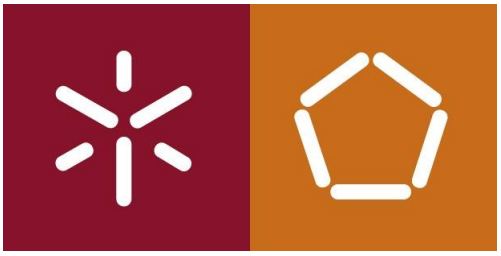

1º Semestre 2020/21
(Mestrado em Engenharia Informática)

System Deployment and Benchmarking

Trabalho Prático - Fase 2

Número	**Nome Completo**
PG42816	Cândido Filipe Lima do Vale
A89982	Joel Alexandre Dias Ferreira
A85700	Pedro Miguel Araújo Costa
A80789	Rui Filipe Brito Azevedo

15 de Janeiro de 2021

Conteúdo

1 Introdução

O presente relatório visa apresentar o processo de automatização de uma instalação de alta-disponibilidade e tolerante a falhas do sistema *Mattermost*, onde a sua análise foi elaborada na primeira fase de trabalho prático.

Na fase de análise do sistema, foram propostas várias arquitecturas para o sistema, cada uma com diferentes níveis de disponibilidade e tolerância a falhas. De todas estas soluções de implementação, escolheu-se a designada por *Multi Machine*, cuja arquitectura e componentes serão apresentados numa primeira fase do relatório.

De seguida, irão ser apresentadas as ferramentas usadas para a automatização do processo de instalação do sistema, onde também serão apresentadas, em detalhe, as arquitecturas de cada camada da aplicação, assim como a forma que camada disponibiliza os seus serviços.

Por fim, e uma vez feita a instalação do sistema, serão feitos testes de carga ao mesmo, de forma a verificar o nível de disponibilidade da aplicação.

2 Arquitectura do sistema

Na primeira fase do trabalho prático, foram apresentadas várias soluções para a arquitectura do sistema. A arquitectura escolhida para implementar foi a designada por *Multi-Machine*.

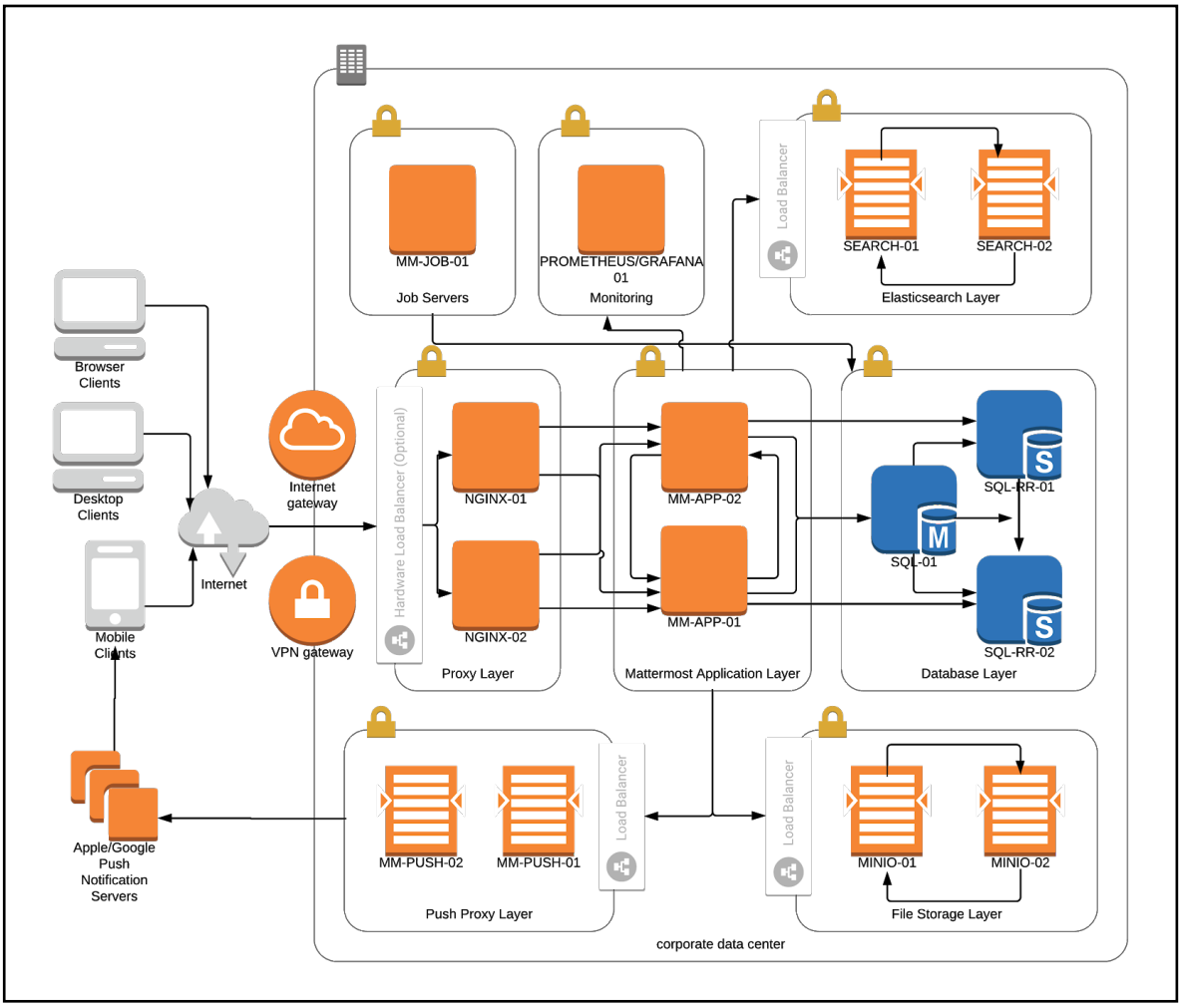

Figura 1: Arquitectura Multi-Machine

Na arquitectura apresentada acima, é possível ver as diferentes componentes da aplicação. Todos os clientes comunicam com um servidor *Proxy* para ter acesso aos serviços da aplicação. Nesta camada, existe uma instância deste tipo de servidores com a funcionalidade de balanceamento de carga dos pedidos que são enviados para os servidores aplicações, que por sua vez, são compostos por duas máquinas. Em relação à base de dados, existem três máquinas responsáveis pela persistência dos dados da aplicação. Estas três máquinas estão organizadas numa arquitectura *master-slave* onde as escritas são feitas no servidor *master* e as leituras em cada uma das três. O sistema de base de dados não é a única camada onde existem dados persistidos, uma vez que os ficheiros multimédia são guardados numa *File Storage*. Dado isto, existe uma máquina dedicada ao armazenamento de ficheiros,

havendo também um sistema de balanceamento de carga para aumentar a disponibilidade do serviço. Por fim, existe uma camada de monitorização responsável por medir métricas da infraestrutura e, por sua vez, conseguir-se ter medidas preventivas atempadas em casos de falha de componentes. A camada de *Push Proxy Layer* não foi possível ser implementada pois o seu serviço é pago pelo que se optou ignorar a sua implementação.

3 Arquitetura implementada

Ainda que a arquitetura anterior fosse ideal, acabamos por não conseguir implementar tudo o que foi mencionado. Assim, apresentamos aqui a nossa arquitetura final.

