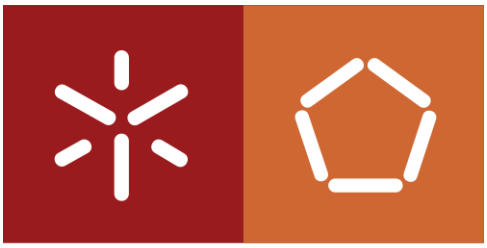

Universidade do Minho
Escola de Engenharia

Escalabilidade de Infraestruturas
Telemetria de Infraestruturas

Mestrado Integrado em Engenharia Informática
Infraestruturas de Centro de Dados
4.º Ano, 1.º Semestre

A81047 - Catarina Machado
A81822 - João Costa
A80987 - Tiago Fontes

Braga, dezembro de 2019

Conteúdo

Lista de Figuras

1 Introdução

Todo o *software* mora numa infraestrutura, sendo esta última composta por recursos físicos e virtuais que suportam o fluxo, armazenamento, processamento e análise de dados.

Cada vez mais todo o mundo se encontra conectado à tecnologia de uma forma também cada vez mais exigente, sendo necessário que a indústria IT se adapte e que responda com sucesso às necessidades do mercado, do utilizador e do *software*. Para garantir um bom serviço e uma elevada qualidade é essencial possuir uma infraestrutura segura e confiável, garantindo que o fluxo de informações é processado de maneira ininterrupta, isto é, sem falhas. Desta forma, é crucial que as infraestruturas sejam capazes de dar resposta a desafios relativos à tolerância à falhas, à escalabilidade, à alocação de recursos lidando com grandes volumes de dados, à elevada disponibilidade, eficiência energética, entre muitos outros.

Nesse sentido, o presente trabalho prático tem como principal objetivo implementar um serviço escalável e de elevada disponibilidade de telemetria de infraestruturas computacionais para a plataforma "Tucano". A plataforma "Tucano" foi desenvolvida para realizar a distribuição de alunos por opções, baseada na escolha destes.

Esta problemática é abordada na unidade curricular **Infraestruturas de Centro de Dados**, pretendendo-se ainda consolidar conteúdos leccionados na mesma, nomeadamente no planeamento, *deployment*, análise de desempenho e operação de infraestruturas de elevada disponibilidade e desempenho. Pretende-se assim que sejam estudados os três seguintes principais tópicos:

1. Avaliar o desempenho da plataforma, realizando testes de carga;
2. Garantir elevada disponibilidade da plataforma;
3. Avaliar o efeito da elevada disponibilidade no desempenho da plataforma.

O presente relatório descreve a arquitetura utilizada, identifica os SPOFs existentes na aplicação e as decisões do projeto que permitiram a sua eliminação. Descreve também a variação do desempenho do sistema resultante e a política de balanceamento utilizada.

2 Arquitetura Base do Sistema

O *software* da aplicação "Tucano" encontra-se dividido em *frontend* e *backend* e utiliza uma base de dados *PostgreSQL* para armazenamento de dados.

O *backend* recorre à *framework Phoenix*, escrito em *Elixir*, que compila para a *BEAM (Erlang virtual machine)*. O *frontend* utiliza como biblioteca base o *ReactJS* em conjunto com componentes na sua maioria do *Semantic UI*.

O *backend* e o *frontend* não se encontram interligados, apenas partilham a mesma API.

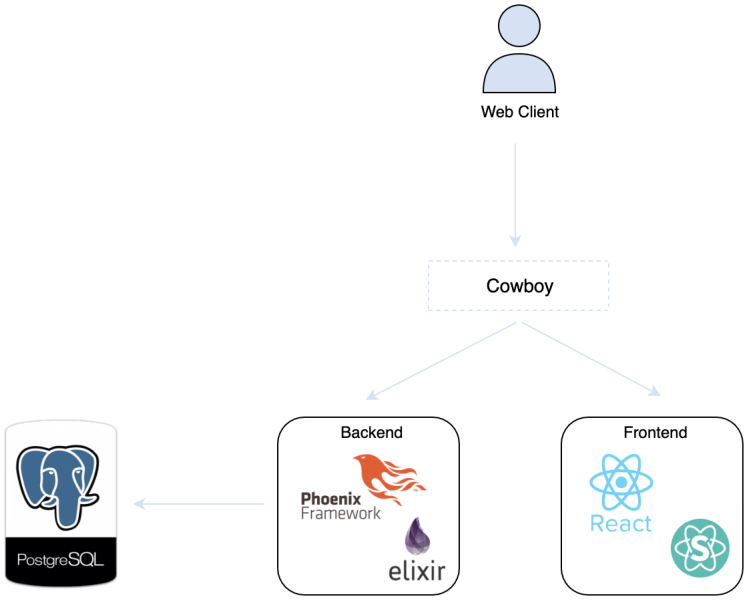

Figura 1: Arquitetura Base da plataforma.

2.1 Identificação de SPOFs

Um *Simple Point of Failure* (SPOF), provém do Inglês *ponto único de falha*, sendo um ponto que caso falhe, todo o sistema irá perder a sua operacionalidade. Assim, e quando apenas existe uma réplica de cada serviço (por exemplo, Base de Dados, *Web Server* ou Servidor Aplicacional), o nosso sistema está bastante vulnerável uma vez que o seu funcionamento será severamente comprometido caso um dos elementos falhe.

Concretizando, uma componente crítica da aplicação é obviamente a componente *backend*, por motivos óbvios, se toda a lógica da aplicação falhar, falham todos os serviços por ela disponibilizados.

Devido à sua implementação, a aplicação apresenta um *bottleneck* de desempenho que é a base de dados, uma vez que grande parte das operações usadas são efetuadas sobre a mesma, e perdendo o acesso às informações lá contidas, a aplicação torna-se inútil. Daí segue que a base de dados é um dos SPOF's desta aplicação, bem como os ***WebServers***. Como tal, um dos principais objetivos do trabalho prático é conseguir uma arquitectura distribuída e que permita resiliência em relação às falhas que podem ocorrer nos diferentes elementos que constituem a aplicação.

2.2 Desempenho da Plataforma: Testes de Carga

Para avaliar o desempenho da plataforma, recorremos ao *Selenium*, ao *JMeter* e ao *BlazeMeter* para realizar testes de carga e de *stress*.

O *Selenium* é uma ferramenta *open-source* de testes muito poderosa, usada principalmente para testes funcionais automáticos ao nível da interface gráfica. Pode ser muito útil também para testes de carga, pois permite que os utilizadores reutilizem os testes funcionais existentes e os executem com diversos utilizadores virtuais em simultâneo.

O *JMeter* é uma ferramenta especializada na realização de testes de carga em recursos estáticos ou dinâmicos oferecidos por sistemas computacionais. Permite configurar vários aspetos, entre os quais o número de *threads*, a quantidade de vezes que cada *thread* será executada e o intervalo entre cada execução.

O *BlazeMeter* é uma plataforma de testes de carga como serviço (PaaS), compatível com o *Apache JMeter*, explicado anteriormente.

Em seguida, iremos apresentar os diferentes testes de carga que efetuamos recorrendo às três ferramentas mencionadas.

É de realçar que todos os testes foram efetuados a partir de uma máquina diferente para assim permitir que a máquina que está a realizar os testes esteja somente a realizar os testes e, desta forma, garantir que os resultados não estão a sofrer qualquer tipo de enviesamento.

