um bom balanço entre preço, performance e capacidade.

3.2.2 Serviços Aplicacionais

Como a aplicação utiliza a base de dados **PostgreSQL** para guardar a informação pertinente, pode-se perfeitamente separar as duas componentes.

Pode-se, portanto, deixar o servidor correr numa máquina e a base de dados noutra, criando instâncias com **memória** e **CPU** dedicadas a estas funcionalidades, levando a um maior desempenho.

Com esta premissa, é necessário ter algumas considerações em conta:

- O PostgreSQL obriga a, num *cluster*, ter apenas **uma** instância **simultânea** a ser executada, devido à maneira como as transações e operações na base de dados funcionam;
- Caso essa instância falhe por algum motivo, deve ser imediatamente migrada para uma outra máquina todos os serviços fornecidos pelo PostgreSQL.

Esta situação em concreto foi já abordada em contexto de aula prática, através da utilização do **Red Hat High-Availability Cluster** (utilizando **Pacemaker**, **Corosync** e **Heartbeat**), em modo *failover*, que permite lançar os serviços numa só máquina e migrar para a secundária, caso haja uma falha na principal.

Este *cluster* será então composto por um número arbitrário de máquinas, com uma delas a servir como primária e todas as outras como *backup* para efeitos de migração de serviços, em caso de falha. Por existir a limitação do crédito financeiro, serão apenas utilizadas **duas** máquinas mas, novamente, é escalável para um número maior com facilidade.

Além dos 3 serviços base (**Pacemaker**, **Corosync** e **Hearbeat**), o *cluster* tem, também, de ter 2 recursos ativos: **PostgreSQL** e **Filesystem**, porque é necessário dar *mount* ao volume lógico antes de o utilizar para operações de leitura e escrita, pelo que o serviço de base de dados só é ligado apenas quando o **Filesystem** acaba de dar *mount* ao volume.

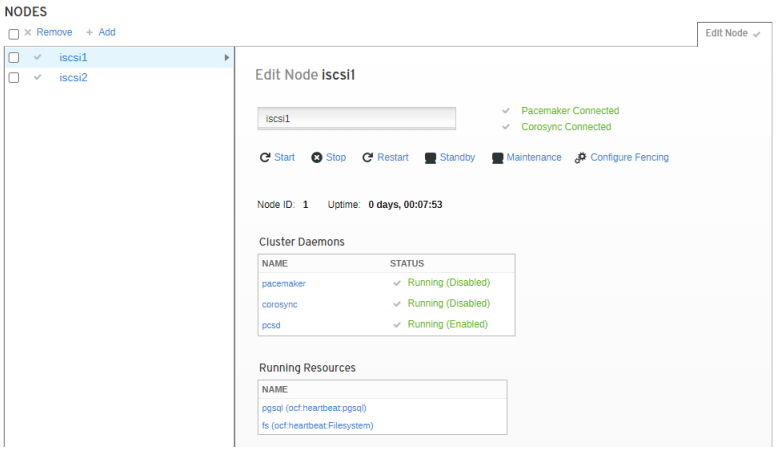


Figura 8: Visualização dos nodos do *cluster*

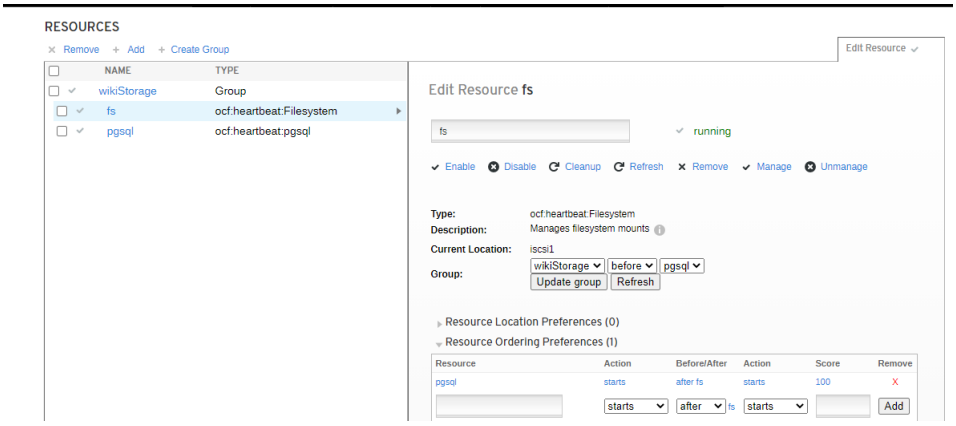


Figura 9: Visualização dos recursos do *cluster*

Para ligar o serviço do PostgresQL ao armazenamento, é utilizado o **ISCSI**, também utilizado no contexto das aulas práticas. Os nós **DRBD** servirão como **ISCSI targets** e os nós deste *cluster* servirão como **ISCSI clients**. O **ISCSI** serve para estabelecer ligações a unidades de armazenamento externas (**targets**) e, através de um único ponto de escrita (**clients**), será replicado para o primeiro **target** disponível, através do **Multipath** (para garantir balanceamento de carga), garantindo assim que há sempre forma da base de dados guardar a informação, mesmo com falhas em uma das máquinas intermédias.