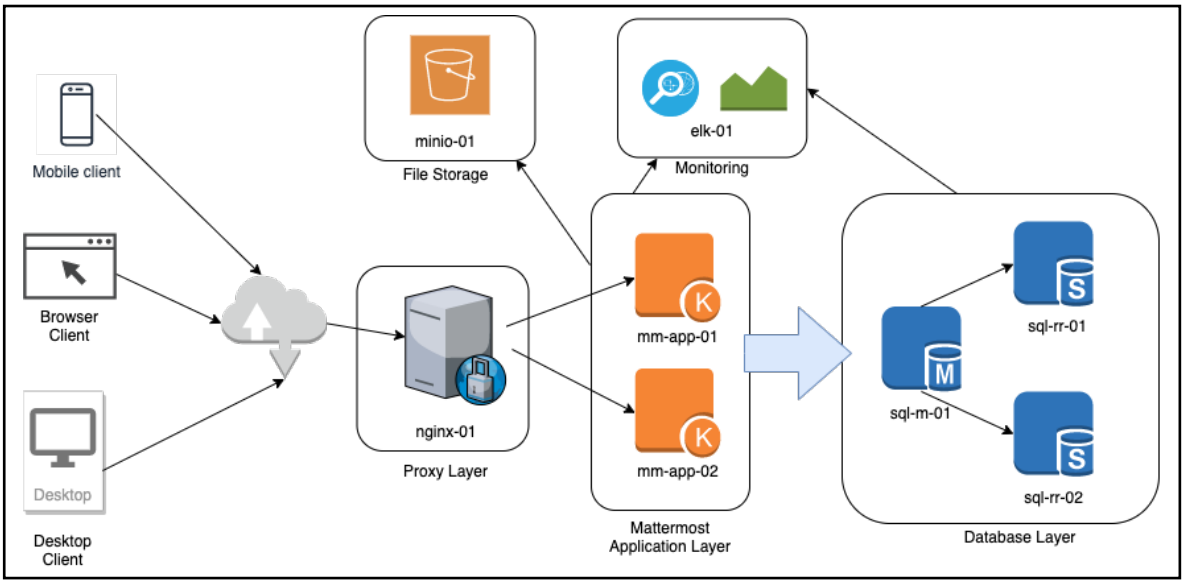

Figura 2: Arquitectura Implementada

4 Automatização

De maneira a automatizar a instalação da aplicação, foi usada a ferramenta *open source Ansible*, capaz de gerir, automatizar e configurar máquinas a partir de uma localização central.

O *Ansible* torna o processo de instalação de uma infraestrutura muito mais ágil, uma vez que todas as instalações são feitas a partir de ficheiros de configuração baseados em *YAML*. Uma grande vantagem desta ferramenta é a reutilização de *playbooks* pois elimina o processo e custo de uma reinstalação de raiz de um sistema. O poder de reutilização de *playbooks* faz com que seja possível utilizar módulos desenvolvidos por terceiros e integrá-los em diferentes projectos, como é o caso deste trabalho prático, em que foram usados módulos desenvolvidos pela comunidade e

que tornaram o processo de instalação mais eficaz.

Foram utilizados inventários dinâmicos, o que faz com que todo o nosso processo de deploy seja automático, não sendo necessária intervenção manual em nenhum passo.

De seguida, serão apresentadas, em mais detalhe, as arquitecturas de cada camada aplicacional, a forma que cada camada elimina os pontos únicos de falha do sistema e aumenta a disponibilidade do mesmo, e a forma como é feita a automatização.

4.1 Servidor Proxy

O servidor proxy, que utiliza NGINX, funciona como intermediário entre o end-user e os servidores aplicacionais. As suas funcionalidades são variadas, sendo, parte delas, a capacidade de fazer load balancing dos vários servidores, fazer *reverse proxying* e, ainda, permite a utilização de SSL que, por sua vez, permite a utilização do protocolo de comunicação segura, HTTPS.

Neste projeto, apenas foi implementado NGINX com a funcionalidade de load balancing uma vez que não necessitamos da funcionalidade de *reverse proxying*. Conseguimos obter um certificado SSL através da ferramenta OpenSSL. A estratégia de load balancing utilizada foi *round-robin* garantindo que a carga é distribuída para os servidores de forma justa.

Contrariamente ao definido na arquitetura, utiliza-se apenas uma máquina como proxy. Suponho que, para criar mais réplicas, se pudesse utilizar um swarm com vários nodos. Isto tornaria a camada proxy tolerante a falhas. Num cenário não académico, poderia considerar-se usar NGINX Plus que já disponibiliza soluções para obter High Availability.

A nível de Ansible, os scripts necessários para o setup de nginx foram bastante diretos. Basta instalar os packages necessários seguidos do nginx e correr o processo. A nível de templates configuramos o NGINX para trabalhar então com os vários endereços dos servidores aplicacionais.

4.2 Servidor Aplicacional

Para o caso dos servidores aplicacionais decidiu-se inicialmente efetuar o deploy dos mesmos utilizando *docker swarm*. Ou seja teríamos um nodo que seria o *master* e outro que seria o *worker*, existindo por isso duas réplicas do servidor aplicacional.

Apesar deste nosso objetivo, infelizmente, não o conseguímos cumprir devido aos problemas consecutivos que encontrávamos com a utilização da imagem do

container do *mattermost*. Existiam problemas com o override dos ficheiros de configuração, mesmo quando estes tinham sido 100% confirmados por nós anteriormente.

Devido a estes problemas decidiu adotar-se a estratégia mais naive de instalação local nas máquinas virtuais respetivas. Foram criadas duas máquinas virtuais através dos inventários dinâmicos usados no ansible e criados os serviços do mattermost, onde foi configurado o fichiero *config.json* com a base de dados *master* e as bases de dados *slaves* e com o file storage (MinIO).

4.3 Base de dados

O sistema de base de dados é composto por uma arquitectura *Master-Slave*, uma máquina *Master* e duas *Slaves*. Esta arquitectura permite tanto aumentar a disponibilidade do sistema, bem como diminuir os pontos únicos de falha em relação a esta camada da aplicação.

Quanto à disponibilidade, a arquitectura *Master-Slave* torna possível definir uma máquina como *Master*, onde todas as escritas do sistema são feitas na mesma, e definir máquinas como *Slave* que apenas contêm a réplica da base de dados e são só usadas para leituras. Desta maneira aumentamos a disponibilidade do sistema a nível de leituras, uma vez que esses mesmos pedidos podem ser feitos a qualquer uma das máquinas *Slave*, e a nível de escritas, pois só o *Master* faz estas operações. No ficheiro de configuração da aplicação é possível definir quais os *IP's* das máquinas *Master* e o das *Slave*.

Em relação à eliminação dos pontos únicos de falha, verifica-se que foram eliminados com a introdução de redundância nos dados. A falha de uma das máquinas *Slave* não compromete o correcto funcionamento do sistema e, no caso de falha do *Master*, uma das outras duas máquinas assume o papel de *Master*, garantindo mais uma vez, o correcto funcionamento da aplicação.

A replicação do sistema de base de dados foi implementada através da funcionalidade de replicação do *MySQL*. Esta funcionalidade permite com que os dados de um servidor de base de dados sejam copiados para um ou vários servidores (réplicas) de uma forma assíncrona. A replicação foi feita usando o método tradicional do *MySQL* baseado na replicação de eventos através de ficheiros binários de *log*. Este método implica que os ficheiros de *log* e as suas posições estejam sincronizados entre a fonte de dados e a réplica.

Para a automatização do processo de instalação, foram criados quatro *roles* com o *Ansible*, cada um responsável por uma parte específica da instalação.

4.3.1 *mysql-install*

Este *role* é usado por todas as máquinas que vão fazer parte da camada de base de dados e é responsável por fazer a instalação do *MySQL* como um serviço, alterar o parâmetro *bind-address* para o *IP* interno da máquina, e, por fim, atribuir um identificador único à máquina que é usado para definir o parâmetro *server-id*, necessário para que cada máquina na arquitectura *Master-Slave* seja unicamente identificada.

4.3.2 *mysql-master*

Este *role* é usado para configurar o servidor *Master*. É responsável por definir o método de replicação da base de dados que neste caso é o método baseado em ficheiros binários de *log*, criar o utilizador *mattermost*, dar permissões de replicação ao mesmo e criar um ficheiro de *dump* da base de dados.

4.3.3 *mysql-slave*

É com este *role* que as máquinas *Slave* são configuradas. Numa primeira fase, logo após o serviço de base de dados estar a correr, a funcionalidade de *Slave* é desligada para se puder configurar a replicação. Esta configuração passa por definir qual a máquina que é o seu *Master*, definir o utilizador que está a ser usada para a replicação, carregar o ficheiro de *dump* na base de dados e, por fim, reinicializar o serviço de *Slave*.