2.2.1 Teste 0: *Login* + Inserção de *manager* + Consulta de *managers*

Através da ferramenta *Selenium*, começamos por efetuar um teste que consistia no administrador fazer *login* na aplicação, em seguida inserir um novo *manager* e, por fim, consultar todos os *managers*. No Anexo A encontra-se o código *Java* desenvolvido para o efeito.

O teste foi realizado para um número de *threads* igual a 1, 2, 4, 8 e 16. O tempo de execução médio de cada *thread* é uma média de 5 execuções. Os tempos obtidos encontram-se em segundos, na tabela presente na Figura 2.

Login + Inserção de manager + Consulta de managers						
Threads	Média Exec.	Exec. 1	Exec. 2	Exec. 3	Exec. 4	Exec. 5
1	4.7648	5.145	5.375	4.127	4.914	4.263
2	8.2159	10.0675	10.587	7.063	6.827	6.535
4	9.3598	14.839	7.605	9.479	6.401	8.475
8	13.6494	12.537	13.236	15.577	13.916	12.981
16	31.94	28.309	36.717	31.221	30.688	32.765

Figura 2: Tempos de Execução Teste 0 (em segundos).

Figura 3: Gráfico tempos de Execução Teste 0 (em segundos).

Tal como se pode ver a partir do gráfico presente na Figura 3, a partir das 8 *threads* o tempo de execução do conjunto de operações mencionados aumenta substancialmente.

Devido às características da ferramenta *Selenium* e às limitações da nossa máquina, torna-se impossível efetuar este teste utilizando mais *threads*, o que torna o teste pouco representativo da realidade que queremos testar.

2.2.2 Teste 1: *Login*

Em seguida, decidimos utilizar a ferramenta *JMeter* para efetuarmos testes de carga ao *backend*. Este teste consistia em entrar na ***landing page*** do Tucano e, em seguida, efetuar o ***login*** na aplicação. Desta forma, são testados os dois seguintes *endpoints*:

1. **GET** /
2. **POST** /api/auth/sign_in

   Body data: {email: "admin@tupi.pi", password: "admin123"}

O teste foi realizado para um número de *threads* igual a 100, 200, 300, 400 e 500. Os gráficos dos resultados do pedido POST do *login* para as simulações com 100, 200 e 300 *threads* encontram-se na Figura 5, 6 e 7, respetivamente.

Na tabela seguinte é possível visualizar o *throughput*, a latência do pedido GET e do pedido POST, e a percentagem de erro no pedido GET e no pedido POST.

Login					
Threads	Throughput (por min)	Latência GET (ms)	Latência POST (ms)	Erro GET (%)	Erro POST (%)
100	198.913	90	25909	0	0
200	195.196	104	50611	0	4.5
300	294.334	149	60206	0	31.67
400	390.65	417	60093	0	50.25
500	484.928	384	60394	0	57.4

Figura 4: Métricas Teste 1.

Tal como se pode ver através da tabela, com 100 utilizadores em simultâneo a aplicação dá um erro de 0%, o que significa que todos os pedidos são realizados com sucesso. No entanto, quando o número de *threads* aumenta para 200, o erro sobre para os 4.5%. Isto significa que a partir de aproximadamente 200 utilizadores, a aplicação já não responde com a mesma disponibilidade.

Como seria de esperar, o *throughput* e a latência aumentam conforme o aumento do número de *threads*. Também é possível observar que os pedidos à página inicial não geram qualquer erro, uma vez que o seu conteúdo é maioritariamente estático.

O tempo de resposta (latência) é superior no caso do pedido POST tal como previsto, visto que exige acesso à base de dados para confirmação das credenciais.

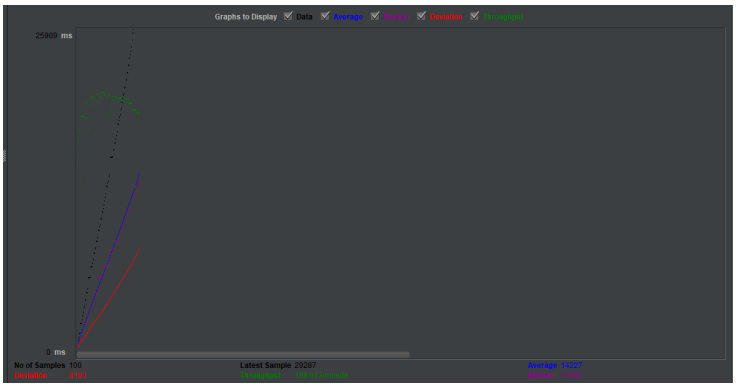

Figura 5: Gráfico dos Resultados do pedido POST do Teste 1 para 100 *threads*.

Figura 6: Gráfico dos Resultados do pedido POST do Teste 1 para 200 *threads*.

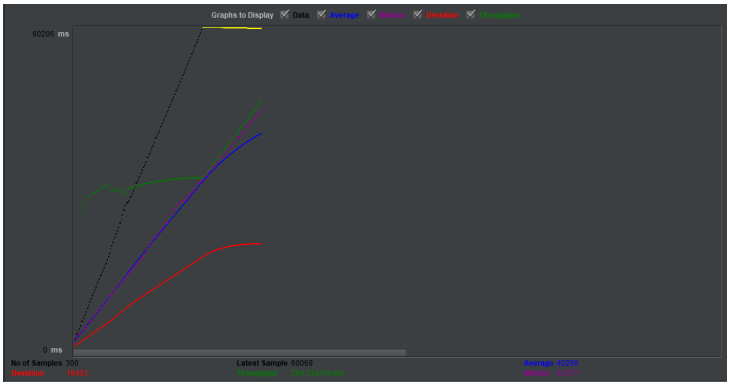

Figura 7: Gráfico dos Resultados do pedido POST do Teste 1 para 300 *threads*.

2.2.3 Teste 2: *Login* + Consulta de *managers*

Utilizando novamente a ferramenta *JMeter*, realizamos um novo teste de carga ao *backend* onde desta vez, após o *login*, o administrador visualiza a lista de todos os *managers* existentes no sistema. Assim, o teste é constituído pelos seguintes três passos:

1. **GET** /
2. **POST** /api/auth/sign_in

   Body data: {email: "admin@tupi.pi", password: "admin123"}
3. **GET** /api/v1/managers

Os gráficos dos resultados do pedido GET dos *managers* para as simulações com 100, 200 e 300 *threads* encontram-se na Figura 9, 10 e 11, respetivamente.

Na tabela seguinte é possível visualizar o *throughput*, a latência do pedido GET e do pedido POST, e a percentagem de erro no pedido GET e no pedido POST.

Login + Consulta managers						
Threads	Throughput (por min)	Throughput GET MAN (por min)	Latência POST (ms)	Latência GET MAN (ms)	Erro POST (%)	Erro GET MAN (%)
100	107.064	109.822	45521	326	0	0
200	194.09	200.354	49447	83	2	2
300	294.903	296.785	60052	144	27.33	27.33
400	390.549	390.035	60258	592	45.75	45.75
500	488.17	478.248	54265	579	59.8	59.8

Figura 8: Métricas Teste 2.

Também neste teste, constata-se que a aplicação começa a falhar a partir de aproximadamente 200 utilizadores. Tal como o grupo havia previsto, e como havia sido mostrado anteriormente, as implicações dos pedidos ***Login*** têm repercussões no seguimento do *pipeline* de pedidos.

