UNIVERSIDADE DO MINHO
ESCOLA DE ENGENHARIA

Escalabilidade e Telemetria de Infraestruturas

Ana Marta Santos Ribeiro A82474
Jéssica Andreia Fernandes Lemos A82061
Miguel José Dias Pereira A78912

MIEI - 4º Ano - Infraestruturas de Centro de Dados

Braga, 24 de Outubro de 2020

Conteúdo

1 Introdução

Para obtermos uma alta disponibilidade de um serviço numa infraestrutura é essencial o estudo e a análise do mesmo. Assim, é possível escalar o serviço em caso de alteração da carga do sistema e ainda podemos eliminar pontos de falhas únicos.

O objetivo deste projeto consiste em analisar a plataforma *Tucano* que é utilizada para realização da distribuição de alunos por opções, baseada na escolha destes.

Com o intuito de caracterizar o processo de desenvolvimento do sistema começamos por analisar o problema em questão bem como todos os seus componentes. Posteriormente, apresentamos e explicamos a arquitetura definida cujo principal objetivo é fornecer um serviço escalável, sem pontos de falha únicos e que seja tolerante a falhas. De seguida, expomos o processo de implementação da infraestrutura. Por fim, de modo a avaliar o desempenho do *Tucano*, exibimos os resultados dos testes de carga antes e depois da implementação da arquitetura.

2 Análise dos Componentes da Infraestrutura

O objetivo deste projeto consiste em avaliar o desempenho da aplicação **Tucano**, que é a plataforma utilizada para a realização da distribuição de alunos por opções, baseada na escolha destes. Desta forma, é necessário implementar uma estrutura com o intuito de realizarmos testes de carga anteriormente e posteriormente ao seu desenvolvimento para que seja possível comparar os resultados obtidos.

Esta aplicação é constituída por 2 componentes:

- *backend*
- *frontend*

Para o armazenamento dos dados desta aplicação é utilizado um Sistema de Gestão de Base de Dados *PostgreSQL*.

Desta forma, na secção seguinte iremos apresentar a estrutura que permite a elevada escalabilidade e disponibilidade do serviço.

3 Arquitetura do Sistema

Com o intuito de realizar o planeamento da estrutura é fundamental definir a arquitetura do sistema bem como analisar as características pretendidas para o serviço. Desta forma, iremos ter em atenção a resiliência da infraestrutura quanto à tolerância a falhas, a escalabilidade, bem como a identificação e eliminação dos SPOFs.

Assim sendo, optamos por projetar a arquitetura em várias camadas, que serão descritas em secções posteriores. A camada de apresentação é constituída por dois servidores web onde se encontram os serviços de *frontend* responsáveis por disponibilizar a interface ao utilizador. O balanceamento da carga é realizado recorrendo aos LVS. De modo a disponibilizar os serviços de *backend* e *Postgres* implementamos um *High Availability Cluster*. Para realizar o armazenamento dos dados recorremos DRDBs que em conjunto com o protocolo *iSCSI* torna possível o acesso pela rede.

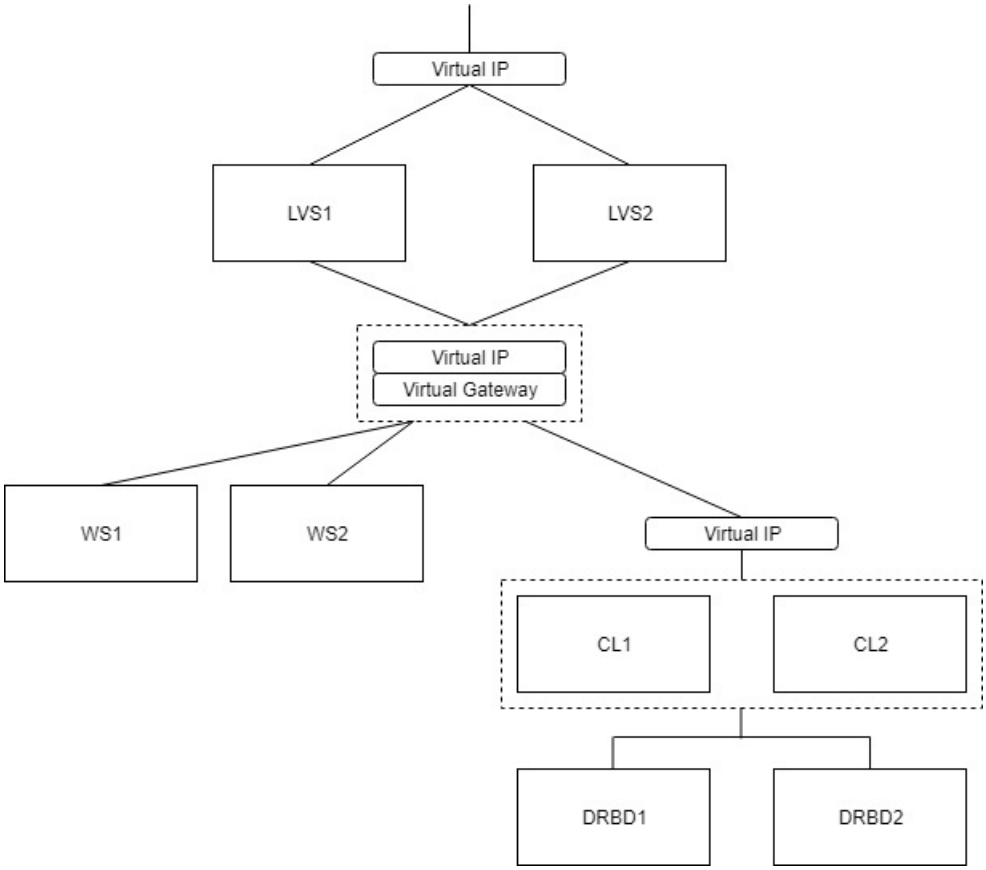

Figura 1. Arquitetura do Sistema

4 Web Servers

Nesta camada é onde se encontra o *frontend* da aplicação, e assim sendo é um ponto crucial, pois todas as interações do utilizador com as páginas web são dirigidas ao *frontend*.

Como tal, esta camada foi pensada para ser facilmente escalável, de modo a prevenir eventuais picos de carga, uma vez que para adicionar um novo *web server* apenas temos de alterar a configuração do *keepalived* no balanceador de carga.

Para configurar o *frontend* da aplicação nesta arquitetura, apenas é necessário definir o endereço relativo ao *backend* no ficheiro *.env*, que corresponde ao IP Virtual definido para o balanceador de carga na configuração do *keepalived*:

```
1 REACT_APP_API=http://192.168.6.200:4001/api
```

5 Balanceamento de Carga

O balanceamento de carga é realizado recorrendo a um LVS, que recebe as conexões dos utilizadores da aplicação, e redireciona os pedidos consoante a porta nos quais os recebe.

Com apenas um LVS estaríamos perante um eventual SPOF, e assim sendo fazemos uso de dois LVS em *failover*, um *master*, que fica encarregue do balanceamento/encaminhamento, e um *backup*, que fica em *standby* até que o outro falhe.

Para obtermos esta configuração fazemos uso do protocolo *VRRP* e do software *Keepalived*, que fazem a gestão dos IPs Virtuais e o balanceamento de carga recorrendo ao algoritmo *round-robin*. A utilização do *Keepalived* permite inserir *web servers* adicionais simplesmente alterando a configuração do *Keepalived*, caso seja necessário aumentar a performance dos mesmos.

Obtemos então a seguinte tabela de servidores virtuais com a configuração do *Keepalived*:

```
[root@lvs1 ~]# ipvsadm -ln
IP Virtual Server version 1.2.1 (size=4096)
Prot LocalAddress:Port Scheduler Flags
  -> RemoteAddress:Port           Forward Weight ActiveConn InActConn
TCP  192.168.6.200:3000 rr
  -> 10.10.10.10:3000             Masq    1      0          0
  -> 10.10.10.11:3000             Masq    1      0          0
TCP  192.168.6.200:4001 rr
  -> 10.10.10.50:4001             Masq    1      0          0
```

Figura 2. Servidores virtuais

Temos assim os redirecionamentos dos pedidos na porta 3000 para os *web servers*, *frontend*, e na porta 4001 para os *clusters*, *backend*.

6 Storage

Para o armazenamento da informação produzida pelo *backend* da aplicação utilizamos o serviço de replicação de dados *DRBD*, utilizando duas máquinas em modo multi-primário. A utilização do protocolo *iSCSI multipath* nos *DRBD*, permite generalizar o acesso aos mesmos, e como funciona em *failover*, na eventualidade de um dos componentes ou caminhos falhar, automaticamente seleciona outro pelo qual enviar a informação.

7 Cluster

Nesta camada construímos um *High Availability Cluster*, constituído por dois *nodes*. Como referido na secção anterior utilizamos o protocolo *iSCSI multipath* para aceder aos *DRBDs*:

```
[root@cl1 ~]# multipath -ll
mpatha (360014057c96ae183918409a844ef1d63) dm-2 LIO-ORG ,d1
size=1024M features='0' hwhandler='0' wp=rw
|-+- policy='service-time 0' prio=1 status=active
| `- 3:0:0:0 sdc 8:32 active ready running
`-+- policy='service-time 0' prio=1 status=enabled
  `- 4:0:0:0 sdb 8:16 active ready running
[root@cl1 ~]# lsscsi
[0:0:0:0]    disk    VMware,  VMware Virtual S 1.0   /dev/sda
[2:0:0:0]    cd/dvd  NECVMWar VMware IDE CDR10 1.00  /dev/sr0
[3:0:0:0]    disk    LIO-ORG  d1               4.0   /dev/sdc
[4:0:0:0]    disk    LIO-ORG  d1               4.0   /dev/sdb
```

Figura 3. iSCSI multipath

De seguida foram criadas 2 partições sendo uma para a base de dados e outra para o *backend*. Outra opção seria ter apenas 1 partição para a base de dados e o *backend* encontrando-se localmente em cada máquina. Contudo a solução adotada permite uma maior escalabilidade, pois qualquer alteração que seja efetuada nas configurações do *backend* não terá de ser repetida em cada máquina.