4.3.4 *mysql-after-repl*

A única coisa que este *role* faz é criar a base de dados *mattermost* quando o processo de replicação está concluído. Esta é a última fase do processo de replicação da base de dados, sendo que ao fim desta instalação o sistema será capaz de fazer escritas e leituras mais eficientes e garantir que eventuais falhas das máquinas são transparentes aos utilizadores.

4.4 Armazenamento de ficheiros

Para o armazenamento de ficheiros tinhamos várias opções e, como mencionado na primeira fase, optamos por utilizar um serviço compatível com S3, nomeadamente, o MinIO. Nesta camada são guardados todos os ficheiros multi-média partilhados por utilizadores.

Aqui, contrariamente ao mencionado na arquitetura, também acabou por se utilizar apenas uma instância. Para escalar para várias máquinas, uma boa hipótese passava por utilizar Docker Swarm. Assim, passaríamos a ter grande disponibilidade, tolerância a falhas e, ainda, uma object store mais escalável de forma geral.

A nível de deployment, temos essencialmente duas fases para instalar o MinIO e configurá-lo com o Amazon S3. Numa primeira fase, instalamos o minio e colocamos o serviço a correr. De seguida, instala-se a aplicação aws e ajusta-se credenciais necessárias. Por fim, cria-se o bucket onde vão ser guardados os ficheiros multi-média.

O mattermost lê do ficheiro de configuração toda a informação relevante a esta secção, não sendo necessário mais nada para ter o armazenamento de ficheiros funcional.

4.5 Monitorização

Como já tínhamos visto inicialmente, o Mattermost já tem incorporada uma funcionalidade que permite recolher métricas dos componentes e enviá-las para uma camada de monitorização onde é feita a análise dessas métricas. Na nossa implementação, de modo a tentar simplificar e agilizar o processo de *deployment* do sistema tentamos recorrer a essa funcionalidade, sendo que em caso de sucesso, só seria necessário implementar as ferramentas aconselhadas pelo Mattermost para monitorização (Prometheus) e para visualização das métricas recolhidas (Grafana), editando nos ficheiros dos servidores aplicacionais os campos com o IP da(s) máquinas onde tivéssemos os componentes de monitorização.

Contudo, percebemos que este mecanismo é uma funcionalidade do Mattermost Enterprise e que, apesar de ser possível utilizá-la por um período recorrendo a um free-trial, é necessário recorrer a um registo, o que tornaria o processo de *deployment* muito menos automatizável, indo contra o principal objetivo do trabalho. Tendo isto em consideração, de modo a implementar a monitorização do nosso sistema decidimos recorrer à pilha ELK por já ter sido utilizada em aula e portanto, já estarmos mais familiarizados com o funcionamento da mesma.

4.5.1 Ferramentas Utilizadas

- MetricBeats: Permitem efetuar a monitorização da disponibilidade e performance dos diferentes componentes da infraestrutura através de recolha de métricas dos diferentes servidores como CPU ou memória em utilização ou mesmo acerca de serviços como por exemplo bases de dados (MySQL / Postgresql), através dos diferentes módulos disponíveis.
- FileBeats: Permite coleccionar e enviar tipos de logs variados de diferentes servidores para o Elasticsearch, permitindo-nos portanto, analisar os logs de diferentes servidores de forma centralizada.
- PacketBeat: Ferramenta que permite recolher métricas em tempo real e portanto ter uma visão generalizada do que se passa em toda a infraestrutura de

rede. Permite analisar o tráfego a circular na rede bem como a sua disponibilidade e performance.

- Elasticsearch: mecanismo de busca e análise de dados. Irá ser responsável por receber a informação enviada pelos Beats e permitir efetuar pesquisas eficientes sobre as métricas recebidas.
- Kibana: Permite analisar e explorar grandes quantidades de dados em tempo real, permitindo também uma visualização gráfica dos dados obtidos de forma a permitir uma análise e consulta fácil.

Esta implementação tem a vantagem de ser completamente independente, ou seja, não é necessária qualquer configuração adicional nos restantes componentes do sistema, sendo também possível reutilizar os playbooks utilizados para implementar monitorização em qualquer outro sistema, contudo, tem a desvantagem de acrescenta rmais etapas ao processo de *deployment* do sistema.

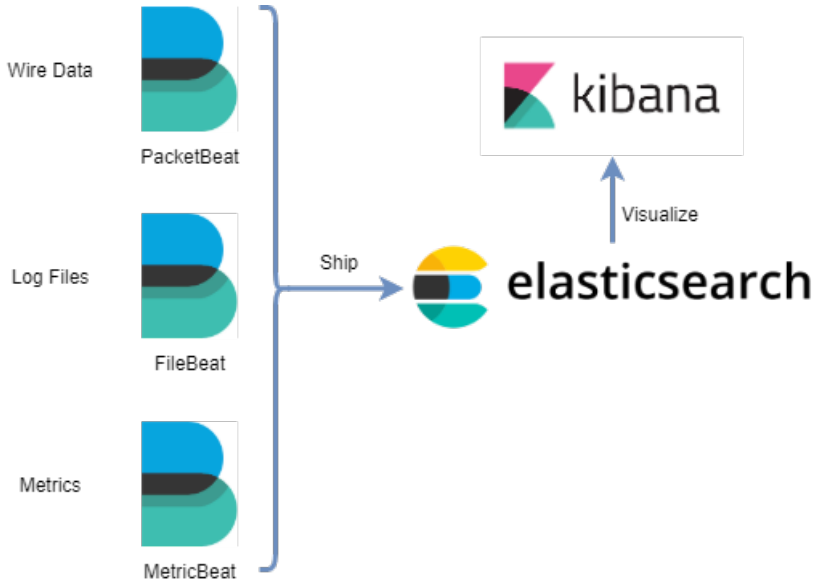

Figura 3: Arquitetura de Monitorização

Para implementar a camada de monitorização visível na figura abaixo, Fig. 3, decidimos inicialmente utilizar duas máquinas virtuais por ser a implementação sugerida na documentação do Mattermost, sendo que seria a implementação adequada para números de utilizadores entre os 5 e os 10 mil o que significaria que optando por implementar já em duas máquinas virtuais, mesmo tendo necessidade de escalar o sistemas, já não é necessário preocuparmos-nos com a camada de monitorização, contudo devido à limitação existente de não podermos ter mais que oito máquinas virtuais numa determinada zona levou-nos a optar por uma implementação mais simples, utilizando apenas uma máquina virtual.

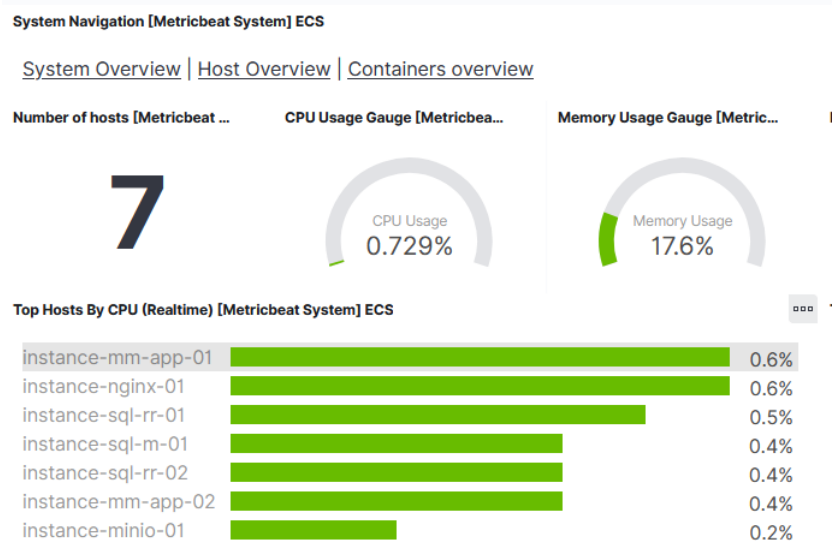

Figura 4: Dashboard Geral Kibana - com as Métricas Recolhidas com o sistema sem carga

4.5.2 Métricas Utilizadas

1. Métricas de Performance do Sistema

   Métricas como utilização de CPU, memória, utilização do disco ou rede (System CPU, System Memory, System FileSystem e System Network no Beats) que são recolhidas de forma a identificar possíveis Bottlenecks de performance que causem degradação de performance no sistema. Esta métricas permitem identificar possíveis pontos de falha antecipadamente permitindo ao administrador do sistema modificar a arquitectura antes que essa falha possa ter implicações relevantes.