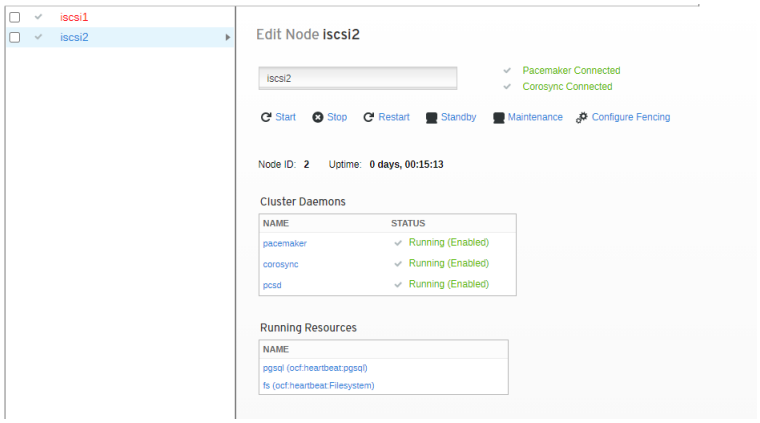


Figura 10: Migração dos recursos da máquina 1 para a máquina 2

3.2.3 Web-Server

A Wiki.js é uma aplicação desenhada utilizando Node.js e, para além disso, não guarda qualquer **estado** do utilizador em memória, uma vez que qualquer operação de **CRUD** requisitada pelo cliente é processada diretamente na base de dados, que já se colocou separada do servidor principal.

Sendo assim, pode ser criado um número totalmente arbitrário de instâncias deste servidor, uma vez que só precisa de ter o IP da base de dados externa (que, no caso da da solução apresentada, é o IP de um diretor que realiza balanceamento de carga sobre 2 instâncias que exercerão esta função), em que o utilizador pode aceder a qualquer um deles sem causar nenhum problema.

No entanto, o cliente não vai estar manualmente a testar se os servidores estão ativos, como é evidente, pelo que é necessário um único IP externo dum diretor que abstraia o balanceamento de carga para todos os servidores disponíveis (será visto em detalhe mais à frente).

Tendo estas observações em mente, foi estabelecido então que seriam usados, para efeitos de balanceamento de carga, **2** instâncias de servidores, apenas por causa das limitações impostas pelo cupão, ao utilizar 8 máquinas simultâneas, 2 discos extra e 2 balanceadores de carga, será utilizado grande parte do crédito oferecido.