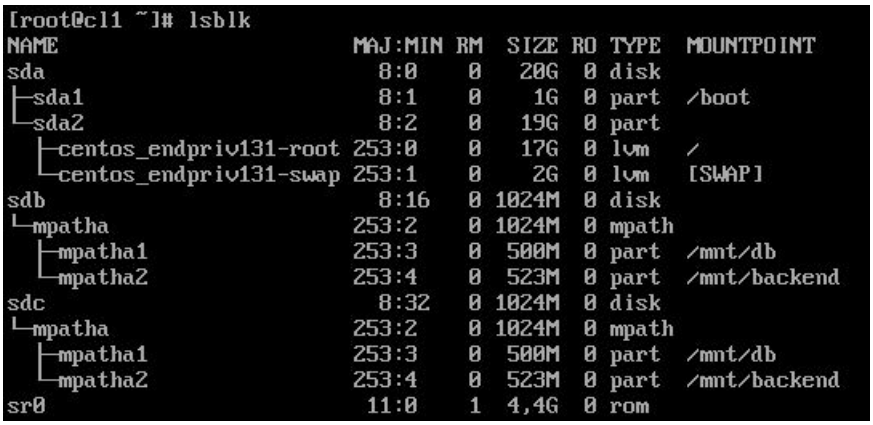

Figura 4. Partições

Depois de configurados a base de dados, *PostgreSQL*, e o *backend*, em cada uma das partições, procedemos à configuração dos recursos dos *clusters* através da interface web do *Pacemaker*.

No primeiro grupo, *db*, temos os recursos relacionados com a base de dados: IP Virtual, *filesystem* e *PostgresSQL*.

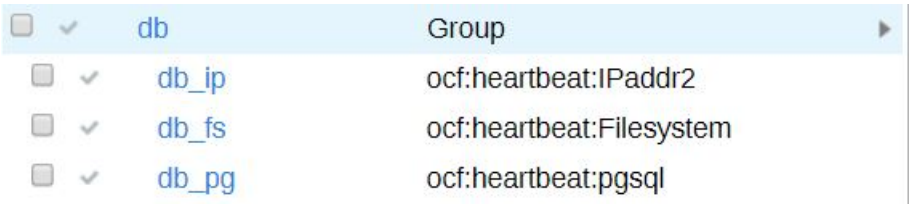

Figura 5. Recursos base de dados

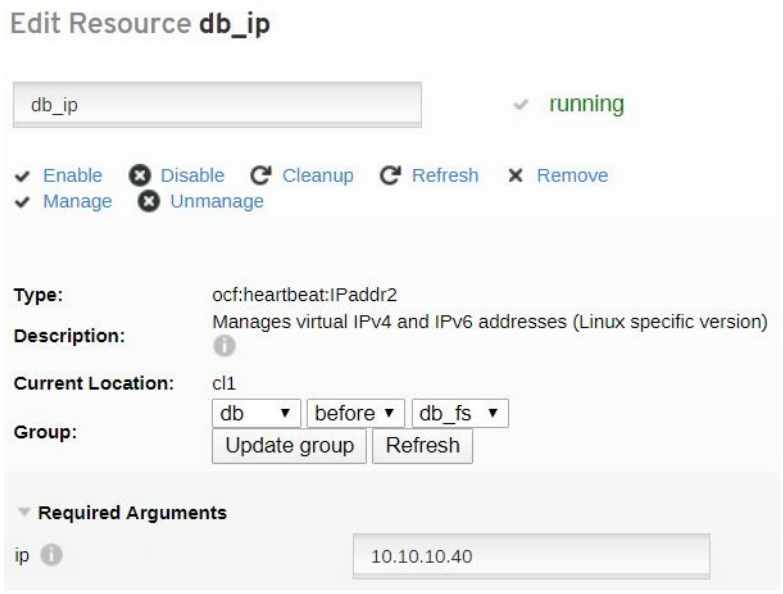

Figura 6. IP Virtual

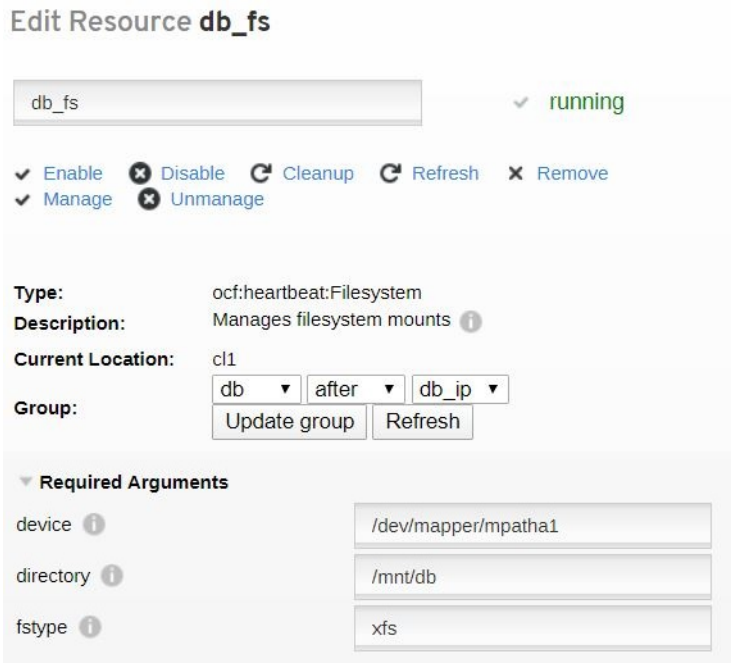

Figura 7. Filesystem

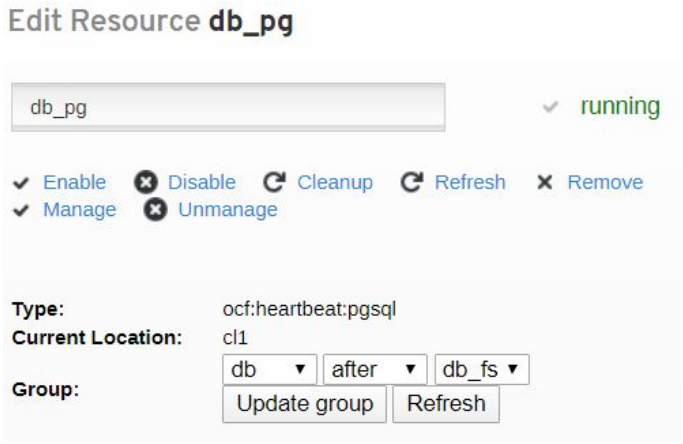

Figura 8. PostgreSQL

No segundo grupo, *tupi*, temos os recursos relacionados com o *backend*: IP Virtual, *filesystem* e *myserv*.