2. Métricas de Performance da Aplicação

   Estas métricas permitem avaliar componentes aplicacionais específicos do sistema como Bases de Dados, Servidores Aplicacionais ou de Proxy. Neste caso, podíamos utilizar os módulos disponibilizados pelos Beats como por exemplo os módulos de MySQL ou Nginx que recolherão métricas associadas a estas aplicações em específico. Este tipo de métricas é útil para monitorizar determinados processos a correr nos servidores e identificar quais deles estão a ter uma maior impacto na performance, permitindo gerir de forma mais eficiente os recursos do sistema.

3. Métricas Relacionadas com a Segurança do Sistema

   Apesar de não termos conseguido com sucesso implementar este tipo de

métricas (foi feita uma tentativa sem sucesso de utilizar o logstash para filtrar por exemplo, tentativas de acesso SSH às máquinas) estas métricas são também importantes na monitorização de sistemas em que se pretende alta disponibilidade. Utilizando por exemplo, o Filebeat com o Logstash para os diferentes componentes poderíamos por exemplo, obter os registos de tentativas de acesso SSH aos diferentes servidores, o que seria útil para identificar possíveis tentativas de acesso indevido aos diferentes componentes do sistema.

5 Testes

Uma vez feita a instalação da aplicação, irá-se proceder à análise de desempenho da mesma. Para isto, foi criada uma bateria de testes, através da ferramenta *JMeter*, onde várias operações sobre o sistema serão feitas, particularmente, operações que apresentam um maior custo de desempenho.

Como foi referido, os testes realizados foram feitos através do *Apache JMeter*, ferramenta que permite realizar testes de carga a um sistema. O *JMeter* permite definir o número de clientes usados para fazer os testes (*threads*) e definir a operação ou conjunto de operações que vão ser executadas durante os mesmos. Uma vez executados os testes, o *JMeter* é capaz de gerar uma *dashboard*, em *HTML*, onde podem ver visualizadas diferentes métricas, como por exemplo, débito máximo, tempo de resposta, percentagem de pedidos que não foram executadas, *etc.*

Como ferramenta auxiliar, foi usado o *BlazeMeter* que permite criar um plano de testes para o *JMeter* de uma forma muito mais intuitiva e rápida. Para isto, basta abrir a aplicação sobre testes, neste caso o *Mattermost*, dizer ao *BlazeMeter* para começar e gravar e, a partir daí, no *browser*, criar o comportamento desejado. No fim, após parar de gravar, o *BlazeMeter* contém todos os pedidos feitos à aplicação, que pode converter para um ficheiro do tipo *jmx*, para ser carregado no *JMeter*

Para que haja uma medida de comparação e para salientar a importância de um serviço de alta disponibilidade, foram feitos testes à aplicação implementada numa arquitectura *single-machine* e numa arquitectura *multi-machine*. Todos os testes foram corridos para 100, 300 e 500 *threads*

5.1 Testes *Single-Machine*

Estes testes foram feitos com todas as camadas da aplicação implementadas na mesma máquina. É de esperar que o seu desempenho seja baixo relativamente a uma arquitectura de alta disponibilidade.

5.1.1 Teste 1 - *GET* da Página Inicial

Statistics

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**100**	**0**	**0.00%**	**150.13**	**83**	**244**	**133.50**	**232.90**	**238.00**	**243.97**	**119.05**	**415.16**	**13.25**
HTTP Request	100	0	0.00%	150.13	83	244	133.50	232.90	238.00	243.97	119.05	415.16	13.25

Figura 5: Teste 1 - *Single-Machine* - 100 *threads*

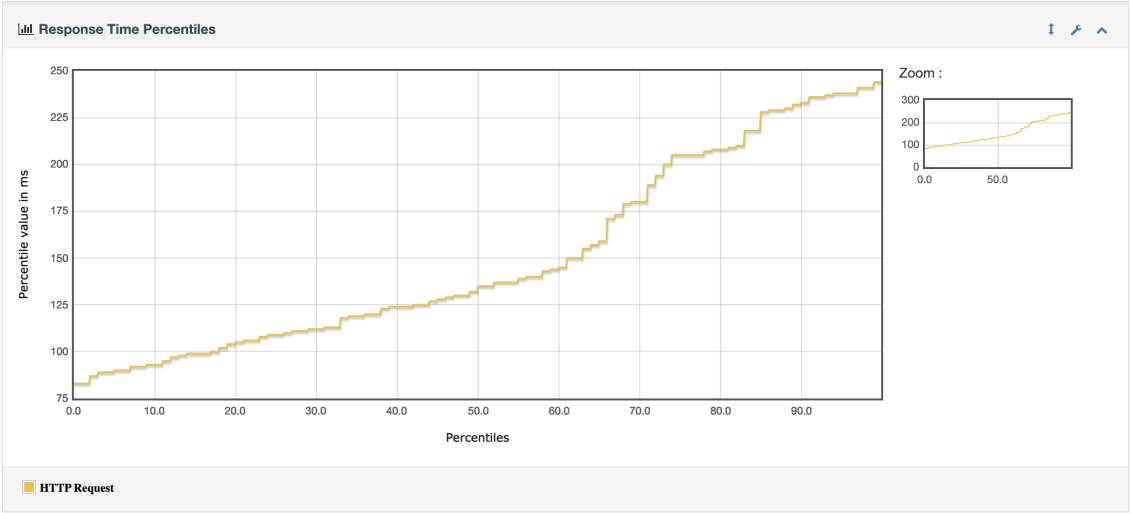

Figura 6: Teste 1 - *Single-Machine* - 100 *threads*

Statistics

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**300**	**0**	**0.00%**	**414.25**	**320**	**1062**	**401.50**	**472.00**	**492.00**	**1057.93**	**276.24**	**963.34**	**30.75**
HTTP Request	300	0	0.00%	414.25	320	1062	401.50	472.00	492.00	1057.93	276.24	963.34	30.75

Figura 7: Teste 1 - *Single-Machine* - 300 *threads*

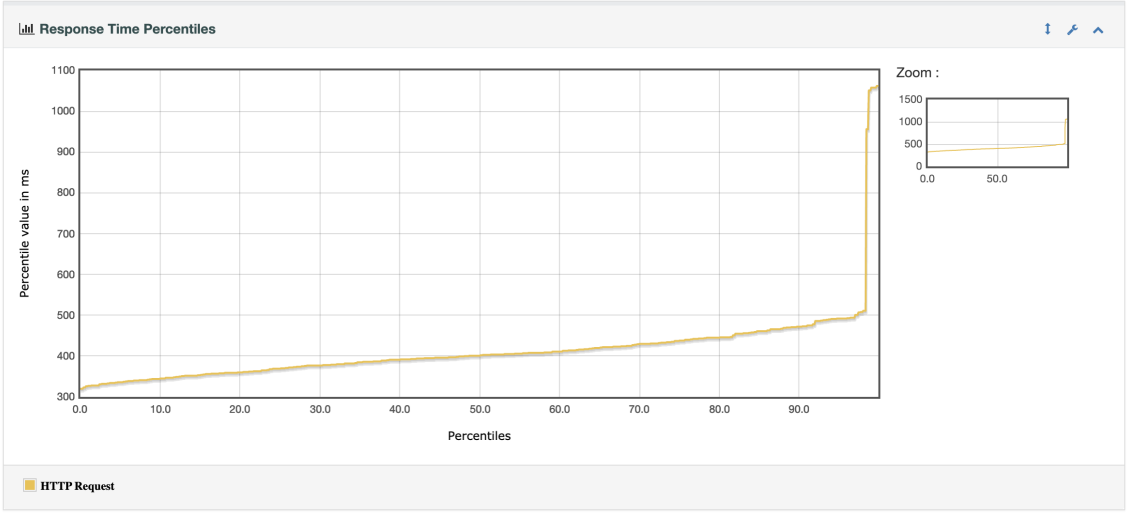

Figura 8: Teste 1 - *Single-Machine* - 300 *threads*

Statistics

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	500	0	0.00%	947.77	279	2683	706.50	1723.80	2193.90	2428.96	165.13	575.84	18.38
HTTP Request	500	0	0.00%	947.77	279	2683	706.50	1723.80	2193.90	2428.96	165.13	575.84	18.38