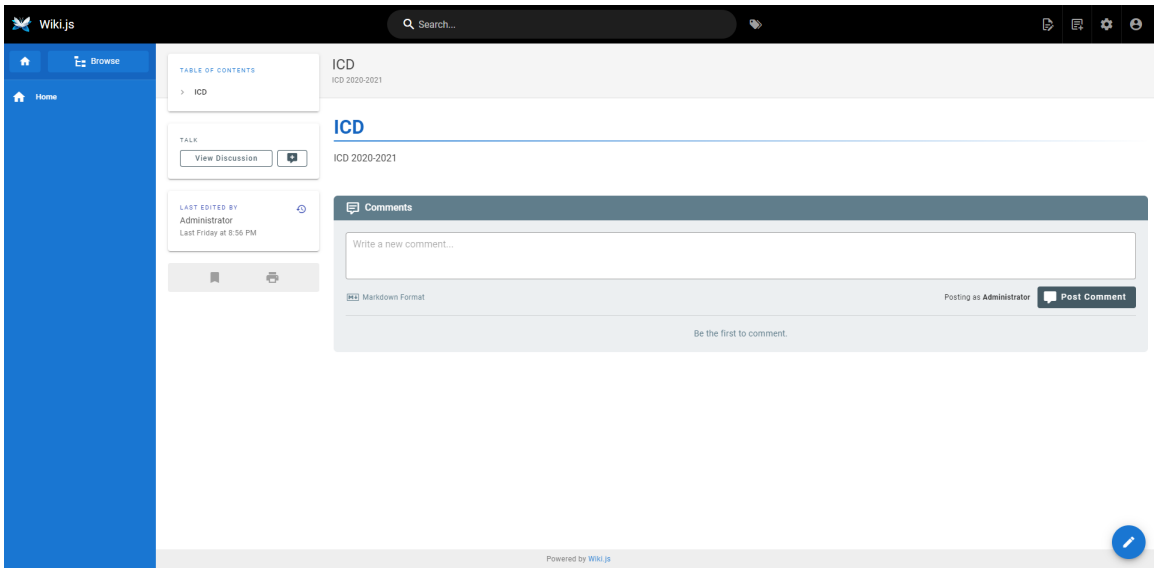


Figura 11: Wiki.js com alta-disponibilidade a funcionar

3.3 Visualização da Arquitetura Proposta

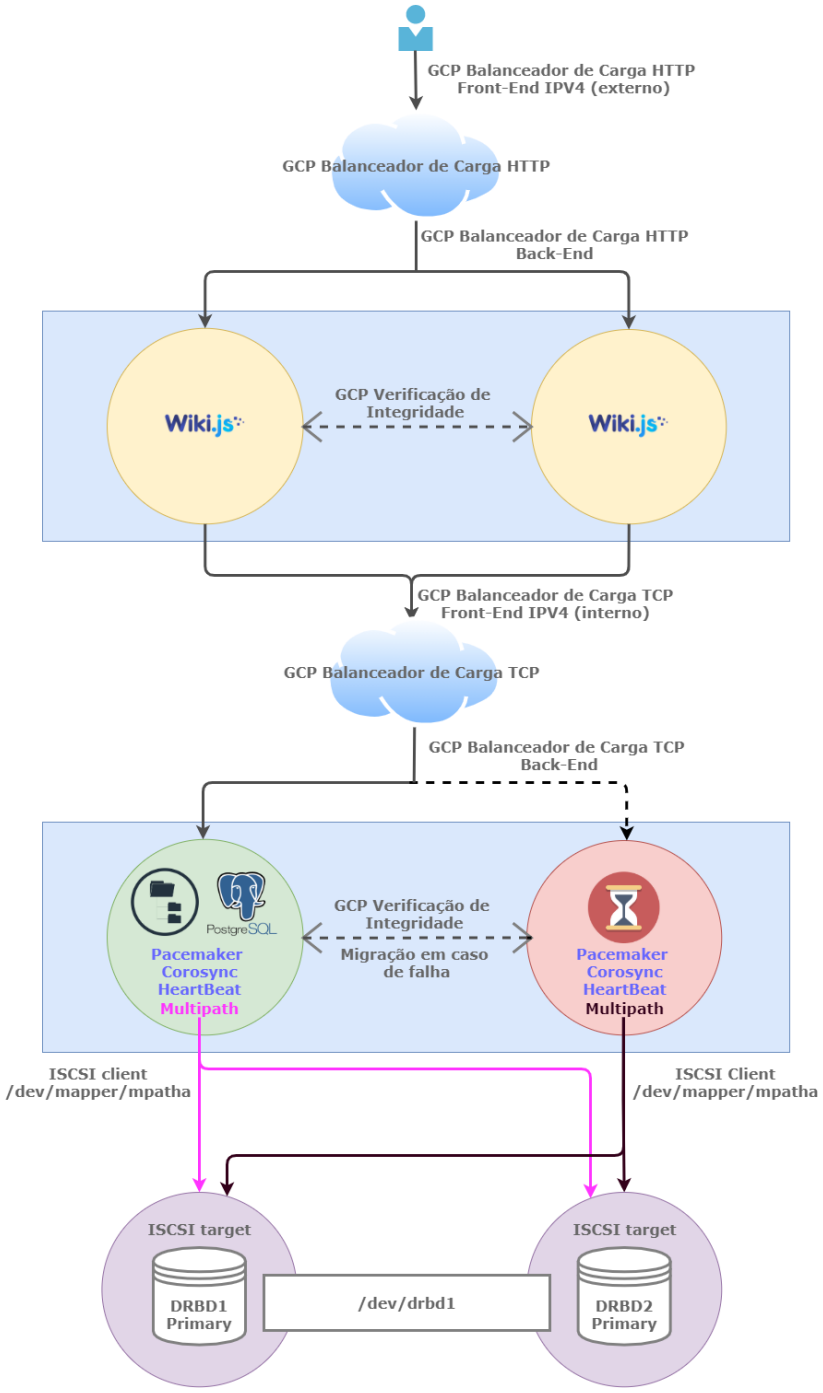


Figura 12: Solução desenvolvida para a plataforma Wiki.js

3.4 Observações acerca da GCP

3.4.1 Balanceadores de Carga

Durante as aulas práticas foram duas abordadas duas ferramentas: **IPVS** e **Keepalive** para as funções de balanceamento de carga e redirecionamento de tráfego através dum diretor.

Todo o projeto é realizado na *Google Cloud Platform*, **GCP**, e, portanto, impossibilita a utilização destas ferramentas, porque todas as instâncias de máquinas virtuais são gerenciadas

por uma rede virtual interna da Google. Esta rede cria endereços IP efémeros, onde a cada *restart* da máquina é-lhe atribuído um novo endereço.

Ter que alterar, nas configurações destas ferramentas, os endereços de IP de cada vez que uma máquina é reiniciada é um pesadelo de manutenção, pelo que é uma estratégia inviável de gestão de tráfego.

A Google, no entanto, fornece aos administradores do sistema outras ferramentas com o mesmo propósito, tirando partido dos IP's flutuantes implementados na sua rede, em que permitem a configuração estática dum endereço IP para ser utilizado.

O grupo decidiu, então, tirar partido dos **balanceadores de carga HTTP e TCP** fornecidos nas ferramentas de redes virtuais, que permitem associar um IP virtual fixo (que atua como o endereço do diretor para os grupos de instâncias de máquinas virtuais), levando a que o balanceamento de carga seja deixado ao critério da Google.

Esta abordagem tem bastantes benefícios, como por exemplo:

- Abstração do mecanismo de **balanceamento de carga**, a Google decide qual o melhor para uma dada configuração;
- **IP fixo** do *front-end* do balanceador, isto é, o IP do diretor que atua como *gateway* para as várias máquinas do *back-end* do balanceador, que têm IP's voláteis;
- **Verificador de integridade** embutido, é possível indicar uma porta para realizar a verificação de integridade com um determinado protocolo e usando determinados intervalos de tempo;
- Garantia de **alta disponibilidade**, caso o verificador de integridade determine que uma máquina está a ter erros, remove-a do *back-end* e são utilizadas as restantes máquinas funcionais.

3.4.2 Firewall

É necessário abrir as portas necessárias na **Firewall** da rede virtual da GCP, uma vez que os vários serviços disponibilizados pela aplicação trabalham em portas diferentes, que são bloqueadas por defeito por motivos de segurança.

A figura seguinte demonstra como estão configuradas na *cloud*.