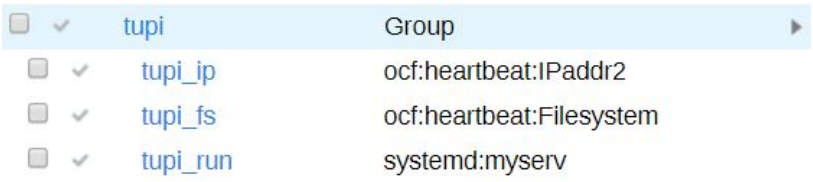

Figura 9. Recursos backend

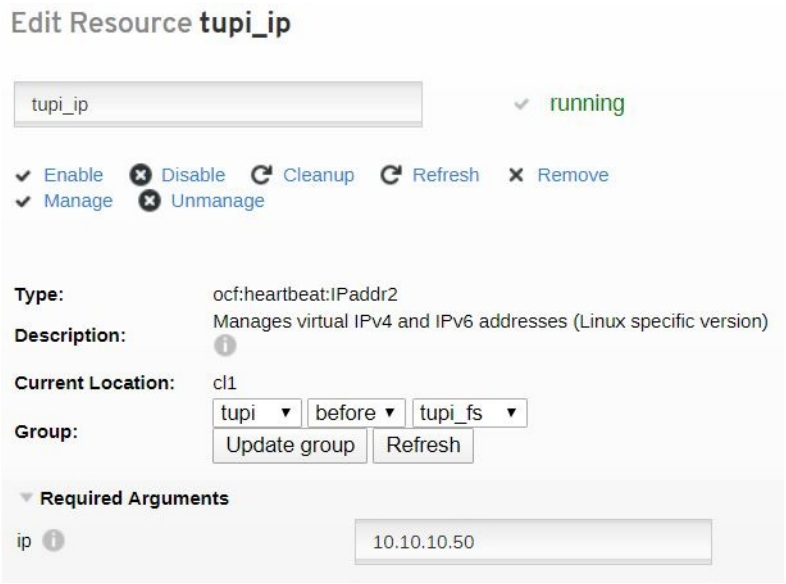

Figura 10. IP Virtual

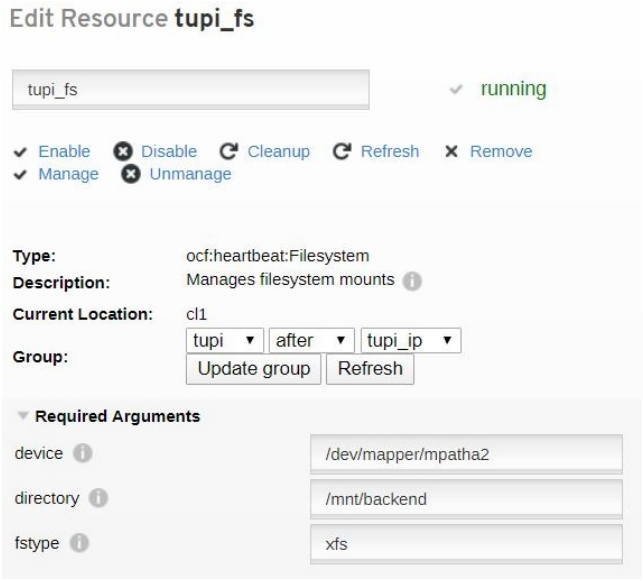

Figura 11. Filesystem

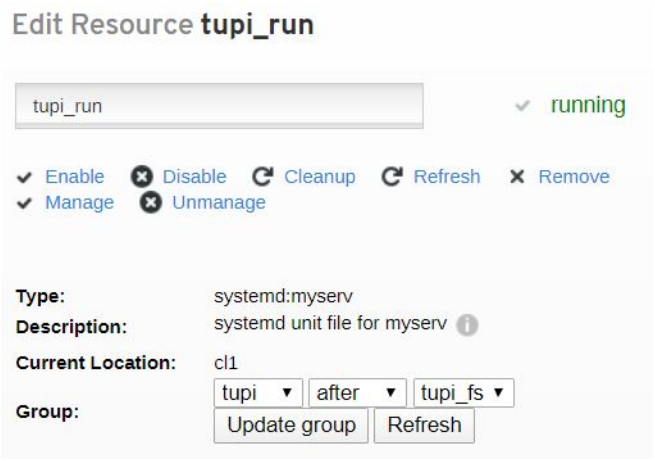

Figura 12. Myserv

Este último é um serviço criado para correr o servidor do *backend* na porta 4001, sendo que o *script* é o seguinte:

```
#!/bin/bash


if [ -d /mnt/backend/tupi/ ]; then
        cd /mnt/backend/tupi/
        echo "export PATH=$PATH:/opt/elixir/bin" >> /home/$USER/.bashrc
        source /home/$USER/.bashrc
        echo $PATH
        export HOME=/
        mix do local.hex --force
        MIX_ENV=prod PORT=4001  elixir -S mix do compile, phx.server
fi
```

Figura 13. Myserv script

Finalmente temos os dois *clusters*, cl1 e cl2, prontos a correr os serviços:

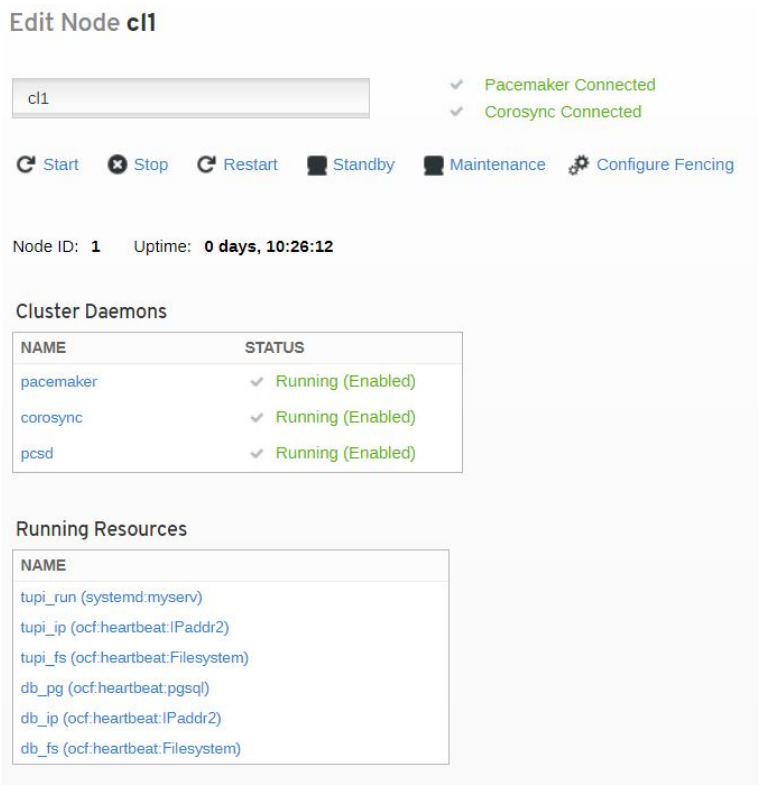

Figura 14. Cl1

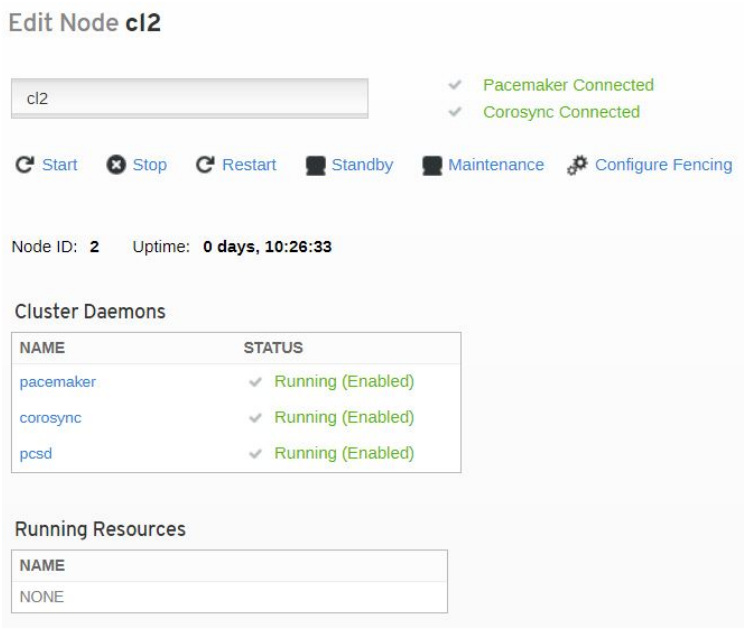

Figura 15. Cl2

8 Testes de carga

Para realizarmos testes de carga ao **Tucano** recorremos à ferramenta *Apache JMeter*, uma das mais utilizadas para medir a performance de um sistema e que oferece uma interface gráfica para a criação, execução e análise de testes.

De forma a avaliar o desempenho da aplicação optamos por realizar três testes, nomeadamente, ao login por parte dos alunos, o processo da candidatura por parte dos mesmos e ainda a obtenção dos resultados do *contest* e o envio dos mesmos.

Para a apresentação dos resultados optamos por apresentar os gráficos resultantes antes e depois da implementação da arquitetura do sistema.

8.1 Login

Este teste é bastante simples em que apenas realizamos o processo de autenticação com 30 alunos. Para tal, no *JMeter*, utilizamos uma thread para cada um.

Através da análise de ambos os gráficos *Transactions per second*, na **Figura 16** e **Figura 17**, é possível verificar que têm um comportamento semelhante embora o número de transações por segundo sem a arquitetura varia entre 4 e 5 enquanto que com a arquitetura entre 3 e 4. Pelo que podemos concluir que com a arquitetura existe, em média, um menor número de transações durante o *login*.