Figura 9: Teste 1 - *Single-Machine* - 500 *threads*

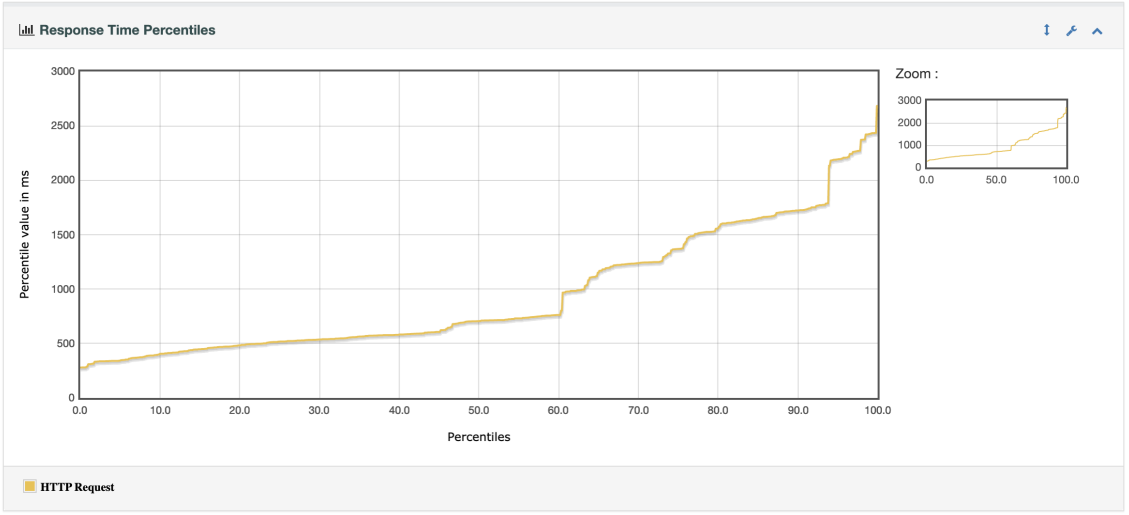

Figura 10: Teste 1 - *Single-Machine* - 500 *threads*

Uma vez que este tipo de teste não apresenta grande carga de processamento, como era de esperar, todos os pedidos foram efectuados com sucesso. No entanto, pode-se observar uma degradação no tempo de resposta à medida que o número de *threads* aumenta.

5.1.2 Teste 2 - *Login*

Executions			Response Times (ms)							Throughput	Network (KB/sec)	
#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
2800	7	0.25%	581.19	43	9839	146.00	472.90	2997.50	9221.97	163.12	252.48	87.77

Figura 11: Teste 2 - *Single-Machine* - 100 *threads*

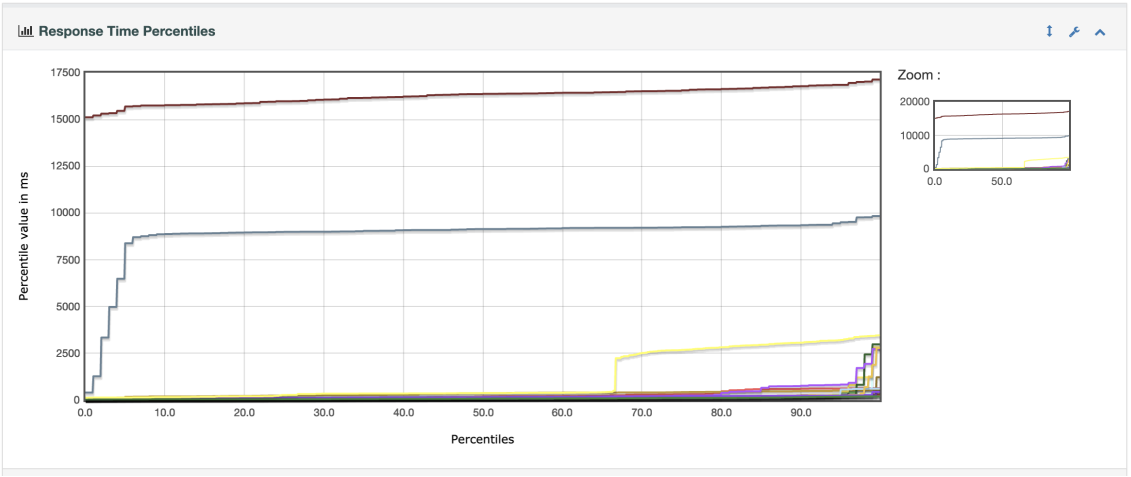

Figura 12: Teste 2 - *Single-Machine* - 100 *threads*

Executions			Response Times (ms)							Throughput	Network (KB/sec)	
#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
8400	3613	43.01%	1302.73	40	21429	253.00	1327.90	9146.95	20684.95	214.55	220.28	103.90

Figura 13: Teste 2 - *Single-Machine* - 300 *threads*

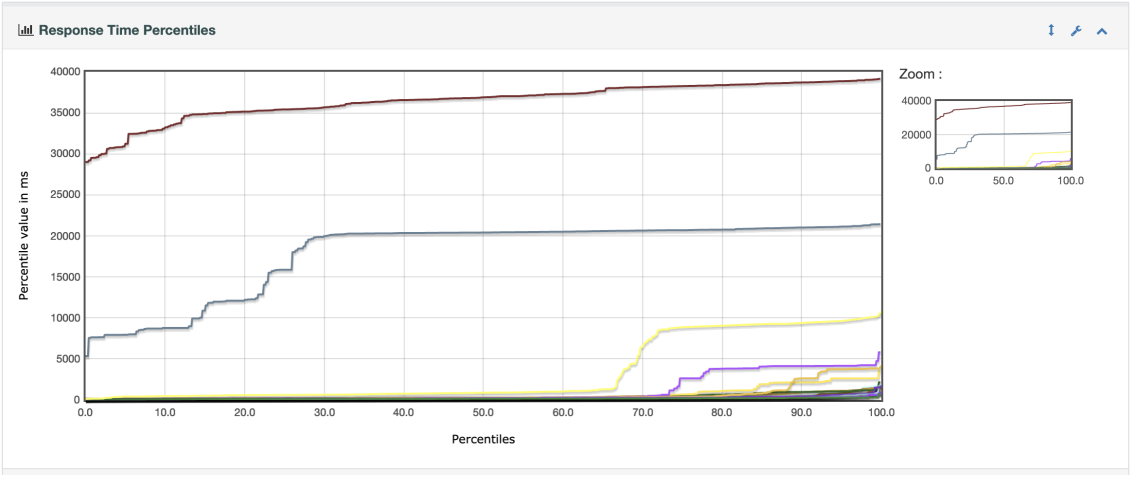

Figura 14: Teste 2 - *Single-Machine* - 300 *threads*

Executions			Response Times (ms)							Throughput	Network (KB/sec)	
#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
14000	1820	13.00%	2050.54	37	46153	356.00	1869.50	4752.90	42632.91	196.87	279.24	103.77