☐	hacluster	Entrada	Aplicar a todas	Intervalos de IP: 0.0.0.0/0	tcp:2224
☐	pgcluster	Entrada	Aplicar a todas	Intervalos de IP: 0.0.0.0/0	tcp:5432
☐	wikihttp	Entrada	Aplicar a todas	Intervalos de IP: 0.0.0.0/0	tcp:3000

Figura 13: Regras de Firewall criadas

As portas abertas são, nomeadamente:

- **2224**: serve como controlador do **Red Hat High-Availability Cluster**
- **5432**: onde está à escuta o *daemon* de **PostgreSQL**
- **3000**: onde o servidor **Node.js** da Wiki.js (que contém o **GraphQL**) está à escuta

3.4.3 Grupos de Instâncias

É possível agrupar várias máquinas virtuais num só **Grupo de Instâncias**, com o objetivo de criar balanceadores de carga nessas máquinas.

Foram criados **dois** grupos de instâncias, um para o *cluster* de alta disponibilidade onde está alojado o **PostgreSQL** e o **Filesystem** e outro para as **instâncias** do servidor Node.js.

A imagem seguinte ilustra a configuração final dos balanceadores criados.

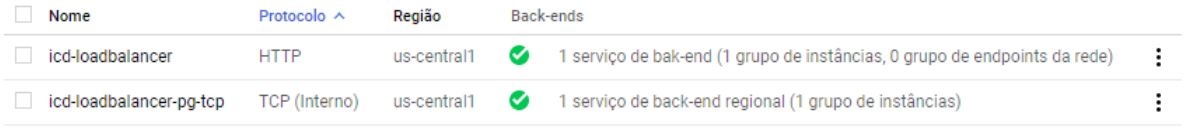


Figura 14: Balanceadores de carga criados

As figuras seguintes expõem as características do balanceador HTTP, releva-se o facto de estarem **2/2** instâncias íntegras, para garantir que existem duas instâncias do *web-server* a correr simultaneamente.

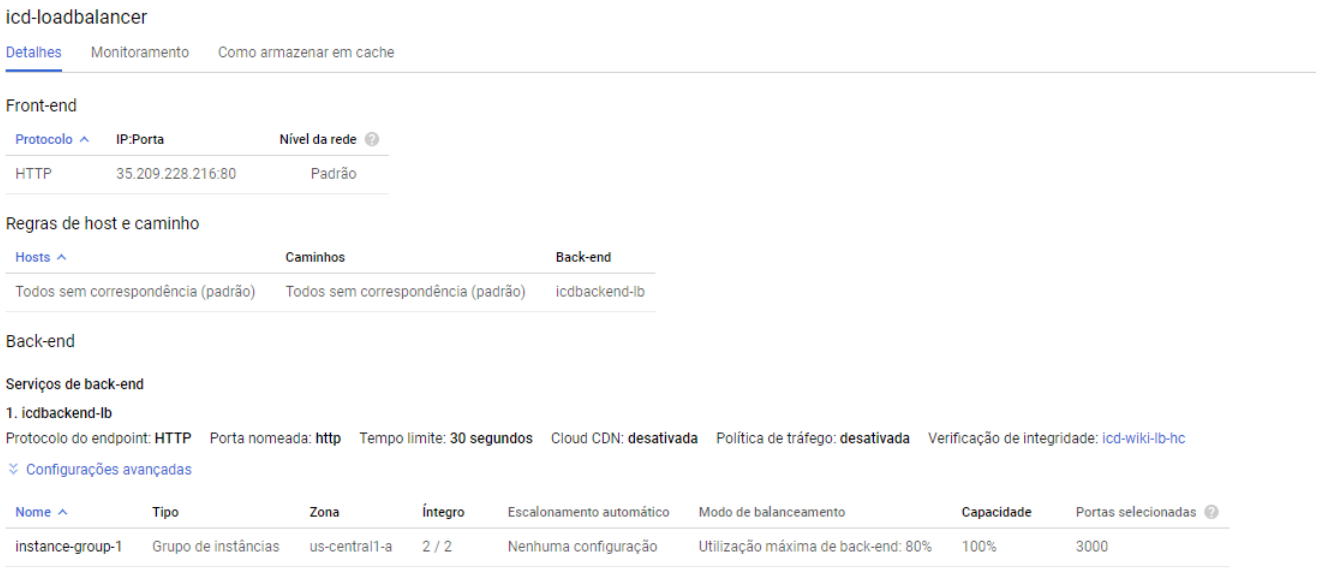


Figura 15: Balanceador de carga HTTP

Figura 16: Grupo de instância associado ao balanceador HTTP

Para o balanceador de carga TCP, o requisito é ter sempre **1/2** instâncias íntegras, porque só é possível ter uma instância do PostgreSQL a correr num *cluster* (o verificador de integridade vai tentar verificar se existe algo à escuta na porta 5432 da máquina secundária e, como não existe, vai assumir que ela não está ativa), obrigando a migração para a outra máquina mal seja detetado uma falha na primária.