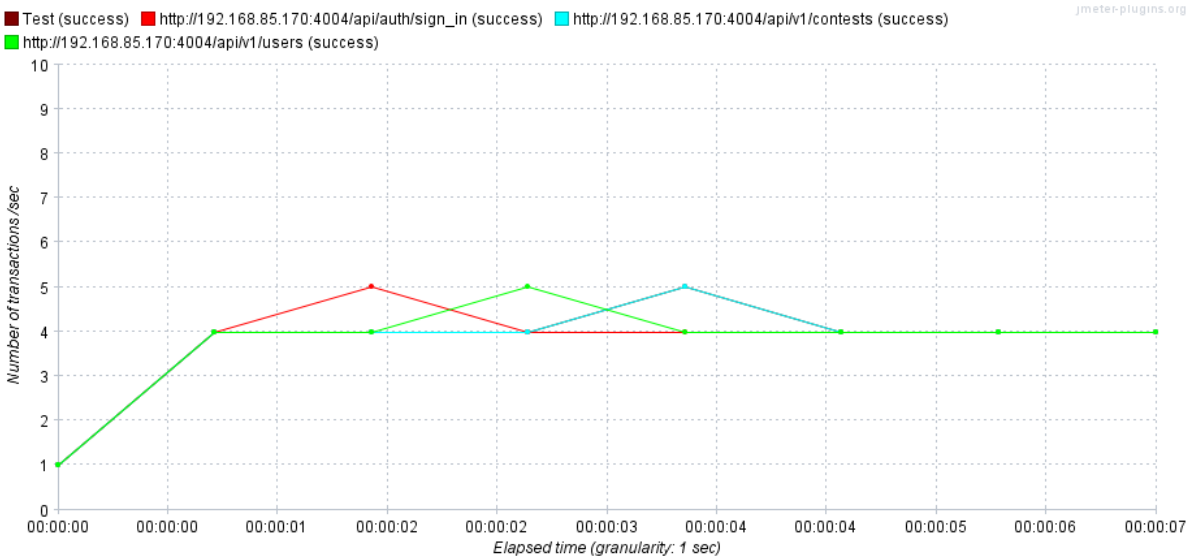

Figura 16. Transactions per second sem arquitetura

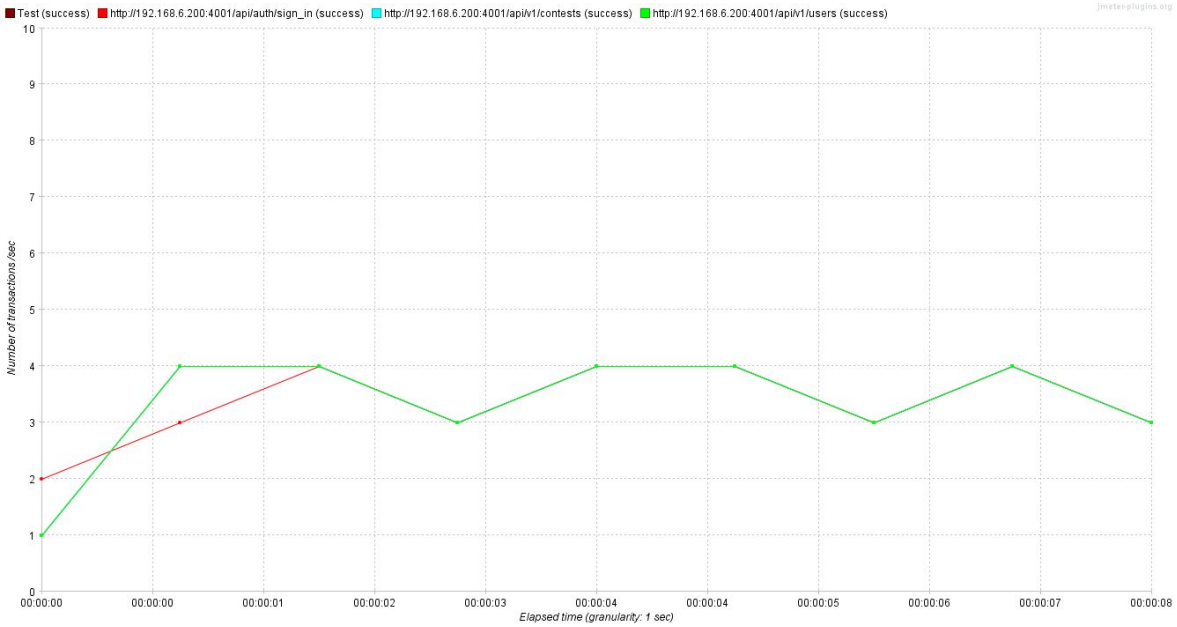

Figura 17. Transactions per second com arquitetura

Como podemos verificar nos gráficos das **Figura 18** e **Figura 19**, a latência da resposta ao longo do tempo vai aumentando, sendo que a latência verificada antes da implementação da arquitetura é menor que depois da mesma para os mesmos intervalos de tempo.

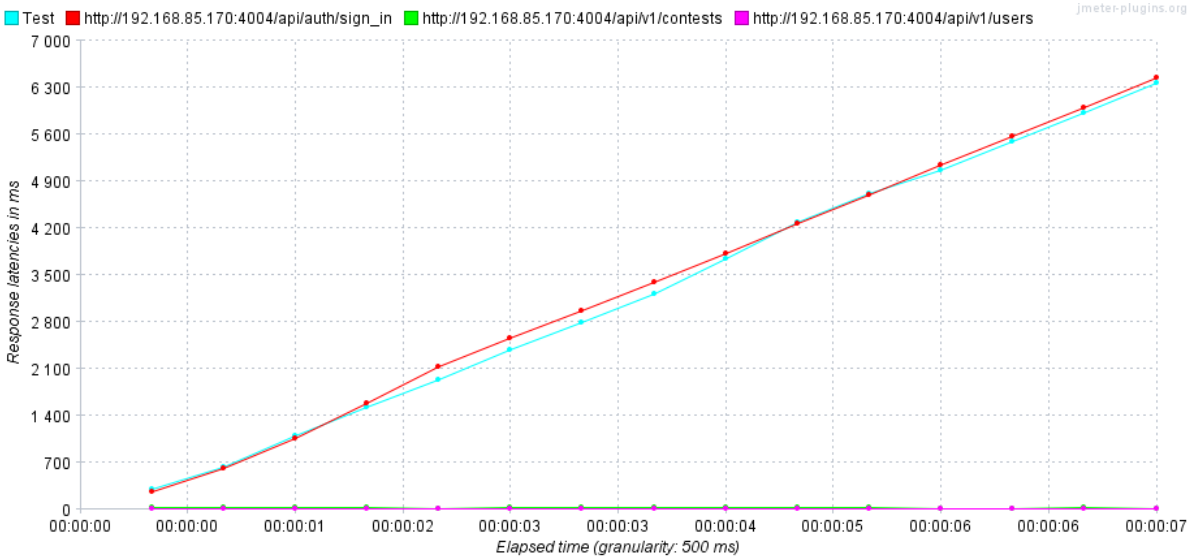

Figura 18. Response Latencies Over Time sem arquitetura

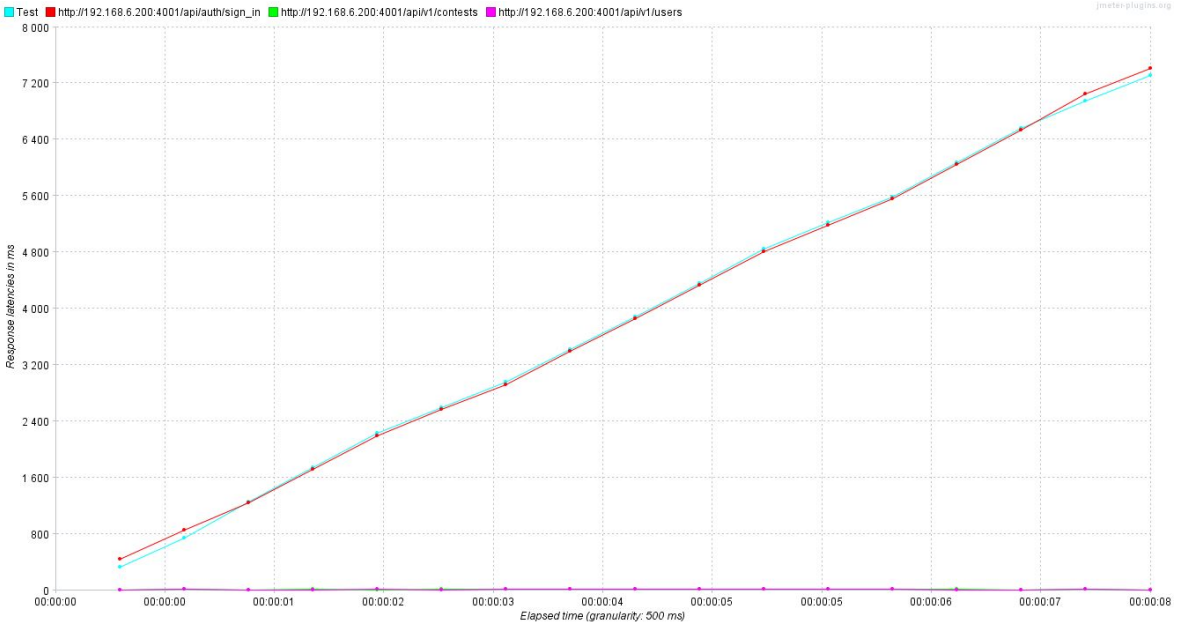

Figura 19. Response Latencies Over Time com arquitetura

Os gráficos apresentados na **Figura 20** e **Figura 21** representam o tempo de resposta em função do número de threads. Estes permitem identificar quando a aplicação passa a ter uma performance inferior ao esperado, ou seja, o tempo de resposta para o número de utilizadores começa a aumentar. Pela observação de ambos os gráficos verificamos que apresentam um comportamento semelhante, o que se reflete em tempos médios também muito idênticos.