Concretizando, o grupo realiza três pedidos HTTP, pelo que o primeiro incide sobre a página inicial, o segundo um POST para ser efetuado o *Login* e de seguida um GET aos *managers* do sistema. Com este seguimento de pedidos, o grupo consegue verificar a eficiência do *Login*, uma vez que a visualização dos gestores é uma ação que requer esse mesmo *Login*.

Como forma de comprovar esta sequência, a tabela mostra-nos que os clientes que conseguem autenticar-se conseguirão também eles fazer o método GET aos *managers* (relação direta entre o erro do POST com o GET), que atráves da observação da sua latência, verificamos que é um método menos exigente.

Note-se que existe uma grande diferença entre a latência dos pedidos *Login* quando comparados com a operação de listar os *managers* existentes nos sistema, uma vez que existe uma carga adicional no *Login*, tendo em conta a verificação das credenciais aliada às chaves criptográficas que é necessário validar e verificar.

De uma forma genérica, tal como observado no cenário de teste anterior, a simulação com cerca de 200 clientes começa a introduzir erro nos pedidos,

pelo que aqui é necessário atentar no número de utilizadores que conseguimos responder.

Em suma, e tendo em conta os vários cenários de teste apresentados, diríamos que a aplicação consegue realizar pedidos exigentes (como verificação feita no *Login*) para cerca de duas centenas de utilizadores. Contudo, com esta configuração, atingimos um nível de serviço na ordem de 97 +-1 %, pelo que na maioria dos casos reais, não será aceitável um erro superior a 2/3 %.

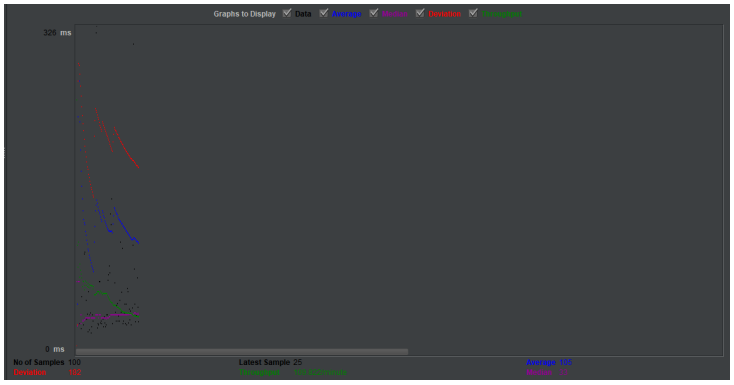

Figura 9: Gráfico dos Resultados do pedido GET *managers* do Teste 2 para 100 *threads*.

Figura 10: Gráfico dos Resultados do pedido GET *managers* do Teste 2 para 200 *threads*.

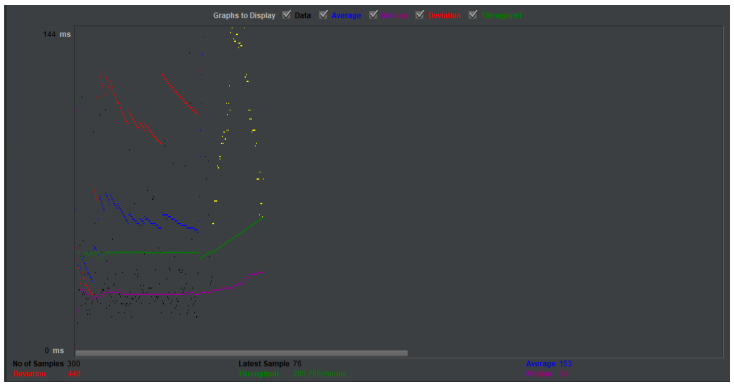

Figura 11: Gráfico dos Resultados do pedido GET *managers* do Teste 2 para 300 *threads*.

2.2.4 Teste 3: *Login* + Inserção de *manager* + Consulta de *managers*

Através da ferramenta *JMeter*, efetuamos também um teste que consiste no administrador fazer *login* na aplicação, em seguida inserir um novo *manager* e, por fim, consultar todos os *managers*.

Para tal, tivemos que ter em atenção que todas as *threads* teriam que inserir um *manager* com diferentes credenciais, para que todos esses pedidos fossem realizados com sucesso.

Com recurso às funcionalidades disponíveis no *JMeter*, cada *manager* a adicionar tinha como e-mail o tempo atual (horas, minutos...) e o número da *thread* que estava em execução, com recurso a funções do *JMeter*. Assim, o *body data* do pedido post de inserção é o que se pode verificar na seguinte lista com três *endpoints* que iremos testar:

1. **GET** /
2. **POST** /api/auth/sign_in
   Body data: {email: “admin@tupi.pi”, password: “admin123”}
3. **POST** /api/v1/managers
   Body data:
   {”data”:{”attributes”:{
   ”email”:”JM${__time()}_${__threadNum()}@tupi.pi”,
   ”name”:”JM${__time()}_${__threadNum()}”},
   ”type”:”manager”}}

Os gráficos dos resultados do pedido POST dos *managers* para as simulações com 100 e 200 *threads* encontram-se na Figura 13 e 14, respetivamente.

Na tabela seguinte é possível visualizar o *throughput* e a latência do pedido POST do *login*, do pedido POST da inserção do *manager* e do pedido GET da lista de *managers*, e a percentagem de erro obtida no pedido POST do *login* para cada uma das simulações com 100, 200, 300, 400 e 500 *threads*.

Login + Inserção Manager + Consulta Managers							
Threads	Throughput Login (por min)	Throughput Add MAN (por min)	Throughput Get MAN (por min)	Latência Login (ms)	Latência Add MAN (ms)	Latência Get MAN (ms)	Erro Login (%)
100	197.174	130.214	93.379	26440	16857	7762	0
200	199.687	131.845	117.706	51173	32195	134163	0
300	290.28	136.955	213.838	60096	50451	124316	34.33
400	386.554	142.02	244.468	60220	61363	123172	53.25
500	470.743	212.724	187.627	60281	60046	225882	60.8

Figura 12: Métricas Teste 3.

Através deste teste, é possível comprovar que o peso computacional de inserção de um *manager* não é tão elevado como o peso de efetuar *login* na aplicação, visto que a latência do *login* é superior à latência da *inserção*.

No entanto, à medida que o número de *threads* aumenta, a latência do pedido de inserção aumenta seguindo uma curva mais acentuada, o que significa que quantos mais utilizadores estiverem a tentar inserir dados em simultâneo, mais "lento" o sistema fica - *overbooked*, o que não se verifica com a operação de *login*.

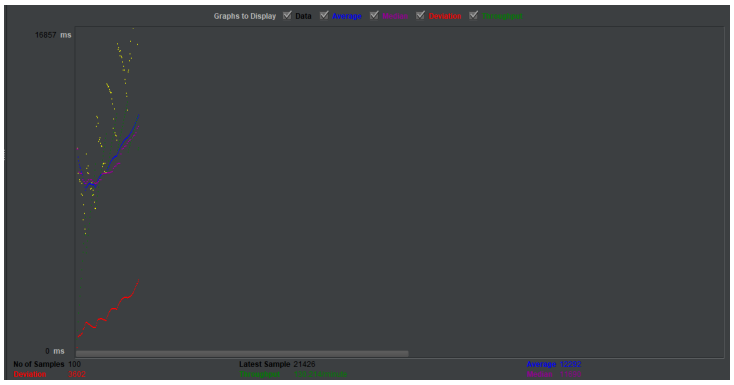

Figura 13: Gráfico dos Resultados do pedido POST *managers* do Teste 3 para 100 *threads*.