Figura 17: Balanceador de carga TCP

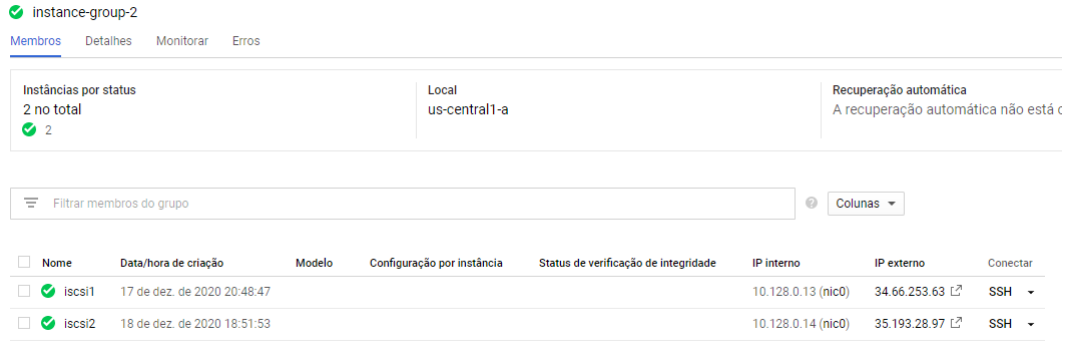


Figura 18: Grupo de instância associado ao balanceador TCP

3.5 Testes de Desempenho

De modo a estabelecer uma relação direta com o cenário inicial observado, foram realizados os mesmos testes com os mesmos propósitos.

Foi utilizado, também, o mesmo computador com o JMeter instalado, uma vez que se pretende comparações à arquitetura inicial.

3.5.1 Teste 1: Página Inicial

De igual modo ao teste criado inicialmente, o único pedido HTTP neste teste é o seguinte:

- **GET** /

Os resultados obtidos para a requisição da página inicial estão disponíveis na imagem seguinte, sobre a forma de tabela.

Threads	Average	Min	Max	Std. Dev.	Error %	Throughput
1000	953	383	1546	181.77	0.00%	524.10901
5000	953	266	4052	625.02	0.00%	886.05352
10000	785	267	1986	370.89	0.03%	946.43195
25000	997	265	29063	602.27	0.67%	572.17403
50000	1083	321	30767	679.81	9,96%	931.14956

Figura 19: Novos resultados do Teste 1

De notar que o **Throughput** mantém o mesmo comportamento que os testes iniciais, isto é, aumenta consoante o número de *threads*, que é expectável uma vez que os testes são realizados num só computador.

Quanto à **latência**, comparando aos resultados iniciais, obtém-se uma melhoria significativa. Agora, para **25000** clientes, tem-se um tempo de resposta inferior a 1 segundo, enquanto antes rondava os 3 segundos com taxa de erros altíssima.

Relativamente à **Error %**, obtém-se uma taxa inferior a 1 % para um número de clientes até **25000**.

Para ter uma percentagem semelhante a esta nos testes iniciais, era necessário um número de clientes inferior a 5000, que se traduz numa melhoria bastante significativa.

3.5.2 Teste 2: Login

Este teste utiliza o mesmo código GraphQL que os iniciais, assim como os mesmos pedidos HTTP, que são:

- **GET** /
- **POST** /graphql
  - Body Data (JSON): presente no anexo A.1

Na imagem seguinte estão os resultados do teste realizado.

Threads	Average	Min	Max	Std. Dev.	Error %	Throughput
1000	1496	266	4354	1106.35	0.00%	394.24404
5000	860	266	2973	426.16	0.03%	1210.80034
10000	1801	265	10343	1452.64	0.24%	1258.89092
25000	1649	217	10982	1499.06	11.31%	1461.73186
50000	1141	244	8817	947.42	25.54%	1307.24081

Figura 20: Novos resultados do Teste 2

Como habitual, o **Throughput** mantém o mesmo comportamento que os testes anteriores, tende a aumentar com o aumento do número de clientes pelas mesmas razões já mencionadas anteriormente.

Relativamente à **latência**, também baixou bastante, comparativamente ao teste inicial. Por exemplo, para **10000** clientes obtinha-se latências, em média, de 8 segundos e, agora, obtém-se cerca de 2 segundos.

No entanto, adicionar um pedido POST extra gerou dificuldade de atendar os 25000 clientes que no teste base conseguia realizar sem problemas. Isto leva a que, nestas condições, devido à **Error %**, o sistema tem alta disponibilidade e perfomance até cerca de **10000** clientes.

Pelos valores da tabela, é possível tirar a conclusão que o **balanceamento de carga** está a ter um impacto muito positivo nos valores obtidos, uma vez que a distribuição de carga por várias máquinas leva a pacotes não serem perdidos tão facilmente e, para além disso, serem processados mais rapidamente, pelo que baixa muito a **latência**.

3.5.3 Teste 3: Login + Criar Página

O teste de seguida utiliza, também, os mesmos objetos JSON nos dados dos pedidos HTTP, uma vez que são os mesmos que os testes iniciais.

- **GET** /

- **POST** /graphql
  - Body Data (JSON): presente no anexo A.1
- **GET** /en/new-page
- **POST** /graphql
  - Body Data (JSON): presente no anexo A.2

Está disponível na imagem seguinte os resultados provenientes do JMeter para este teste.

Threads	Average	Min	Max	Std. Dev.	Error %	Throughput
1000	1709	267	7047	1318.93	0.08%	488.99756
5000	3130	268	11926	2743.93	0.87%	1053.7963
10000	5987	762	23596	5708.1	10.35%	886.10783
25000	3965	449	34367	6006.79	33.63%	2029.57105
50000	2538	309	35457	4840.99	54.51%	1578.53835

Figura 21: Novos resultados do Teste 3

Novamente, o **Throughput** mantém o mesmo comportamento que os testes anteriores, pelas razões já discutidas.