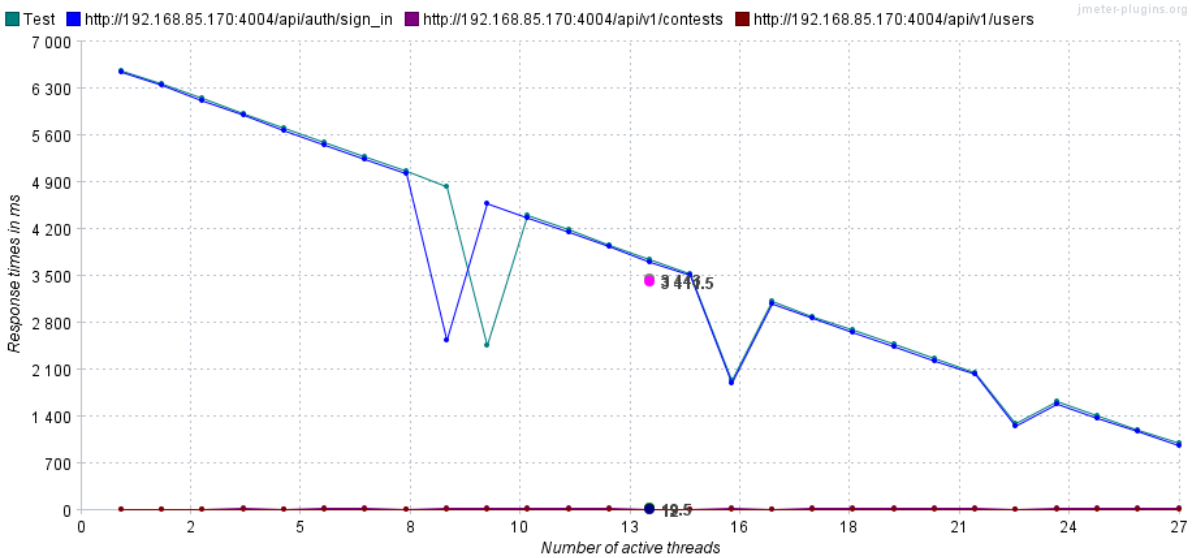

Figura 20. Response Time vs Threads sem arquitetura

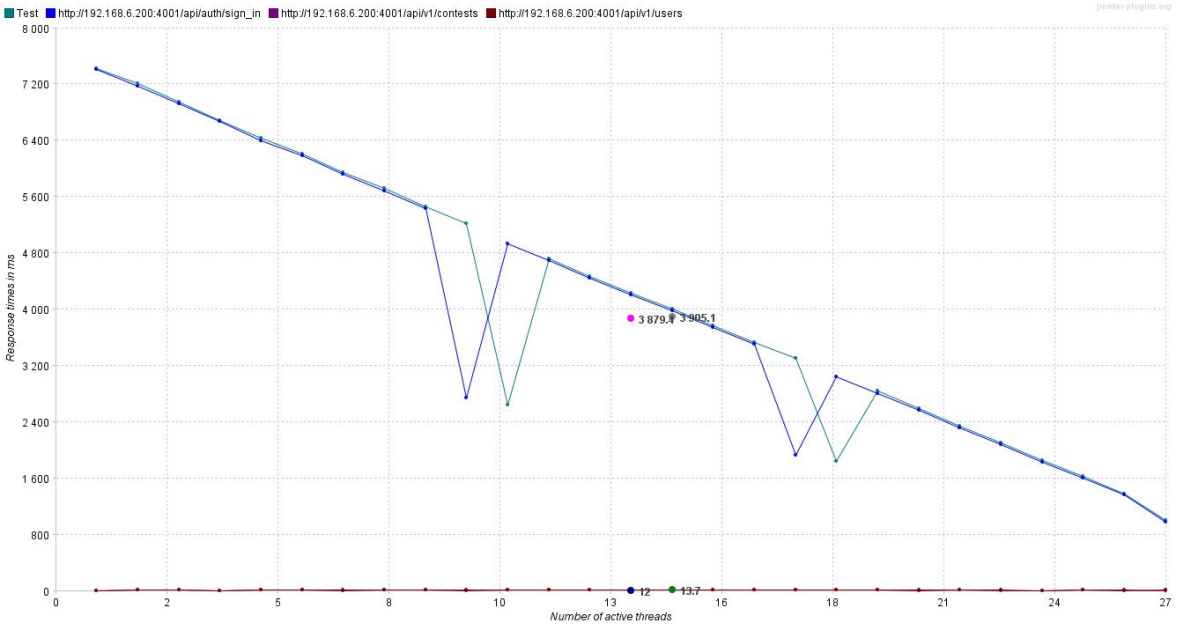

Figura 21. Response Time vs Threads com arquitetura

Na **Figura 22** e **Figura 23** conseguimos visualizar o número de transações por número de threads. Nos dois gráficos é possível constatar que o número tem tendência a aumentar com o número de threads. Contudo, com a implementação da arquitetura o número de transações é consideravelmente superior.

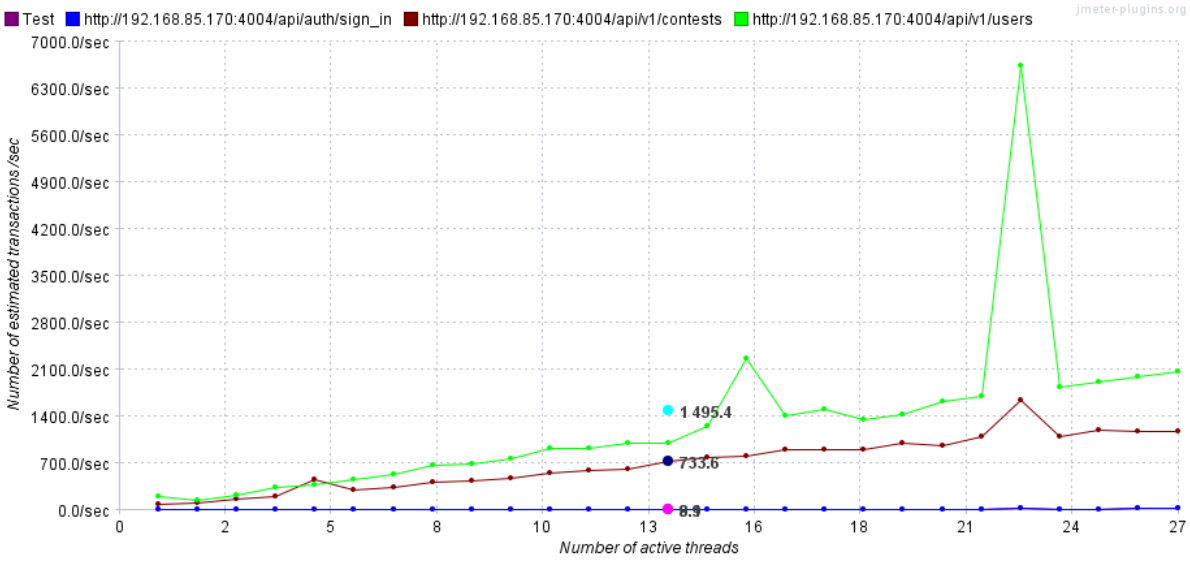

Figura 22. Transaction Throughput vs Threads sem arquitetura

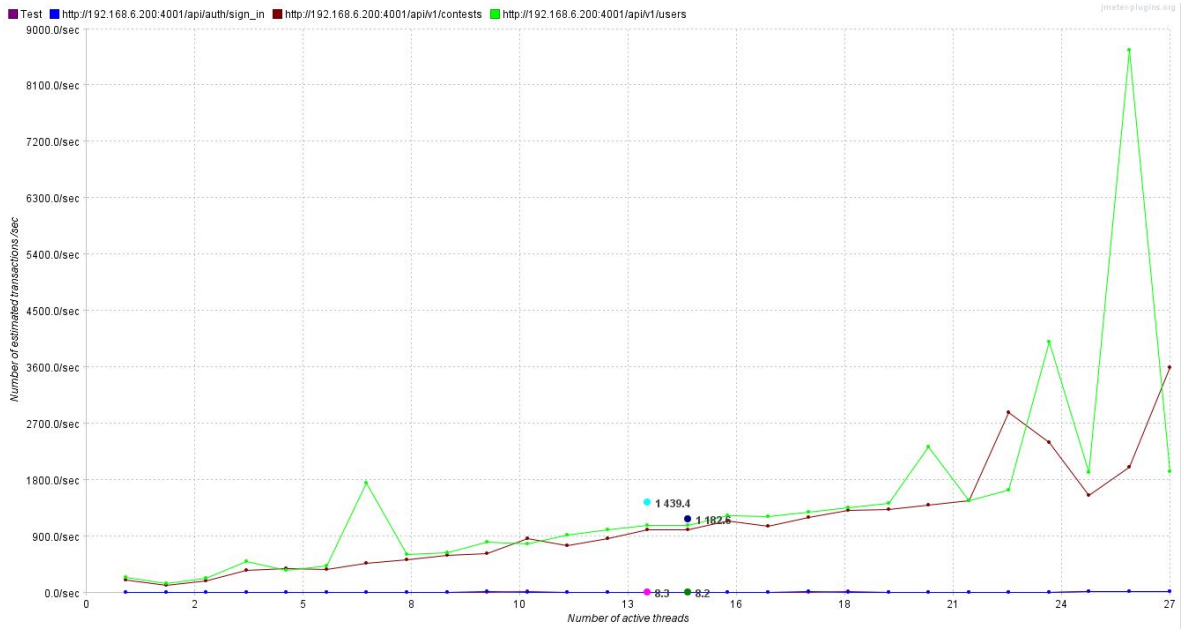

Figura 23. Transaction Throughput vs Threads com arquitetura

8.2 Candidatura

Este já é um processo mais complexo que engloba uma sequência de passos por parte do aluno, nomeadamente, o login, a escolha do *contest* pretendido e guardar uma *application*. Sendo que neste teste mantivemos o número de alunos e o número de threads, ou seja, 30.

Através do gráfico da **Figura 24** e **Figura 25** conseguimos perceber o número de transações efetuadas por segundo das várias operações que envolvem a sequência definida. Tal como para o teste do login, o número de transações é inferior com implementação da arquitetura, embora a diferença não seja tão significativa.