Figura 14: Gráfico dos Resultados do pedido POST *managers* do Teste 3 para 200 *threads*.

2.2.5 Teste 4: *Login User* + Candidatar a um concurso

Realizamos ainda um novo teste de carga ao *backend*, cujo objetivo foi simular vários utilizadores a candidatarem-se ao mesmo perfil em simultâneo.

Para tal, inicialmente criamos um novo concurso, com diferentes *subjects*, a partir de um *manager* e povoamos esse concurso com 300 diferentes *users*.

Os *users* previamente criados possuiam um e-mail utilizando o mesmo modelo: "a[**NÚMERO**]@tupi.pi", onde o **NÚMERO** varia de 1 até 300. Desta forma, voltamos a recorrer às funcionalidades do *JMeter* para conseguirmos fazer *login* através de cada um dos *users*, onde o **NÚMERO** é igual ao número da *thread* em execução (e serão, no máximo, 300 *threads*). Para simplificar o *body data* do pedido *post* de autenticação, a *password* de todos os *users* é igual.

Para a realização deste teste, criamos uma sessão através da extensão *BlazeMeter* de modo a facilitar a simulação dos vários passos necessários e, em seguida, abrimos a sessão através do *JMeter* de modo a conseguirmos correr o teste com várias *threads* e obtermos as respetivas métricas.

O teste foi constituído pelos seguintes passos:

1. **POST** /api/auth/sign_in

   Body data: {"email":"a${__threadNum()}@tupi.pi","password":"admin123"}
2. **GET** /api/v1/users
3. **GET** /api/v1/contests
4. **GET** /api/v1/contests/perfis/applicant
5. **GET** /api/v1/subjects
6. **GET** /api/v1/applications/339/applications

7. **GET** /api/v1/applications/339/applications/settings

8. **PATCH** /api/v1/applications/339

Para este teste, optamos por simular apenas com 100, 200 e 300 *threads*. Os gráficos dos resultados da conjunção de todos os pedidos encontram-se na Figura 16, 17 e 18, respetivamente.

Na tabela seguinte é possível visualizar o *throughput* e a latência do pedido POST e do pedido PATCH, e a percentagem de erro no pedido POST para cada uma das simulações.

Login + Realizar Concurso					
Threads	Throughput Login (por min)	Throughput PATCH (por min)	Latência Login (ms)	Latência PATCH (ms)	Erro Login (%)
100	164.28	58.259	31911	138	0
200	196.216	169.523	60070	81	25
300	293.17	234.941	60284	112	49.33

Figura 15: Métricas Teste 4.

Quando um utilizador se candidata a um concurso, primeiramente é necessário que este efetue o *login*, uma vez que todas as operações do sistema são operações possíveis exigem esse nível de autenticação. Assim sendo, após completar esse passo, o utilizador terá de aceder à aba **contests** para poder selecionar o concurso que se pretende selecionar. Depois, surgirão as várias *subjects* adjacentes a esse mesmo concurso, as quais podem sofrer alterações na ordem de preferência. No final, é submetido o resultado do concurso clicando no botão *Save* (**PATCH** que corresponde à gravação do estado). Como podemos ver pela observação da tabela, existe uma linearidade (que era prevista) relativamente ao *throughput* quer do *login*, quer no *patch*. Tal como haviamos referido anteriormente, o custo associado ao processo de *login* está associado à desencriptação da *password*, bem como ao processo de consulta da base de dados. Assim sendo, o acesso concorrente por parte de 200 utilizadores significará, em média, um acesso negado a cerca de 25% dos mesmos. Conforme se aumenta o número de acessos, a latência dos pedidos também aumenta. Se fosse testado com um número de *threads* superior, prevê-se um aumento no *throughput* (pelo menos até ao nível de saturação), bem como o tempo de resposta dos vários pedidos.

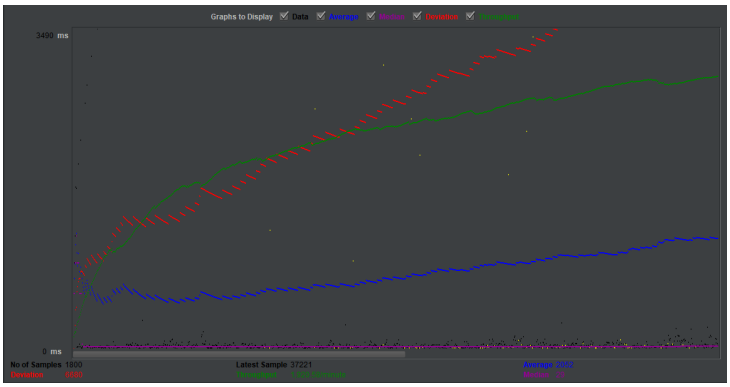

Figura 16: Gráfico dos Resultados Gerais do Teste 4 para 100 *threads*.

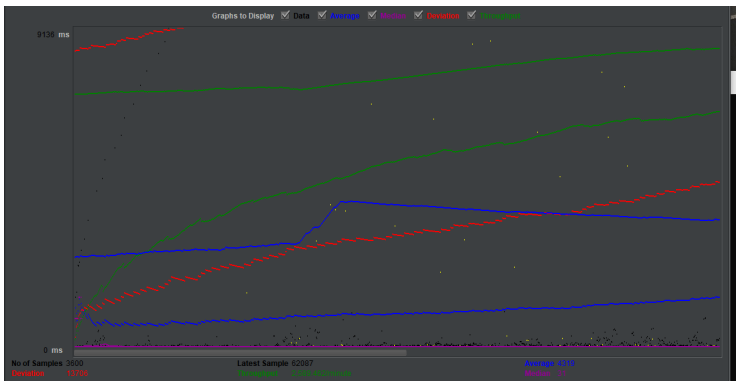

Figura 17: Gráfico dos Resultados Gerais do Teste 4 para 200 *threads*.

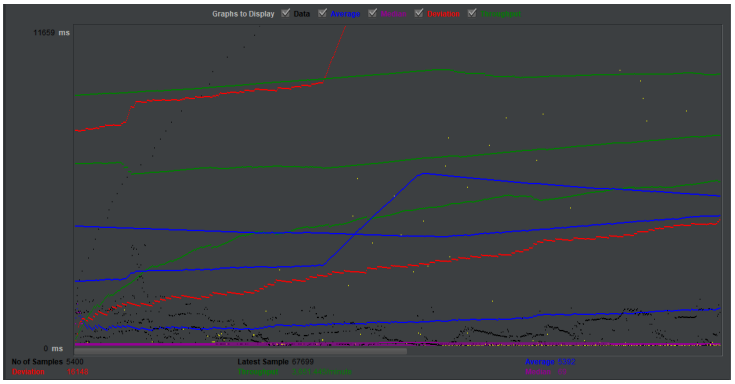

Figura 18: Gráfico dos Resultados Gerais do Teste 4 para 300 *threads*.

2.3 Disponibilidade da Plataforma

O objetivo de qualquer plataforma é garantir alta disponibilidade, de modo a que o sistema informático seja resistente a falhas de *hardware*, *software* e energia, e, desta forma, os seus serviços se encontrem disponíveis o máximo de tempo possível.

Quanto mais redundância existir na aplicação, menores serão os SPOF (*Single Point Of Failure*), e menor será a probabilidade de interrupções no serviço.