A **latência**, como já aconteceu nos testes anteriores, baixou significativamente. Por exemplo, para **5000** clientes, previamente havia um tempo de resposta de cerca de 7 segundos que, agora, passa a ser em média 3 segundos.

A **Error %** era expectável subir e, de facto, alcançou 10% para **10000** clientes, pelo que para existir alta disponibilidade e desempenho, a aplicação é limitada a cerca de **5000** utilizadores, neste tipo de cenários, uma vez que é inferior a 1%.

Olhando para a tabela, é possível tirar a conclusão que o **balanceamento de carga** está, novamente, a ajudar muito os servidores de Wiki.js, pelo simples motivo de se notar uma **latência** muito baixa comparativamente ao teste inicial. Ter respostas mais rápido é sinal que estão a ser processadas mais rapidamente e não são descartadas.

3.5.4 Teste 4: Login + Criar Página + Procurar Página

Este teste também utiliza os mesmos dados em JSON que os testes iniciais, uma vez que os pacotes HTTP são muito semelhantes (só muda o cabeçalho).

- **GET** /
- **POST** /graphql
  - Body Data (JSON): presente no anexo A.1
- **GET** /en/new-page
- **POST** /graphql
  - Body Data (JSON): presente no anexo A.2
- **GET** /
- **POST** /graphql
  - Body Data (JSON): presente no anexo A.3

Nestas condições, obtiveram-se os seguintes resultados:

Threads	Average	Min	Max	Std. Dev.	Error %	Throughput
1000	1495	268	8664	1474.49	0.17%	538.93829
5000	3784	397	22440	4322.75	0.92%	1004.52095
10000	4814	601	27483	5442.89	18.76%	1409.09091
25000	3169	336	42595	5584.82	48.52%	2248.07177
50000	2087	267	30136	3865.79	62.21%	1904.29569

Figura 22: Novos resultados do Teste 4

De igual modo aos casos anteriores, o **Throughput** mantém o mesmo comportamento, pelas razões já enunciadas.

Relativamente à **latência** baixou significativamente, apesar de não ser tanta disparidade como os casos anteriores. Baixou de cerca de 7 segundos para aproximadamente 4 para **5000** clientes, por exemplo.

A **Error %** aumentou em todos os casos, porém, é de notar que se mantém uma elevada disponibilidade para um número de clientes perto de **5000**, como no teste anterior.

É cada vez mais notório que o sistema não suporta **10000** clientes simultâneos, ter quase 20% de taxa de erros é totalmente insustentável, uma vez que equivale a recusar 4 em cada 5 pedidos do utilizador.

Como já verificado nas situações anteriores, o **balanceamento de carga** está a ajudar o sistema a ter esta melhoria significativa de performance, porém, já se começa a notar um dos gargalos principais que está a causar este aumento abrupto da taxa de erros: a instância única de **PostgreSQL**, que será analisado na secção 3.6.

3.5.5 Teste 5: Login + Criar Página + Procurar Página + Criar Comentário

Por fim, com uma réplica exata do teste executado inicialmente, foram gerados os seguintes pacotes HTTP para envio aos servidores:

- **GET** /
- **POST** /graphql
  - Body Data (JSON): presente no anexo A.1
- **GET** /en/new-page
- **POST** /graphql
  - Body Data (JSON): presente no anexo A.2
- **GET** /
- **POST** /graphql
  - Body Data (JSON): presente no anexo A.3
- **GET** /en/new-page
- **POST** /graphql
  - Body Data (JSON): presente no anexo A.4

Os resultados obtidos para este último teste encontram-se na imagem seguinte.

Threads	Average	Min	Max	Std. Dev.	Error %	Throughput
1000	1352	265	7437	1045.54	0.04%	621.84221
5000	3723	533	16694	3254.88	1.17%	1114.91346
10000	4016	462	28384	5293.5	19.00%	1693.05321
25000	3305	331	46346	6205.46	41.66%	1711.65852
50000	2515	261	64779	5976.77	73.58%	1904.91479

Figura 23: Novos resultados do Teste 5

Como seria de esperar, o **Throughput** tem o comportamento que teve em todos os testes anteriores, pelo que as conclusões são as mesmas.

Comparando ao teste inicial, a **latência** também baixou significativamente, mas não tanto como nos testes anteriores a este. Por exemplo, para **5000** clientes, apenas baixou cerca de 1 segundo, no total, apesar de ser ligeiramente alta, mesmo para servidores localizados nos Estados Unidos.

A maior surpresa deste teste foram os resultados obtidos na coluna **Error %**. Os testes indicam que, para **5000** utilizadores, neste cenário de elevada carga simultânea o sistema consegue ser altamente disponível e com desempenho aceitável, enquanto que o mesmo já não é verdade quando se duplicam os clientes simultâneos.

Os **balanceadores de carga**, novamente, estão a ter um impacto enorme nestes cenários para garantir a alta disponibilidade ($\approx 99\%$) que os testes apontam que a plataforma tem. De facto, garantir que uma máquina não realiza os pedidos todos, simultaneamente, leva a que não sejam descartados pacotes com pedidos dos clientes, levando a uma maior **disponibilidade** e uma menor **latência**, que é um dos objetivos principais desta arquitetura.

3.6 Análise Final

3.6.1 Utilização de CPU

A situação que causou resultados mais extremos foi no **Teste 2**, onde existem melhorias imensas em tanto **latência** como **Error %**, pelo que uma análise extra foi realizada quanto à carga exigida a cada máquina no *back-end* no **balanceador de carga** implementado.