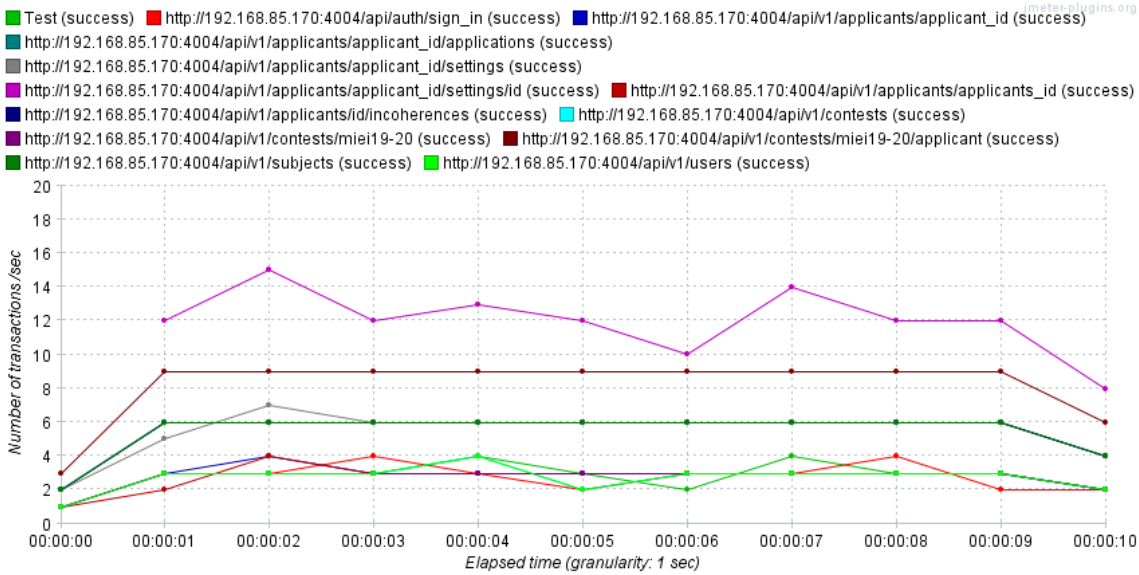

Figura 24. Transactions per second sem arquitetura

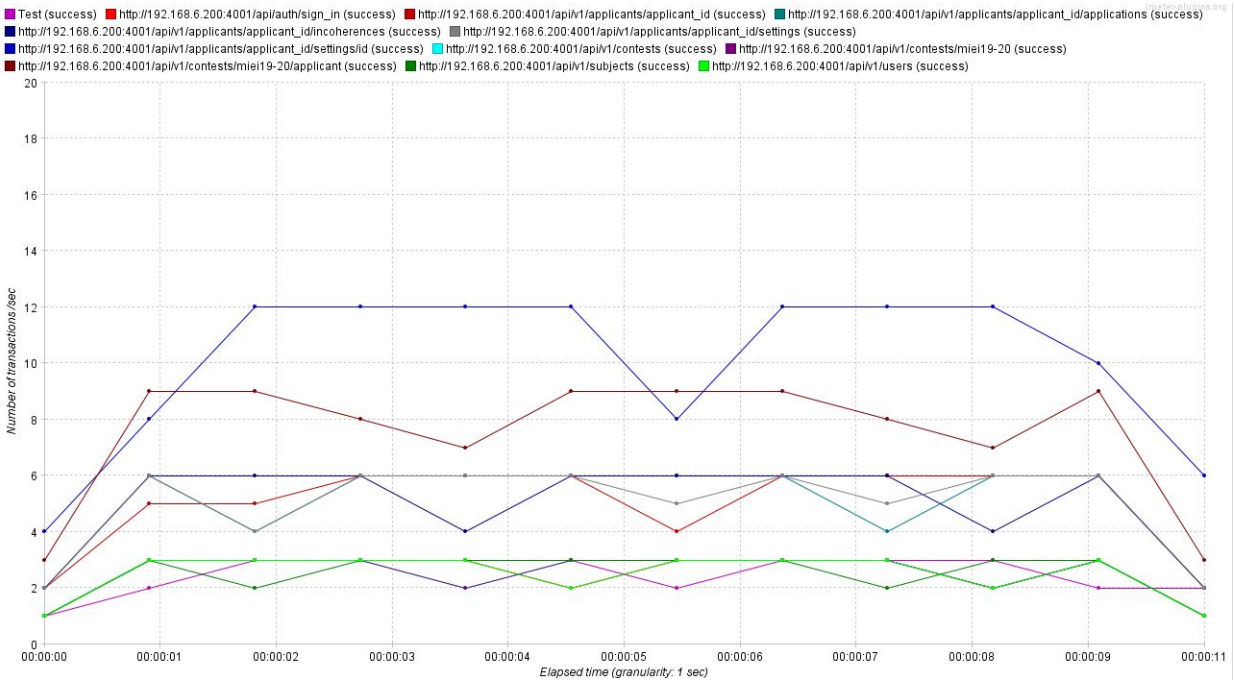

Figura 25. Transactions per second com arquitetura

Este teste da candidatura é constituído por uma sequências de operações, pelo que os gráficos ilustrados na **Figura 26** e **Figura 27** apresentam a latência destes. Assim sendo, verificamos que de um modo geral o comportamento é semelhante, embora para algumas operações, como por exemplo para as *subjects* verifica-se uma maior variação.

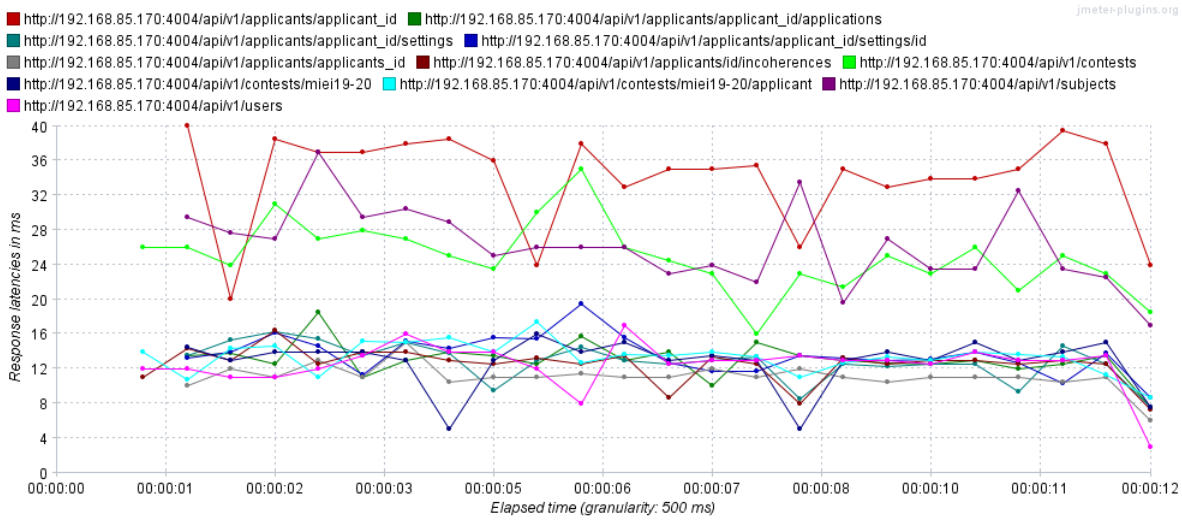

Figura 26. Response Latencies Over Time sem arquitetura

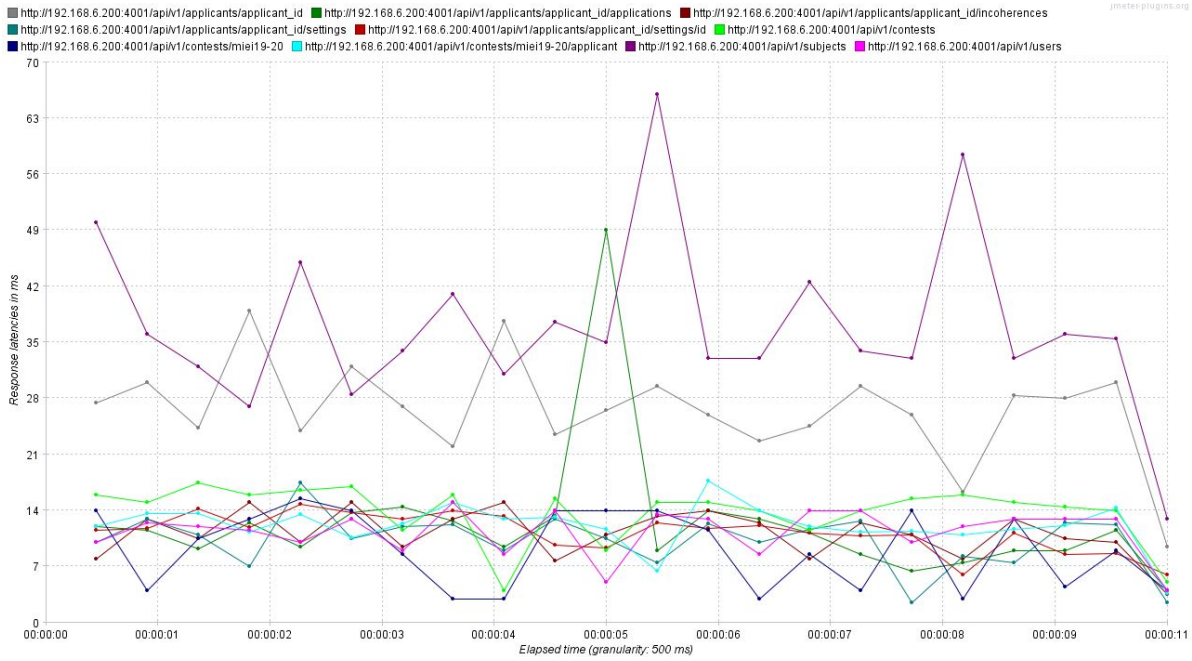

Figura 27. Response Latencies Over Time com arquitetura

Como já foi referido anteriormente, este teste corresponde a uma sequência, sendo que o tempo de resposta difere consoante o passo correspondente. Tendo em conta os diferentes passos executados, constatamos através da **Figura 28** e **Figura 29** que grande parte das operações possui um comportamento semelhante encontrando-se na mesma gama de valores. Contudo existem algumas exceções, como por exemplo as *subjects* em que o tempo de resposta é superior com a implementação da arquitetura. É também

possível visualizar que algumas linhas apesar de serem semelhantes apresentam picos localizados.