No caso da aplicação Tucano, devido à quantidade de SPOFs existentes, tal como mencionado na Secção 2.1 e comprovado através dos testes efetuados na Secção 2.2, a disponibilidade da plataforma não é tão elevada como desejado.

Desta forma, sentimos a necessidade de construir um sistema com *hardware* acessível, *Cluster*, altamente escalável e de custo reduzido. O *Cluster* permite agregar vários computadores ou servidores como se fosse uma única máquina de grande porte. Desta forma, não existirá nenhum ponto que, ao falhar, implique a indisponibilidade de outro qualquer ponto (SPOF) e a aplicação tornar-se-á numa aplicação de elevada disponibilidade.

3 Arquitetura Implementada

Nesta secção iremos abordar a solução encontrada para fazer frente ao problema. Assim, depois de analisar os procedimentos realizados nas aulas práticas, algum aconselhamento por parte do docente e muita pesquisa na *internet*, chegamos a um exemplo de arquitetura de alta disponibilidade. Desse modo, e realizando uma abordagem *buttom-up*, começamos por desenvolver a componente de *storage*.

3.1 *Storage*

A nossa componente de *storage*, de maneira a aumentar a disponibilidade dos nossos dados, utilizamos um *Distributed Replicated Block Device (DRBD)*. O disco é acedido através do protocolo *Internet Small Computer Systems Interface (ISCSI)*.

3.2 tupi (*Cluster*)

Como nos é impossível saber como a aplicação foi construída em termos de código-fonte, não sabendo assim se esta permite que várias instâncias da aplicação, em paralelo. Assim, como lançar várias instâncias da aplicação está fora de questão, é necessário utilizar outros métodos para garantir a disponibilidade da nossa *App*.

A solução encontrada foi então a utilização de um *High-Availability Cluster* em modo *fail-over*. Sucintamente, um *cluster* é um sistema constituído por, no mínimo, 2 máquinas, ás quais são atribuídos serviços previamente configurados. De maneira a garantir que a aplicação está *up* o maior tempo possível, o *cluster* está em constante verificação do estado das "suas"máquinas, caso detete algum problema numa delas migra os serviços que correm nessa máquina para uma outra que esteja disponível.

Nessa linha de pensamento, utilizamos um *cluster* com 2 máquinas e que distribuem 2 serviços.

O primeiro serviço é o da base de dados *postgresql* da *App*. Este divide-se em 3 "sub-serviços"que são **ip_psql**, que atribui um IP ao serviço, um **fs_db** que faz *mount* da partição dos dados que estão no *disco-DRBD*, e o **db_pg** que arranca o motor de base de dados *postgresql*.

No outro serviço existem 2 "sub-serviços"que são **ip_app** que atribui um IP à app, e **Tupi_App** que é o serviço que arranca o *backend* da aplicação.

Nas imagens que se seguem é possível ver na interface de *HACluster* da *Red Hat*, a organização do cluster de nome **tupi**.

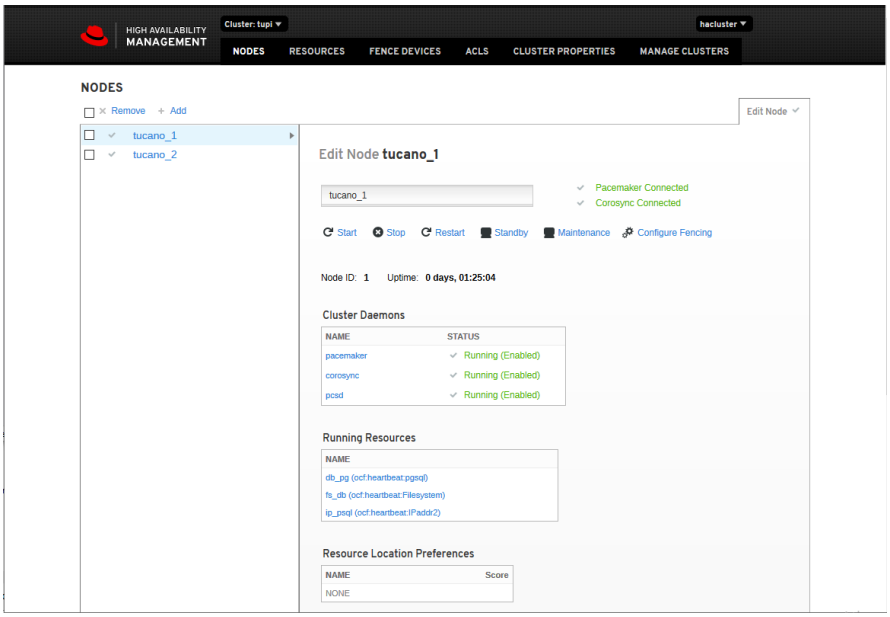

Figura 19: Vista dos nós do *Cluster* através da interface da Red Hat.

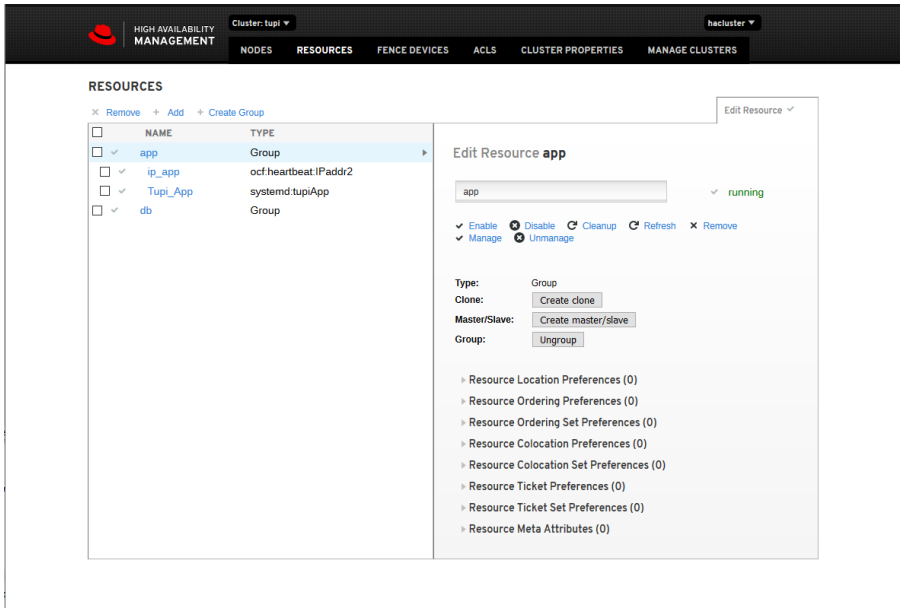

Figura 20: Vista dos serviços do *Cluster* através da interaface da Red Hat.

3.3 *Web Servers*

Para hospedar o *Frontend*, e como estes não possuem estado, optamos por lançar 2 estâncias destes.

3.4 Diretor (*LVS*)

De maneira a balancear o tráfego e tolerar qualquer falta dos *Web Servers*, utilizamos um *Linux Virtual Server(LVS)*, com base no serviço *KeepAlived*, que nos permite a constante monitorização dos *Web Servers*, para que tanto em motivos de balanceamento, como em caso de falha, seja possível tomar medidas de maneira a manter a aplicação no ar e da melhor forma possível no momento.

3.5 *Overview* de todo Sistema

De maneira a concretizar, o que foi explicado anteriormente, iremos apresentar um diagrama que mostra a arquitetura de todo sistema.