Após a análise dos resultados, era necessário verificar o que estava a acontecer em concreto na máquina virtual, pelo que o grupo procedeu à ferramenta de **Monitoramento** da GCP, para retirar os gráficos temporais de utilização de CPU para as máquinas envolvidas.

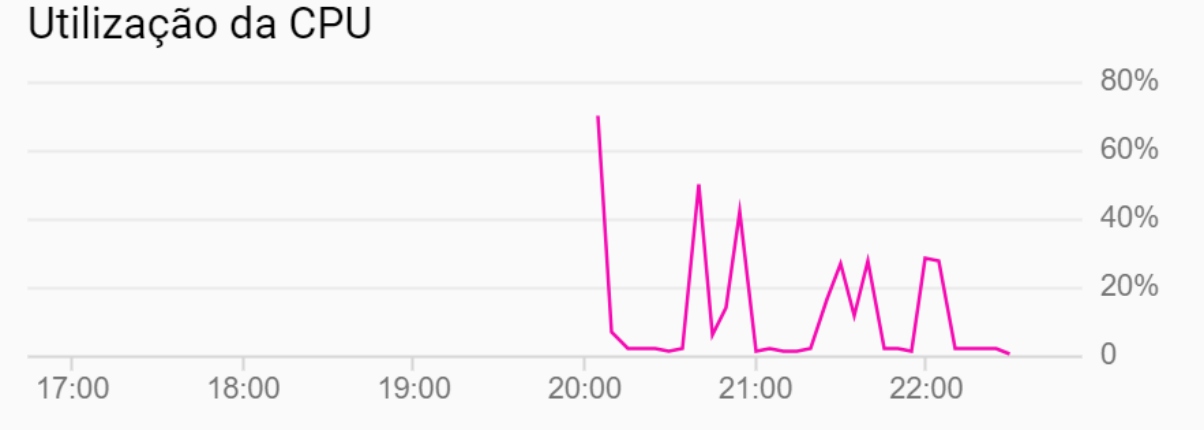
Figura 24: Utilização de CPU na versão da Wiki.js inicial no Teste 2

O segmento temporal relevante é apartir das **21:00h**, antes disso foi o tempo de inicialização da plataforma e outros mecanismos que não são relevantes aos testes.

A primeira coisa que se conclui é que o processador estava constantemente a atingir taxas de uso de **100%** (pelo gráfico estava mesmo a entrar em situações de *overload*, com taxas de uso superiores a 100%, como são núcleos virtuais, a Google permite distribuição por outros núcleos, nestas situações).

Naturalmente, isto é o que causa os pacotes a serem descartados, levando à **Error %** ser tão alta como foi nos testes iniciais. O CPU não consegue lidar com tantos pedidos simultâneos porque está a ser totalmente utilizado, pelo que todos os restantes pacotes são descartados da fila de espera por *timeout*.

(a) Utilização do CPU no primeiro servidor *web*



(b) Utilização do CPU no segundo servidor *web*

Figura 25: Utilização do CPU na versão final da Wiki.js no Teste 2

Novamente, o segmento relavante é apartir das **21:00h**, antes disso foi o tempo de inicialização das máquinas e outros eventos não relevantes à altura dos testes.

Aqui já se pode verificar que o CPU nunca é utilizado a **100 %**, pelo que os pacotes são processados a tempo antes de serem descartados por *timeout*.

O **balanceamento de carga** é a razão da **latência** ser tão baixa quando se utiliza a arquitetura proposta, todos os pacotes são processados atempadamente e, portanto, leva a uma taxa de **disponibilidade** tão alta como a observada nos testes.

Comparativamente ao gráfico anterior tem-se, em média, uma utilização a rondar os 40% do processador, ao contrário dos 75% obtidos anteriormente, que é uma enorme melhoria, como já foi provado pelos testes anteriores devido à baixa **Error %** e **latência**.

A imagem seguinte mostra, com os gráficos sobrepostos, as diferenças entre utilização de CPUs em ambas as arquiteturas estudadas. De notar que foi adicionado uma linha verde extra a indicar o segmento temporal relevante (divisão da direita).

Figura 26: Comparação Utilização de CPU das duas versões da Wiki.js no Teste 2

Este gráfico utiliza o recurso de **Explorador de Métricas** da GCP.

Apartir do mesmo, é possível visualizar o **balanceamento de carga** a atuar em ambas as máquinas que atuam como *web-servers* da arquitetura proposta, as linhas azul-neon e cor de rosa estão praticamente sobrepostas enquanto que a linha azul está muito acima delas, o que indica uma maior utilização de CPU, uma vez que não distribui a carga por vários servidores, que é o que acontece na solução proposta.

3.6.2 PostgreSQL

Como já foi discutido em secções anteriores, a arquitetura depende duma **única** instância de PostgreSQL a correr, com múltiplas máquinas em modo *failover* (no caso da solução discutida, é apenas 1, totalizando 2 máquinas ativas).

Quando se analisa os testes realizados na solução proposta, verifica-se um aumento abrupto da **Error %**, que é justificado pelo gargalo imposto pelo PostgreSQL correr numa única instância.

Uma vez que todo o processamento terá de ser feito apenas na máquina com a serviço disponível da API deste sistema de base de dados, caso ela utilize todo o seu poder de processamento, naturalmente deixará alguns pedidos por responder. É semelhante à situação dos CPUs anterior, quanto mais carga se coloca numa só máquina, mais pacotes ela deixará por responder por ter o seu CPU sempre ativo.