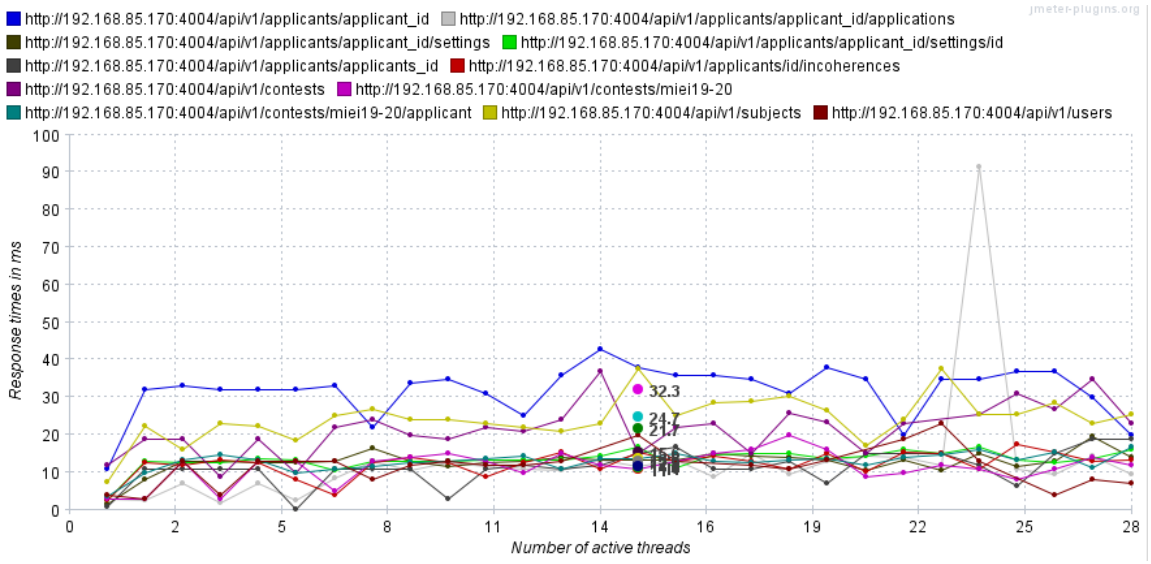

Figura 28. Response Time vs Threads sem arquitetura

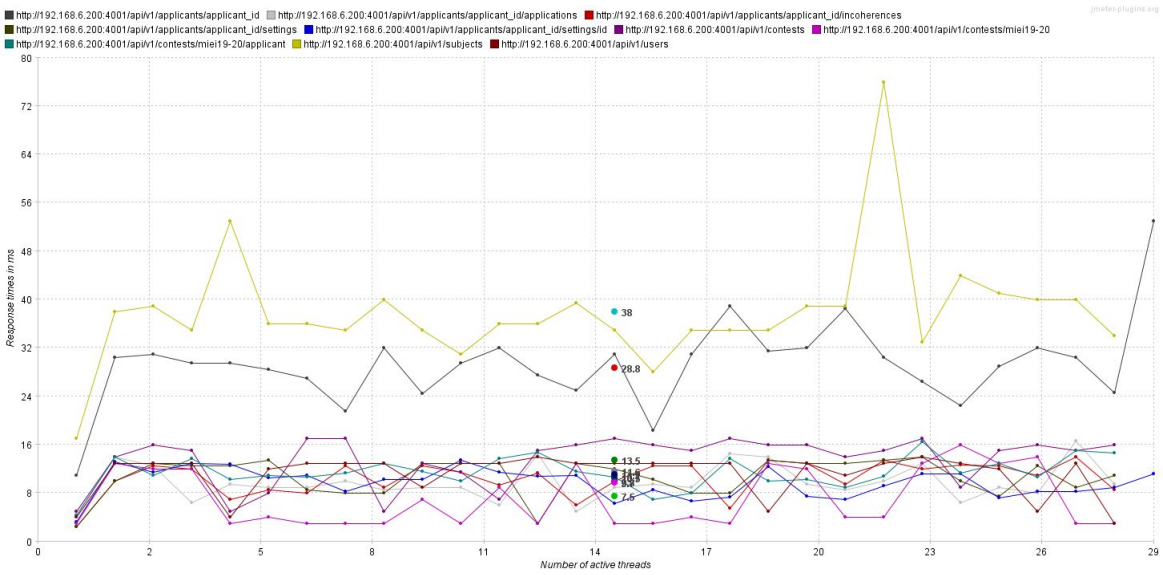

Figura 29. Response Time vs Threads com arquitetura

Através da observação da **Figura 30** e **Figura 31**, verificamos que de uma forma generalizada o comportamento em ambas as situações é semelhante, sendo que o intervalo do número de transações é idêntico. Contudo, sendo este um conjunto de operações é possível verificar umas pequenas variações em algumas.

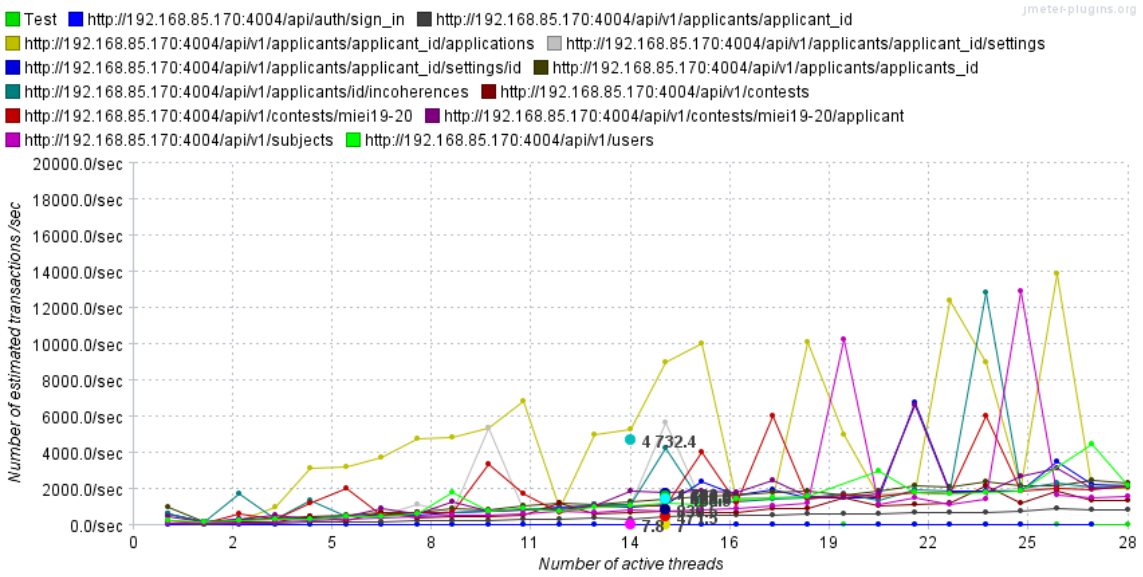

Figura 30. Transaction Throughput vs Threads sem arquitetura

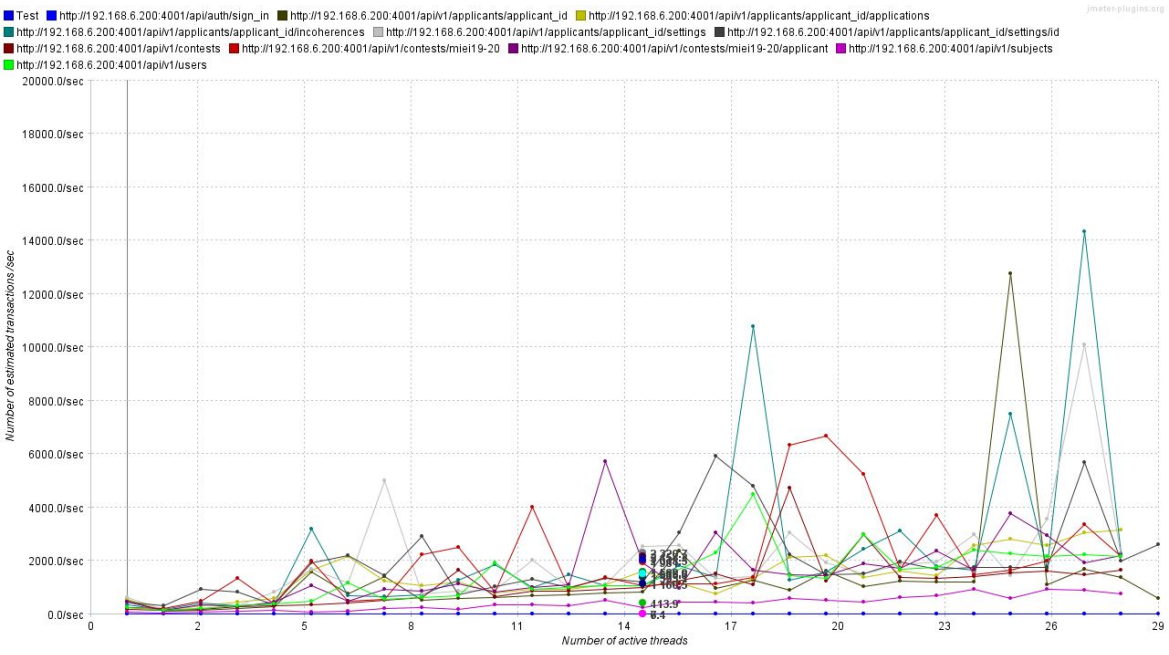

Figura 31. Transaction Throughput vs Threads sem arquitetura

8.3 Resultados da Candidatura

Por fim, testamos o processo de obtenção dos resultados e envio dos mesmos por parte de um *manager*, como tal apenas temos uma thread.

Pela observação dos gráficos da **Figura 32** e **Figura 33**, verificamos que o comportamento difere para algumas operações. Contudo a gama de valores em que se encontra o número de transações é o mesmo em ambos.