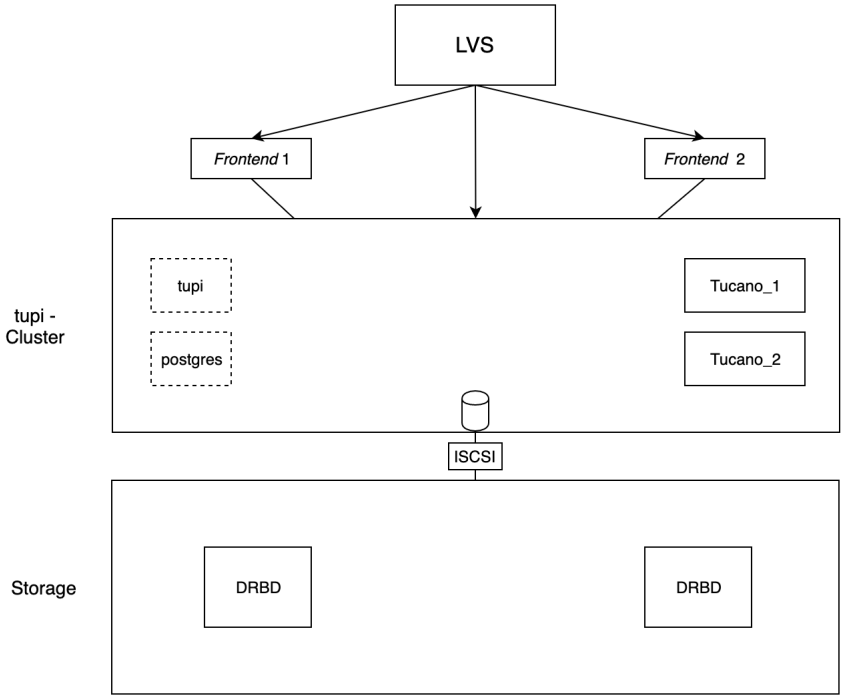

Figura 21: Arquitetura implementada.

3.6 Eliminação de SPOFs

Num sistema de alta disponibilidade é essencial haver tolerância a faltas, pelo que é necessário fazer replicação do *hardware* da infraestrutura para evitar SPOFs *(Single Point of Failure)*.

Como forma de dar resposta à criação de um sistema com elevada disponibilidade, o grupo procedeu à replicação de todos os serviços uma vez, por forma a ter duas réplicas de cada serviço, com a exceção do diretor. Assim, o serviço estará garantido, desde que não ocorra uma falha na camada superior da aplicação. Com isto, todos os outros serviços estarão disponíveis quando uma das máquinas respetivas a um dado serviço falhar, uma vez que a réplica poderá dar respostas. Os resultados serão bastante favoráveis, uma vez que é garantido o serviço e é possível reparar/corrigir as máquinas em falha, sem que se perca capacidade de resposta.

Em suma, o sistema encontra-se preparado para elevada disponibilidade, pelo que só se perdem serviços casos as duas máquinas falhem, ou quando o Diretor não conseguir estar disponível.

3.7 Política de Balanceamento

Para evitar sobrecargas na rede, foi necessário a utilização de uma solução de balanceamento de carga, escolhendo-se assim o *Linux Virtual Server* (LVS). Sendo assim, o LVS foi aplicado no diretor para este poder distribuir os pedidos pelos vários servidores web. Desta forma, obtém-se balanceamento e uma distribuição uniforme dos pedidos pela rede, originando alto desempenho e confiabilidade.

3.8 Desempenho da Plataforma: Testes de Carga

Uma vez implementada a topologia apresentada na Figura 21, voltamos a realizar os mesmos testes de carga já referidos, mas desta vez nesta nova arquitetura de alta disponibilidade e desempenho.

O teste foi realizado utilizando um baixo número de *threads*, devido a erros de *exception out of memory*.

3.8.1 Teste 1: *Login*

O primeiro teste com a nova arquitetura consistiu em efetuar *login* na aplicação, com o auxílio da plataforma *JMeter*, testando assim os dois seguintes *endpoints*:

1. **GET** /
2. **POST** /api/auth/sign_in

   Body data: {email: "admin@tupi.pi", password: "admin123"}

O teste foi realizado utilizando somente 100 e 200 *threads* e os gráficos dos resultados do pedido POST do *login* encontram-se na Figura 23 e 24, respetivamente.

Na tabela seguinte é possível visualizar o *throughput*, a latência e a percentagem de erro do pedido GET e do pedido POST.

Login						
Threads	Throughput GET (por min)	Throughput POST (por min)	Latência GET (ms)	Latência POST (ms)	Erro GET (%)	Erro POST (%)
100	123.63	878.477	45951	6317	0	0
200	157.198	71.22	33735	64099	0	2.3

Figura 22: Métricas Teste 1.

Comparando com o teste presente na Secção 2.2.2, o mesmo teste mas com a arquitetura base do sistema, é suposto obter uma melhoria ligeira uma vez que temos os serviços de forma distribuída. No entanto, a componente da base de dados continua a ser aquela sobre a qual se incide mais.

Como podemos observar na tabela acima apresentada, e comparando com os resultados do teste inicial, relativamente ao *login* apresentam-se resultados realmente superiores em termos de *throughput* e também na latência desse mesmo pedido. Como podemos ver, no caso da simulação com 200 utilizadores, o erro também é ligeiramente inferior (2% vs 4.5%). As melhorias enumeradas podem advir da utilização de um balanceador de carga, e também pelo facto dos recursos das máquinas serem mais eficientes, uma vez que cada componente da arquitetura geral do sistema tem uma máquina dedicada a si.

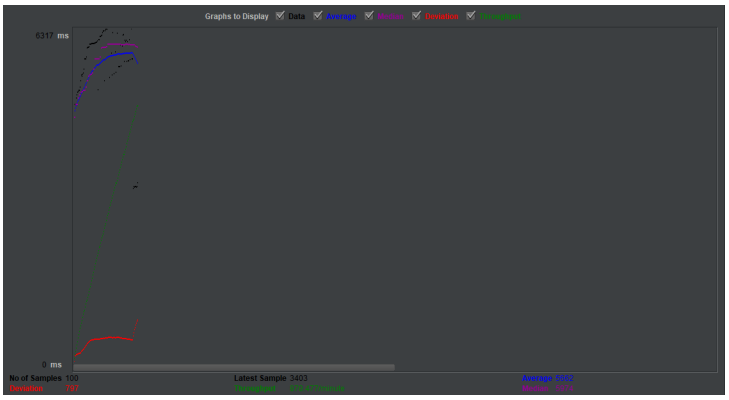

Figura 23: Gráfico dos Resultados do Teste 1 para 100 *threads*.

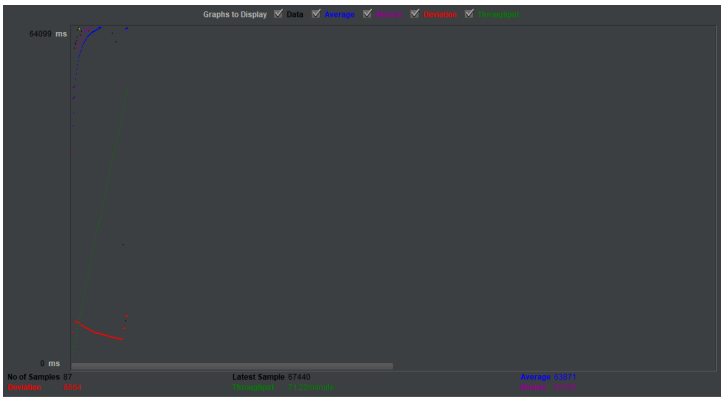

Figura 24: Gráfico dos Resultados do Teste 1 para 200 *threads*.