3.6.3 Limite de Clientes

A aplicação, agora, teve uma grande melhoria quanto ao número de clientes que suporta, simultaneamente.

Comparativamente ao cenário inicial, a nova arquitetura permite:

- **25000** clientes simultâneos num cenário de baixa carga, ao contrário dos **5000** que os testes iniciais indicavam;
- **5000** clientes simultâneos num cenário de carga relativamente alta, o mais próximo duma situação real, ao contrário dos **1000** obtidos anteriormente.

Conclui-se, então, que a aplicação tem alta disponibilidade e desempenho (cerca de 99%) para um número de clientes até **25000**, se a carga for relativamente baixa, e até **5000**, caso sejam enviados vários pedidos de carga elevada ao servidor.

Como estes cenários testam apenas os dois casos de limite, é expectável que a aplicação mantenha a mesma disponibilidade para um número de clientes a rondar **10000**, aproximadamente, tendo em consideração os testes anteriores.

4 Conclusão

O desenvolvimento deste projeto permitiu a consolidação dos objetvos de estudo abordados nas aulas da unidade curricular, nomeadamente a importância de garantir a alta disponibilidade e desempenho, balanceamento de carga e mecanismos de prevenção de perdas de dados.

Todo o projeto foi desenvolvido tendo em conta escalabilidade futura, eliminação de gargalos e garantias que, dependendo da capacidade financeira dos administradores do sistema, é simples a sua expansão, seja em *web-servers*, bases de dados, armazenamento ou serviços aplicacionais extras, uma vez que depende unicamente da criação de uma instância extra de uma máquina, garantindo uma forte modularização dos componentes.

Foram utilizadas todas as ferramentas das aulas práticas assim como a utilização delas num cenário real, assim como a pesquisa e aprendizagem de novas ferramentas, como o **GraphQL** e os balanceadores de carga da Google, que são também bastante poderosas para cenários como o deste projeto.

Concluindo, foi possível obter conhecimentos valiosos através da elaboração deste trabalho prático, uma vez que foram utilizadas ferramentas muito úteis que são relativamente simples e intuitivas de usar, assim como uma introdução prática à Google Cloud Platform, que é muito utilizada em produção empresarial.

A Anexos

A.1 GraphQL escrito em JSON relativo ao login

```
[{
    "operationName":null,"variables":{
        "username":"icd2021@uminho.pt","password":"icd2021","strategy":
    "local"
    },
    "extensions":{},
    "query":"mutation (username : String!,password: String!,
    strategy : String!)authenticationlogin(username :username, password:
    password, strategy :strategy) {
                        responseResult {
                            succeeded
                            errorCode
                            slug
                            message
                            __typename
                        }
                        jwt
                        mustChangePwd
                        mustProvideTFA
                        mustSetupTFA
                        continuationToken
                        redirect
                        tfaQRImage
                        __typename
                    }
                    __typename
                }
    }"
}]
```

A.2 GraphQL escrito em JSON relativo à criação de página

```
[
    {
        "operationName":null,
        "variables":{
            "content":"# Header\nYour content here",
            "description":"",
            "editor":
            "markdown",
            "locale":"en",
            "isPrivate":false,
            "isPublished":true,
            "path":"new-page",
            "publishEndDate":"",
            "publishStartDate":"",
            "scriptCss":"",
            "scriptJs":"",
            "tags":[],
            "title":"Untitled Page"
        },
        "extensions":{},
        "query":"mutation (content : String!,description: String!,
    editor : String!,isPrivate: Boolean!, isPublished : Boolean!,locale: String!,
    path : String!,publishEndDate: Date, publishStartDate : Date,scriptCss: String,
     scriptJs : String,tags: [String]!, title : String!)pagescreate(content :content,
    description: description,editor :editor, isPrivate: isPrivate,isPublished :
    isPublished, locale: locale,path :path, publishEndDate:
    publishEndDate,publishStartDate :publishStartDate, scriptCss:
    scriptCss,scriptJs :scriptJs, tags: tags,title :title) {
                            responseResult {
                                    succeeded
                                    errorCode
                                    slug
                                    message
                                    __typename
                                }
                                page {
                                    id
                                    updatedAt
                                    __typename
                                }
                                __typename
                        }
                        __typename
                    }
        }"
    }
]
```

A.3 GraphQL escrito em JSON relativo à requisição de uma página

```
[{
        "operationName":null,
        "variables":{
            "query":"new-page"
        },
        "extensions":{},
        "query":"query (query: String!)pagessearch(query :query) {
                        results {
                            id
                            title
                            description
                            path
                            locale
                            __typename
                        }
                        suggestions
                        totalHits
                        __typename
                    }
                    __typename
                }
        }"
}]
```

A.4 GraphQL escrito em JSON relativo à criação dum comentário numa página

```
[{
        "operationName":null,
        "variables":{
            "pageId":2,
            "replyTo":0,
            "content":"comment",
            "guestName":"",
            "guestEmail":""
        },
        "extensions":{},
        "query":"mutation ($pageId: Int!,replyTo: Int, $content: String!,guestName
    : String, $guestEmail: String)commentscreate(pageId :pageId, replyTo:
    replyTo,content :content, guestName: $guestName,guestEmail :guestEmail) {
                    responseResult {
                        succeeded
                        errorCode
                        slug
                        message
                        __typename
                    }
                    id
                    __typename
                }
                __typename
            }
        }"
}]
```