Figura 15: Teste 2 - *Single-Machine* - 500 *threads*

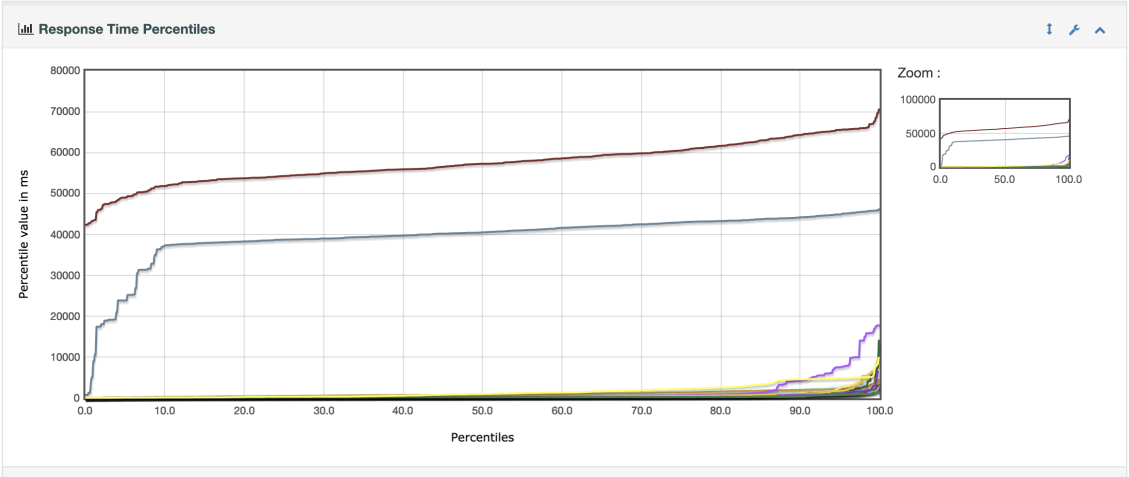

Figura 16: Teste 2 - *Single-Machine* - 500 *threads*

Este teste exige mais carga computacional por lado do servidor visto que é necessário autenticar todos os pedidos recebidos. Dado isto, é possível observar que todos os testes apresentam uma percentagem de erro, ou seja, nem todos os pedidos foram atendidos pelo servidor. Em relação ao débito, é de notar que o seu valor decresce quando se passam de 300 para 500 *threads* e, além disso, o tempo de resposta aumenta sempre que se aumenta o número de clientes.

5.1.3 Teste 3 - *Login* e Envio de Mensagens

Executions			Response Times (ms)							Throughput	Network (KB/sec)	
#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
4000	8	0.20%	451.98	41	9897	151.00	398.90	2210.95	9193.89	208.67	323.97	129.47

Figura 17: Teste 3 - *Single-Machine* - 100 *threads*

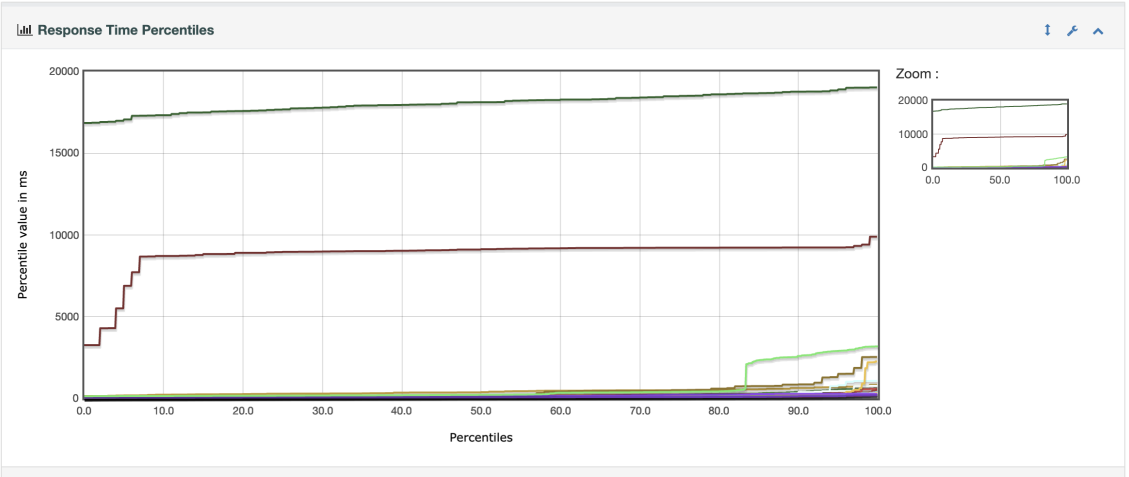

Figura 18: Teste 3 - *Single-Machine* - 100 *threads*

Executions			Response Times (ms)							Throughput	Network (KB/sec)	
#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
12000	4140	34.50%	942.64	42	26063	215.00	713.00	1760.95	23985.93	277.78	320.28	160.31

Figura 19: Teste 3 - *Single-Machine* - 300 *threads*

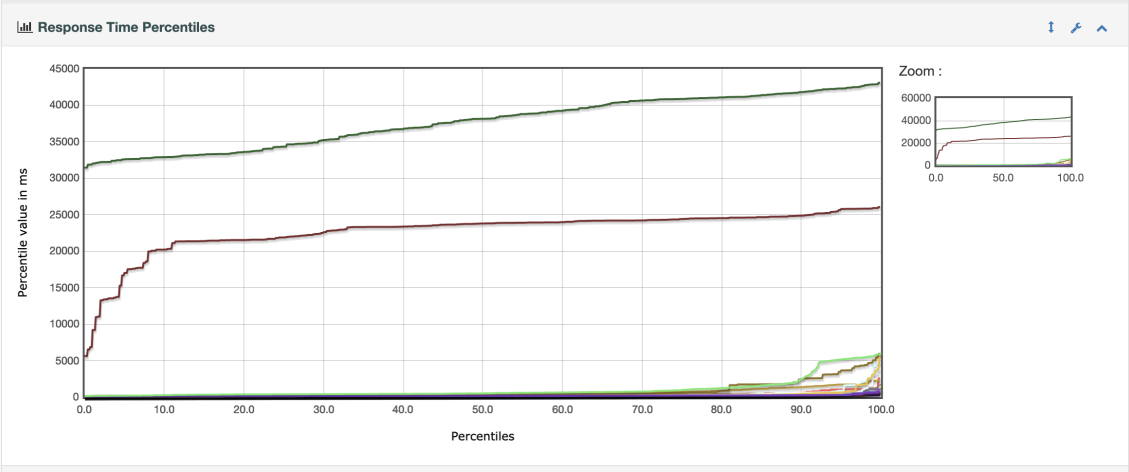

Figura 20: Teste 3 - *Single-Machine* - 300 *threads*

Executions			Response Times (ms)							Throughput	Network (KB/sec)	
#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
20000	3920	19.60%	1642.21	38	42506	314.00	2262.70	4698.85	37236.81	229.32	317.19	136.64

Figura 21: Teste 3 - *Single-Machine* - 500 *threads*

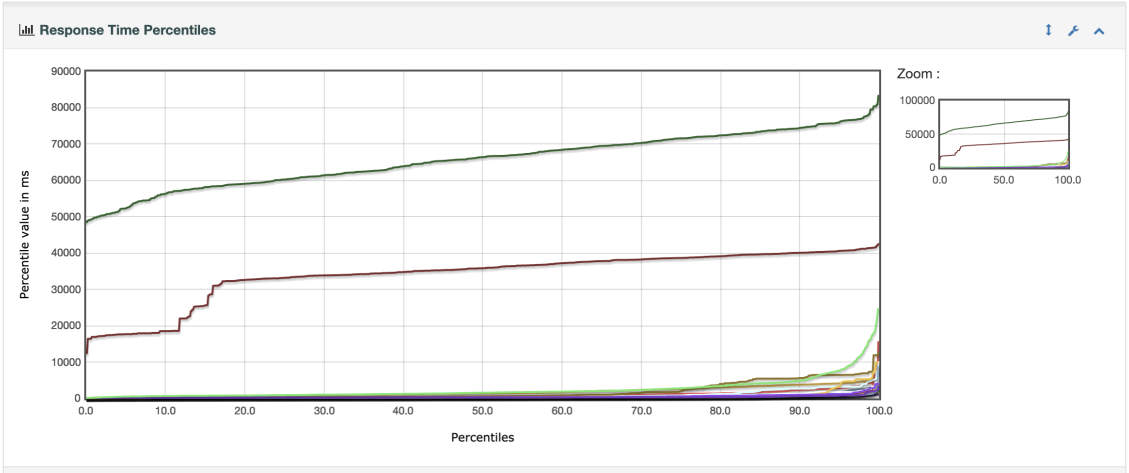

Figura 22: Teste 3 - *Single-Machine* - 500 *threads*

5.2 Testes *Multi-Machine*

5.2.1 Teste 1 - *GET* da Página Inicial

Statistics

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**300**	**0**	**0.00%**	**1173.86**	**76**	**2166**	**1465.00**	**1991.70**	**2106.75**	**2148.96**	**137.99**	**360.17**	**19.76**
HTTP Request	100	0	0.00%	1769.79	1135	2166	1778.00	2119.50	2142.85	2165.91	46.00	180.08	9.88
HTTP Request-0	100	0	0.00%	121.76	76	222	109.50	176.90	184.85	221.68	90.50	33.85	9.72
HTTP Request-1	100	0	0.00%	1630.03	1037	1989	1684.50	1953.40	1960.90	1988.76	50.15	177.58	5.39