3.8.2 Teste 2: *Login* + Consulta de *managers*

O segundo teste com a nova topologia teve como objetivo efetuar o *login* na aplicação e consultar a lista de *managers*:

1. **GET** /
2. **POST** /api/auth/sign_in

   Body data: {email: "admin@tupi.pi", password: "admin123"}
3. **GET** /api/v1/managers

O teste foi realizado utilizando somente 100 *threads*. O gráfico de resultados obtido para o pedido GET apresenta-se na Figura 26. Na tabela seguinte é possível visualizar o *throughput*, a latência e a percentagem de erro do pedido POST e do pedido GET.

Login + Consulta managers						
Threads	Throughput (por min)	Throughput GET MAN (por min)	Latência POST (ms)	Latência GET MAN (ms)	Erro POST (%)	Erro GET MAN (%)
100	421.082	998.336	8243	95	0	0

Figura 25: Métricas Teste 2.

Tendo em conta o cenário de simulação de 100 utilizadores, e refletindo sobre os resultados que obtivemos neste mesmo teste com a arquitetura base (Secção 2.2.3), podemos afirmar de forma inequívoca melhorias consideráveis, quer em termos de débito, quer a nível de tempo de resposta (latência). Tal como é observável, o *throughput* obtido na arquitetura distribuída é muito superior: 107.064 *Login* + 109.822 *GetManagers* contra 421.082 *Login* + 998.336 *GetManagers*. Assim, parece-nos que é bastante evidente as melhorias que esta arquitetura conseguiu providenciar, tendo em conta este caso em específico. Relativamente ao erro, o grupo já esperava que fosse nulo (0%) em ambos os casos, uma vez que esse havia sido o indicativo em todos os testes realizados até então com 100 utilizadores.

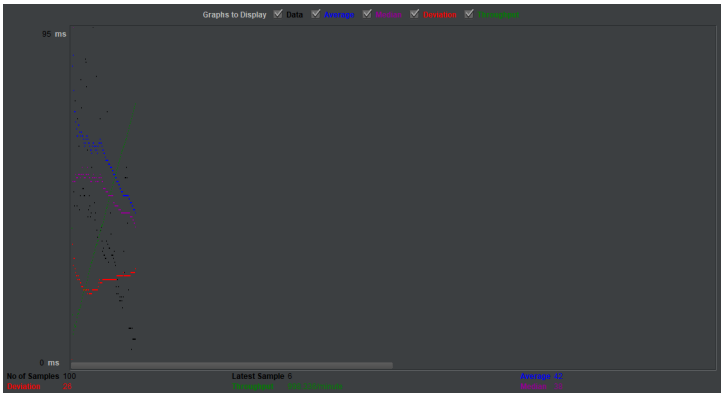

Figura 26: Gráfico dos Resultados do Teste 2 para 100 *threads*.

3.8.3 Teste 3: *Login* + Inserção de *manager* + Consulta de *managers*

O terceiro teste consistiu na sequência de três passos: *login* na aplicação por parte do administrador, inserção de um *manager* e, por fim, consulta da lista de todos *managers*.

Os gráficos dos resultados do pedido POST dos *managers* para as simulações com 100 e 200 *threads* encontram-se na Figura 28 e 29, respetivamente.

Na tabela seguinte é possível visualizar o *throughput* e a latência do pedido POST do *login*, do pedido POST da inserção do *manager* e do pedido GET da lista de *managers*, e a percentagem de erro obtida no pedido POST do *login* para cada uma das simulações.

Login + Inserção Manager + Consulta Managers							
Threads	Throughput Login (por min)	Throughput Add MAN (por min)	Throughput Get MAN (por min)	Latência Login (ms)	Latência Add MAN (ms)	Latência Get MAN (ms)	Erro Login (%)
100	199.99	125.052	179.455	26141	10634	26731	0
200	196.609	104.031	175.467	48186	44164	94545	0
300	289.245	203.56	165.913	60224	60027	176209	48

Figura 27: Métricas Teste 3.

Comparando os resultados obtidos num teste idêntico mas utilizando a arquitetura base da infraestrutura (Secção 2.2.4) é possível verificar novamente que os valores do *throughput* e da latência foram superiores em geral, podendo-se assim concluir que a nova arquitetura fornece um melhor desempenho à plataforma.

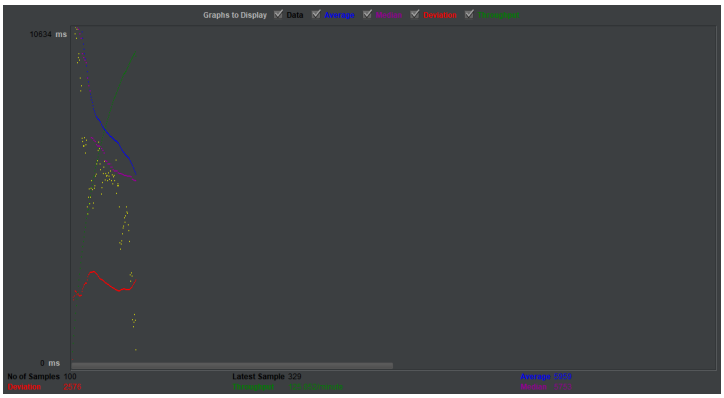

Figura 28: Gráfico dos Resultados do pedido POST *managers* do Teste 3 para 100 *threads*.

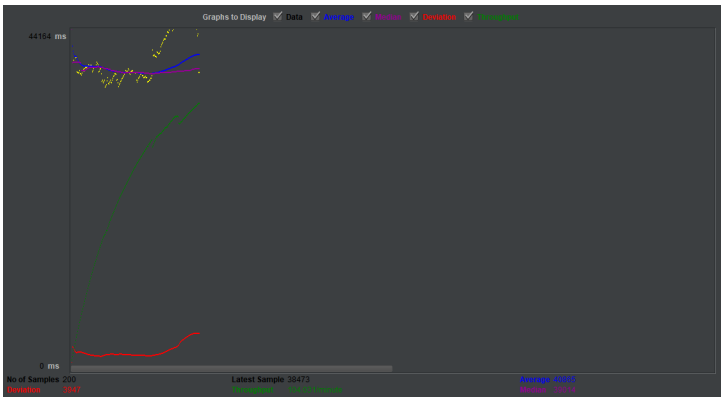

Figura 29: Gráfico dos Resultados do pedido POST *managers* do Teste 3 para 200 *threads*.

3.8.4 Teste 4: *Login User* + Candidatar a um concurso

O quarto e último teste, tal como descrito na Secção 2.2.5, tem como objetivo um *user* se candidatar a um concurso ativo. Este teste foi efetuado utilizando 100 e 200 *threads*.

Na tabela seguinte é possível visualizar o *throughput* e a latência do pedido POST e do pedido PATCH, e a percentagem de erro no pedido POST para cada uma das simulações.

Login + Realizar Concurso					
Threads	Throughput Login (por min)	Throughput PATCH (por min)	Latência Login (ms)	Latência PATCH (ms)	Erro Login (%)
100	99.777	94.329	52908	24	0
200	177.736	169.033	60515	25	15

Figura 30: Métricas Teste 4.