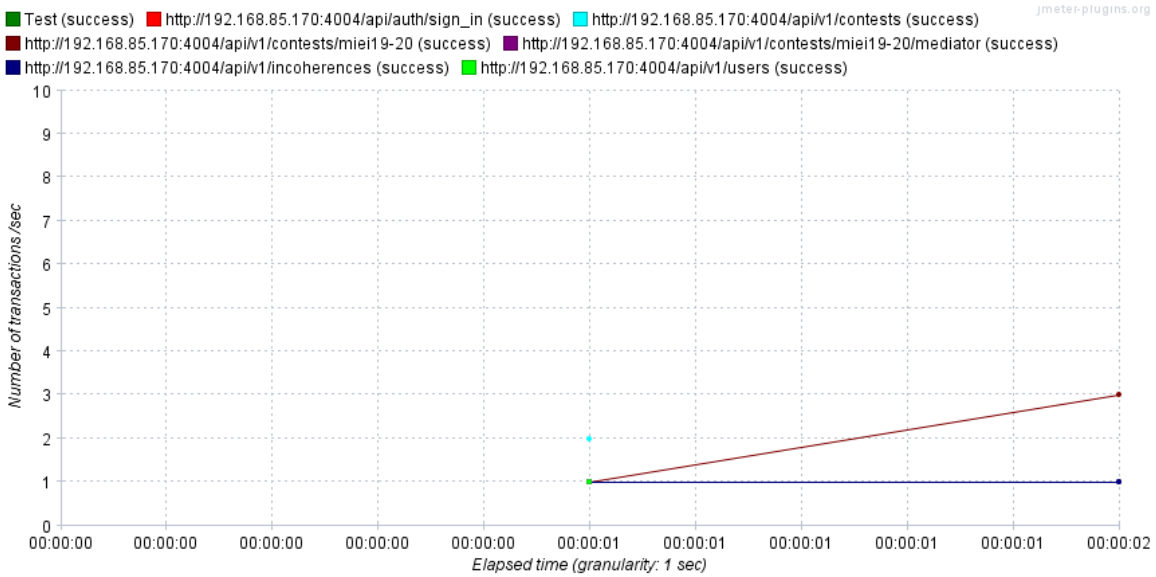

Figura 32. Transactions per second sem arquitetura

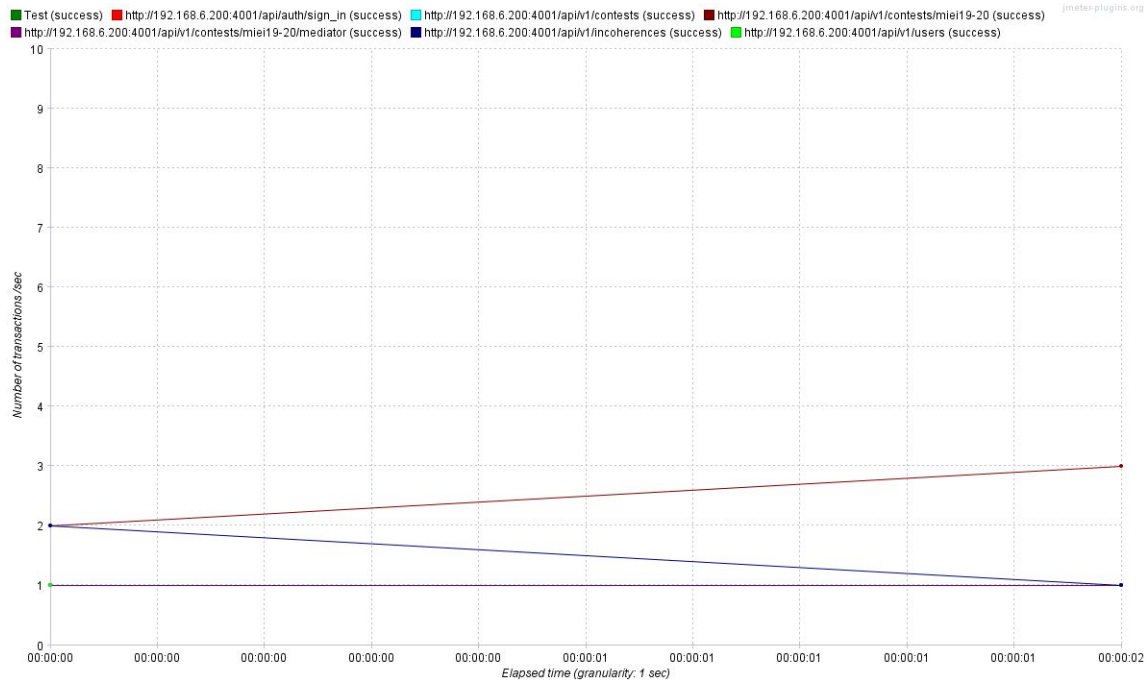

Figura 33. Response Latencies Over Time com arquitetura

Através da visualização da **Figura 34** e **Figura 35** foi possível verificar que os

gráficos apresentam um comportamento igual, sendo os tempos de resposta obtidos muito semelhantes em ambas situações.

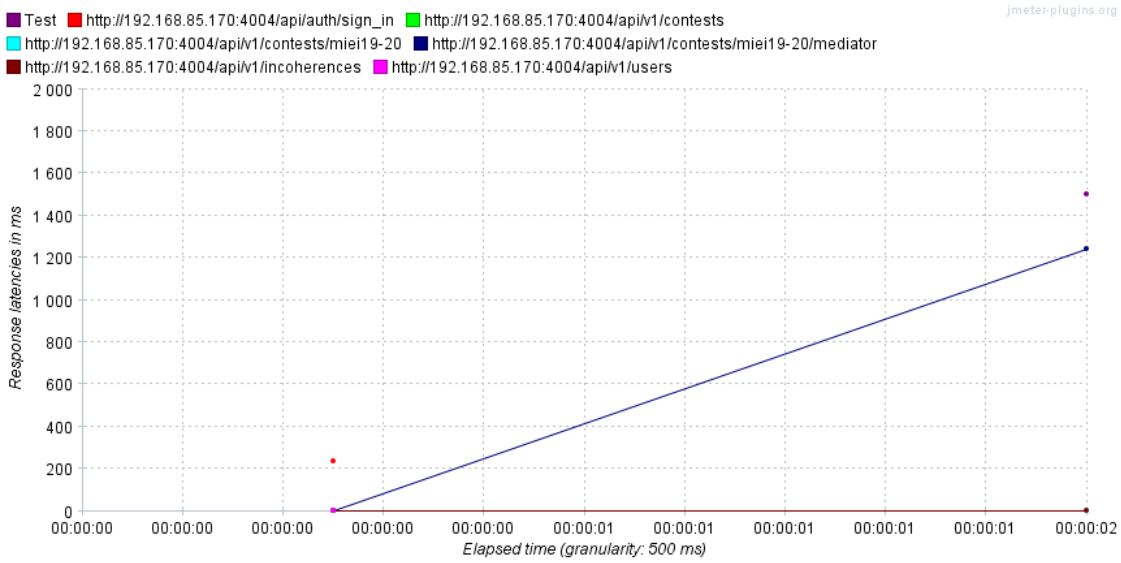

Figura 34. Response Latencies Over Time sem arquitetura

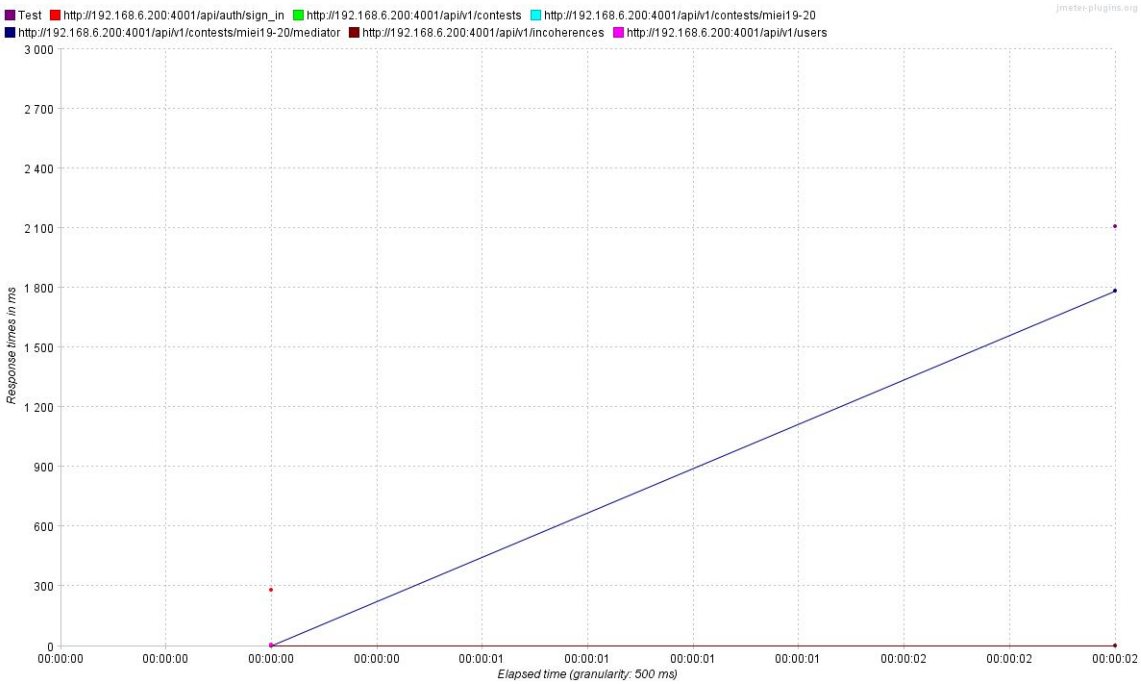

Figura 35. Response Latencies Over Time com arquitetura

9 Conclusão

Com a realização deste trabalho foi possível aplicar os conhecimentos adquiridos na unidade curricular de Infraestrutura de Centro de Dados. Assim sendo, consolidamos as noções sobre a alta disponibilidade de infraestruturas e escalabilidade das mesmas.

A arquitetura implementada permitiu que o serviço fosse escalável e sem SPOFs, que constituíam os principais objetivos do projeto. Em caso de necessidade de uma maior resposta aos utilizadores do sistema é possível modificá-lo através da adição de novas máquinas, que são transparentes aos mesmos, o que permite a alta escalabilidade da infraestrutura. De forma a garantir a alta disponibilidade dos serviços implementamos um *High Availability Cluster*.

Por último de modo a avaliar o desempenho da aplicação e da infraestrutura implementada recorremos à ferramenta *Apache JMeter* que é uma das mais utilizadas para a realização de testes de carga. Com o intuito de analisar e avaliar o desempenho das diversas funcionalidades do *Tucano* desenvolvemos três cenários de teste diferentes. Estes testes foram executados antes e depois da implementação da arquitetura, de modo a poder comparar os resultados obtidos.