Figura 23: Teste 1 - *Multi-Machine* - 100 *threads*

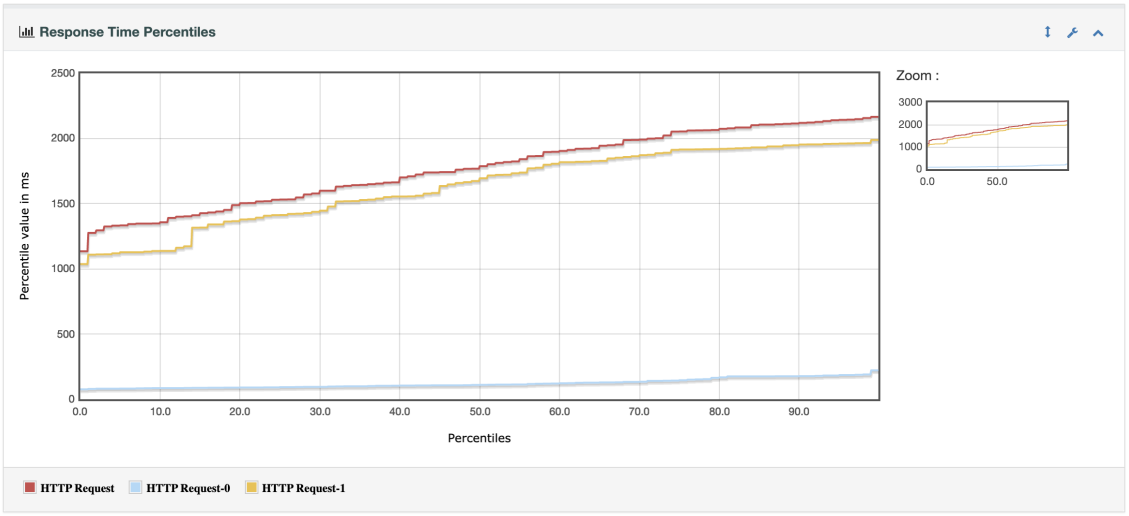

Figura 24: Teste 1 - *Multi-Machine* - 100 *threads*

Statistics

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**900**	**0**	**0.00%**	**2944.33**	**80**	**5150**	**4136.00**	**4686.00**	**4788.95**	**4970.98**	**167.69**	**437.68**	**24.02**
HTTP Request	300	0	0.00%	4434.16	2800	5150	4462.00	4860.50	4946.75	5042.89	55.90	218.84	12.01
HTTP Request-0	300	0	0.00%	195.63	80	701	139.00	295.60	546.00	697.99	212.46	79.47	22.82
HTTP Request-1	300	0	0.00%	4203.19	2180	4932	4290.00	4648.80	4705.00	4851.82	58.16	205.95	6.25

Figura 25: Teste 1 - *Multi-Machine* - 300 *threads*

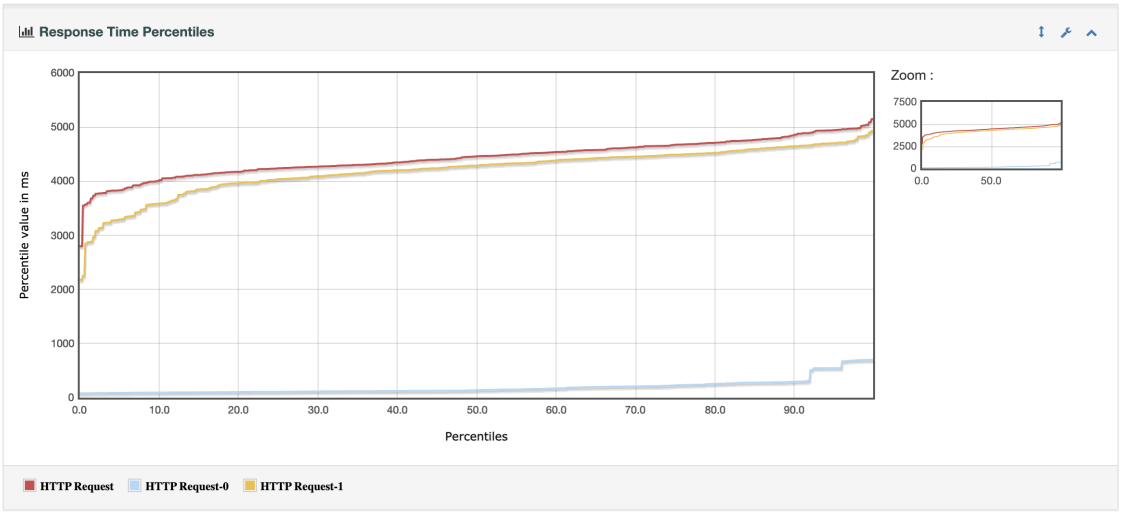

Figura 26: Teste 1 - *Multi-Machine* - 300 *threads*

Statistics

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**1500**	**24**	**1.60%**	**4852.24**	**97**	**8468**	**6875.50**	**7730.00**	**7869.75**	**8149.95**	**176.60**	**460.89**	**24.99**
HTTP Request	500	12	2.40%	7288.97	5264	8468	7317.00	7935.50	8093.85	8281.95	58.87	230.44	12.50
HTTP Request-0	500	0	0.00%	168.05	97	416	162.00	233.90	240.00	276.90	396.83	148.42	42.63
HTTP Request-1	500	12	2.40%	7099.71	5102	8198	7120.00	7738.30	7847.95	8050.91	60.37	213.76	6.33

Figura 27: Teste 1 - *Multi-Machine* - 500 *threads*

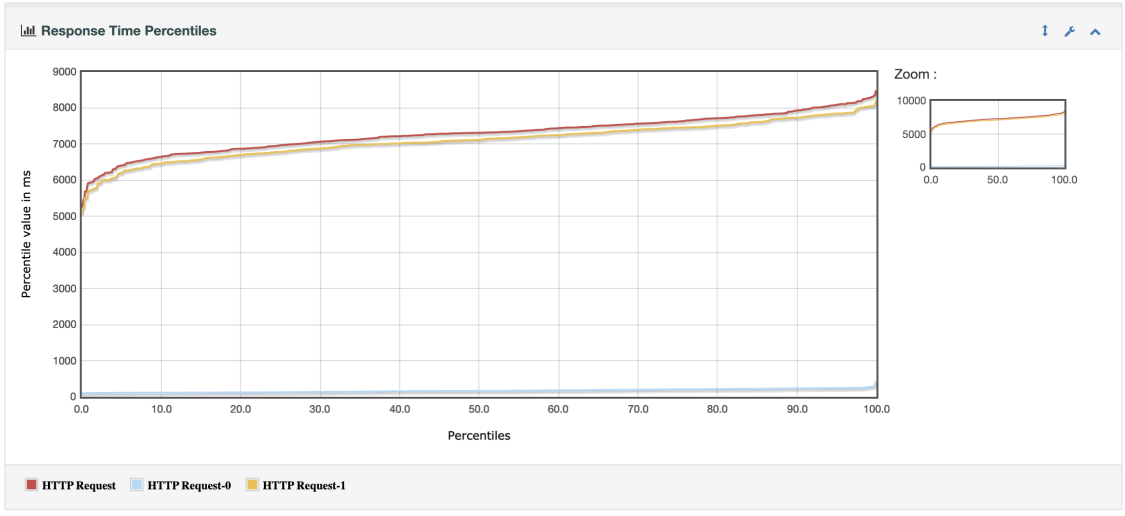

Figura 28: Teste 1 - *Multi-Machine* - 500 *threads*

Em comparação com os resultados obtidos anteriormente, verifica-se quando o número de clientes é de 500, existe uma pequena percentagem de pedidos que não foram atendidos. Para além disso, houve um ligeiro aumento no débito e um grande aumento no tempo de resposta.