Com este último teste verifica-se novamente uma melhoria nos valores de débito e latência e uma diminuição do valor da percentagem obtida (de 25% para 15%), tal como seria esperado.

3.9 Disponibilidade da Plataforma

Neste ponto, tendo uma arquitetura distribuída, é de esperar um aumento de disponibilidade da aplicação pelo menos em termos de desempenho, como foi mostrado anteriormente através da comparação de resultados dos testes de carga.

Em análise de tudo que foi dito anteriormente é possível afirmar que o nosso sistema garante alta disponibilidade.

Apesar de existirem algumas complicações com as rotas, pois, em termos de redes existem algumas lacunas, devido à nossa inexperiência. Lacunas essas que através de alguma manipulação de rotas, conseguiram ser colmatadas. De maneira a tornar esse processo automático criamos um *Script* (B), que é executando quando o *cluster* atribui uma máquina para a aplicação, e que faz as alterações necessárias ás rotas.

Assim, apenas existem 4 casos em que o nosso sistema não apresenta o serviço para que foi construído, sendo estes casos altamente improváveis e por isso só acontecem em condições especiais:

- `Caso 1:` Se ambas as máquinas que estão ligadas a *storage* falharem, perdemos os nossos dados.
- `Caso 2:` Perdemos ligação à aplicação, caso ambas as máquinas que constituem o *cluster* falhem ao mesmo tempo.
- `Caso 3:` Em caso de falha de ambos os *Web Server*, perdemos o *frontend* da aplicação.
- `Caso 4:` Se o nosso *LVS* falhar, não será possível reencaminhar o tráfego para a nossa aplicação.

Em modo de síntese, o nosso sistema ficará indisponível caso falhem as duas máquinas de cada componente. Como dito anteriormente, estes são casos que acontecem em condições especiais, podendo assim afirmar que a nossa arquitetura garante alta disponibilidade.

4 Conclusão

O planeamento de uma infraestrutura é uma parte fundamental para o bom funcionamento de uma aplicação, de modo a que esta consiga responder com sucesso às exigências do utilizador. Neste projeto, fomos capazes de analisar uma aplicação já existente, assim como todos os seus componentes, e planear uma topologia que promove alta disponibilidade e escalabilidade.

Ao longo do desenvolvimento do presente trabalho prático foram várias as dificuldades sentidas. A primeira dificuldade prendeu-se no *deploy* da aplicação Tucano na máquina virtual, tendo em consideração que nenhum dos membros do grupo tinha trabalhado com *Elixir*, nem está habituado a fazer *deploy* a aplicações em geral. A segunda dificuldade esteve relacionada com a escolha da ferramenta a utilizar para os testes de carga e na respetiva utilização, pois foi também a nossa estreia nessa componente aplicacional. Neste âmbito, foram realizadas várias pesquisas com o intuito de encontrar as melhores soluções que permitissem concretizar os objetivos.

Por fim, a implementação de uma topologia de elevada disponibilidade teve também bastantes contratempos, nomeadamente na descoberta de, já durante a implementação da arquitetura, que não existe comunicação entre os *webservers* e o *backend*, sendo o próprio *browser* que comunica diretamente com o *backend*. Sentimos também algumas dificuldades no que toca às redes, porém, com alguma reflexão e pesquisa conseguimos superar essas adversidades.

Estes foram apenas algumas das dificuldades sentidas, mas que foram colmatadas com sucesso.

A topologia implementada oferece assim uma melhoria bastante satisfatória à aplicação, permitindo que esta responda com mais rapidez e exatidão aos pedidos dos utilizadores. Através da análise dos resultados obtidos nos testes de carga com a arquitetura base do sistema e com a arquitetura implementada, é possível constatar que as percentagens de erro realmente diminuíram, podendo assim inferir que a nova infraestrutura permite que mais utilizadores naveguem na aplicação em simultâneo, com uma taxa de sucesso bastante superior. Também no que diz respeito ao débito e à latência dos pedidos dos utilizadores, os valores obtidos são superiores aos obtidos com a arquitetura base, o que nos leva a concluir que de facto a utilização de uma topologia distribuída fornece uma melhoria notória no desempenho da aplicação.

A Código Teste 0

```
import org.openqa.selenium.By;
import org.openqa.selenium.Keys;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.chrome.ChromeOptions;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.Random;

public class Tester extends Thread{

    public WebDriver browser;
    public double[] tempos;
    public int indice;

    public Tester (WebDriver browser,double[] tempos,int i){
        this.browser = browser;
        this.tempos=tempos;
        this.indice=i;
        System.out.println();
    }

    public void run(){
        //WebDriver browser = new ChromeDriver();
        long inicio = System.currentTimeMillis();
        browser.get("http://172.20.10.4:8080/login");

        //Try para esperar que a pagina seja carregada
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        //Exemplo do Tucano
        WebElement textbox1 = browser.findElement(By.name("email"));
        textbox1.sendKeys("admin@tupi.pi");
        WebElement textbox2 = browser.findElement(By.name("password"));
        textbox2.sendKeys("admin123");
        textbox2.sendKeys(Keys.ENTER);
```

```
    long inicioPedido = System.currentTimeMillis();

    //Try para esperar que a pagina seja carregada
    try {
        Thread.sleep(2000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    browser.get("http://172.20.10.4:8080/admin/managers/new");
    WebElement emailM = browser.findElement(By.name("email"));
    emailM.sendKeys("antonio@tupi.pi");
    WebElement nomeM = browser.findElement(By.name("name"));
    nomeM.sendKeys("Antonio Nobre");
    nomeM.sendKeys(Keys.ENTER);

    try {
        Thread.sleep(2000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    browser.get("http://172.20.10.4:8080/admin/managers/");
    long fim = System.currentTimeMillis();
    double duracao_interacao = (fim - inicio - 2000-2000)/1000.00 ;
    tempos[indice] = duracao_interacao;
    System.out.println("Interação durou " + duracao_interacao + " s");
    browser.quit();
}

public static void main(String[] args) throws Exception {
    int numT = 16;
    ChromeOptions option = new ChromeOptions();
    option.addArguments("headless");
    option.addArguments("window-size=1920x1080");
    Tester[] r = new Tester[numT];
    double[] tempos = new double[numT];
    double sum = 0;

    for(int i=0; i<r.length; i++) {
        r[i] = new Tester(new ChromeDriver(option),tempos,i);
    }

    for(int i=0; i<r.length; i++)
        r[i].start();
    for(int i=0; i<r.length; i++)
        r[i].join();
```

```
        for(int k=0;k<r.length;k++){
            sum+=tempos[k];
        }

        System.out.println("Media dos tempos: " + sum/numT);

    }
}
```

B Script *Bash* alteração de rotas

```
#!/bin/bash
#Script de Preparação de rotas para correr o tucano num Cluster de HA.

#Começamos por retirar a rota atual por defeito da placa NAT
route del default

#Adicionamos uma rota para comunicar com drbd1
route add -net 192.168.111.138 netmask 255.255.255.255 gw 192.168.111.138

#Adicionamos uma rota para comunicar com drbd2
route add -net 192.168.111.142 netmask 255.255.255.255 gw 192.168.111.142

#Removemos a rota da rede dos DRBD's por ser esta a responsável
# pela interferencia com o exterior
route del -net 192.168.111.0 netmask 255.255.255.0

# Adicionamos a nova rota por defeito, que é o caminho para o LVS.
route add -net 0.0.0.0 netmask 0.0.0.0 gw 10.10.100.1
```