5.2.2 Teste 2 - *Login*

Executions			Response Times (ms)							Throughput	Network (KB/sec)	
#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
3400	0	0.00%	338.83	39	5636	83.00	362.00	2423.70	5273.96	278.00	274.58	192.30

Figura 29: Teste 2 - *Multi-Machine* - 100 *threads*

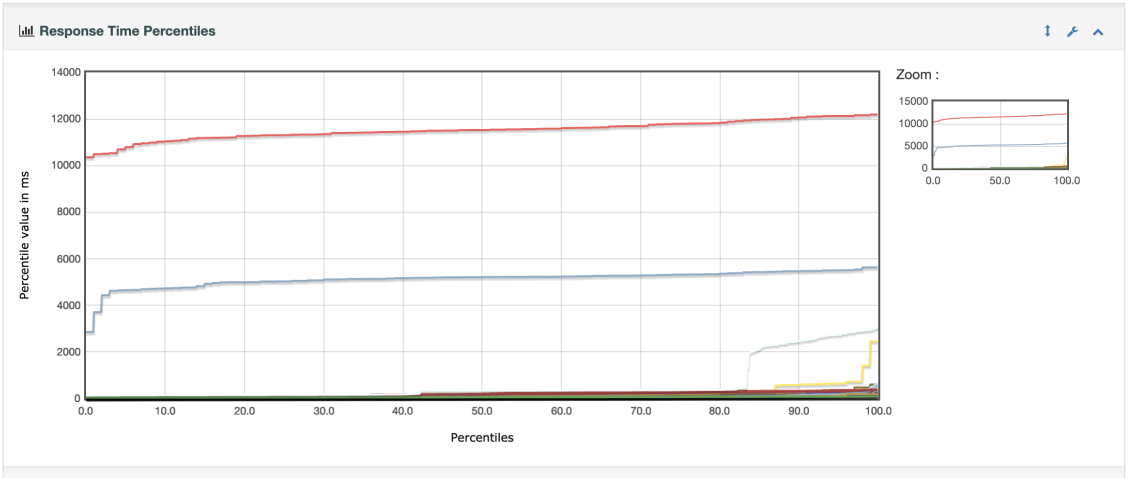

Figura 30: Teste 2 - *Multi-Machine* - 100 *threads*

Executions			Response Times (ms)							Throughput	Network (KB/sec)	
#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
10200	2509	24.60%	723.10	38	17313	159.00	833.80	5089.95	13725.91	343.05	292.53	227.82

Figura 31: Teste 2 - *Multi-Machine* - 300 *threads*

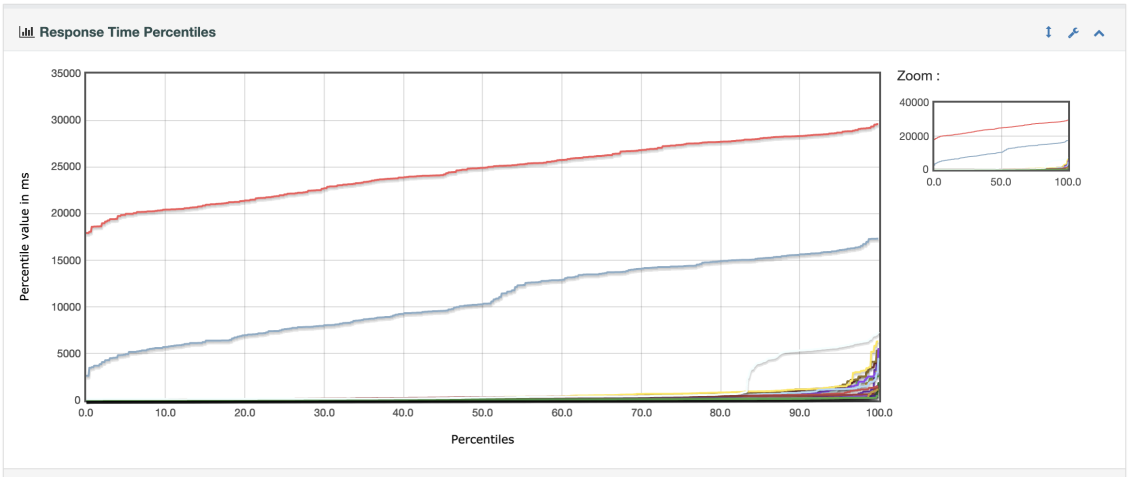

Figura 32: Teste 2 - *Multi-Machine* - 300 *threads*

Executions			Response Times (ms)							Throughput	Network (KB/sec)	
#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
17000	8202	48.25%	613.61	0	17386	210.00	885.00	2021.00	10478.98	503.36	436.20	305.39

Figura 33: Teste 2 - *Multi-Machine* - 500 *threads*

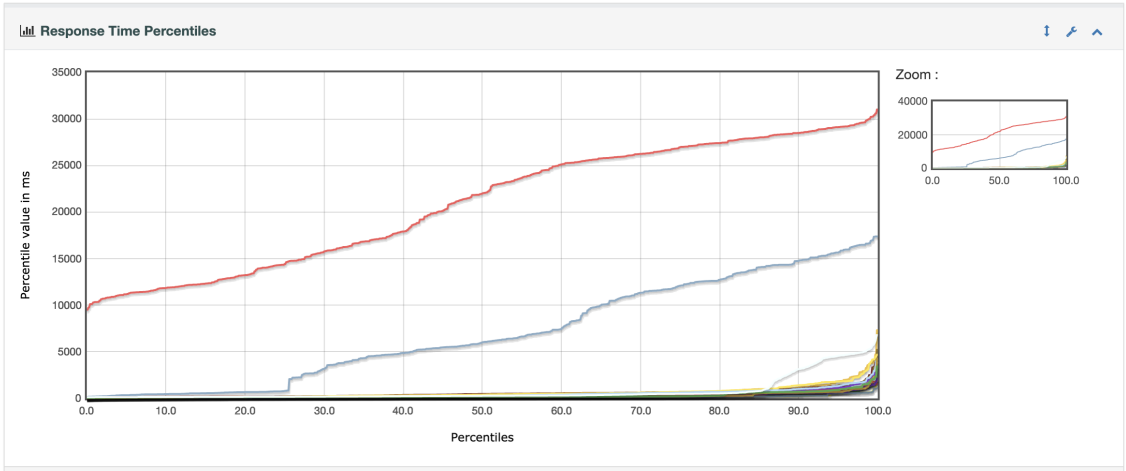

Figura 34: Teste 2 - *Multi-Machine* - 500 *threads*

Para 100 clientes, a percentagem de erros é menor em comparação com o *Single-Machine*, sendo que todos os pedidos foram atendidos. Para 300 *threads*, o débito aumentou ainda significativamente e o tempo de resposta diminui. Para 500 *threads*, a percentagem de erros é significativamente mais alta, bem como o débito atingido.

5.2.3 Teste 3 - *Login* e Envio de Mensagens

Executions			Response Times (ms)							Throughput	Network (KB/sec)	
#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
3800	28	0.74%	337.26	37	6382	79.00	372.90	1852.45	5676.97	260.24	251.99	191.57

Figura 35: Teste 3 - *Multi-Machine* - 100 *threads*

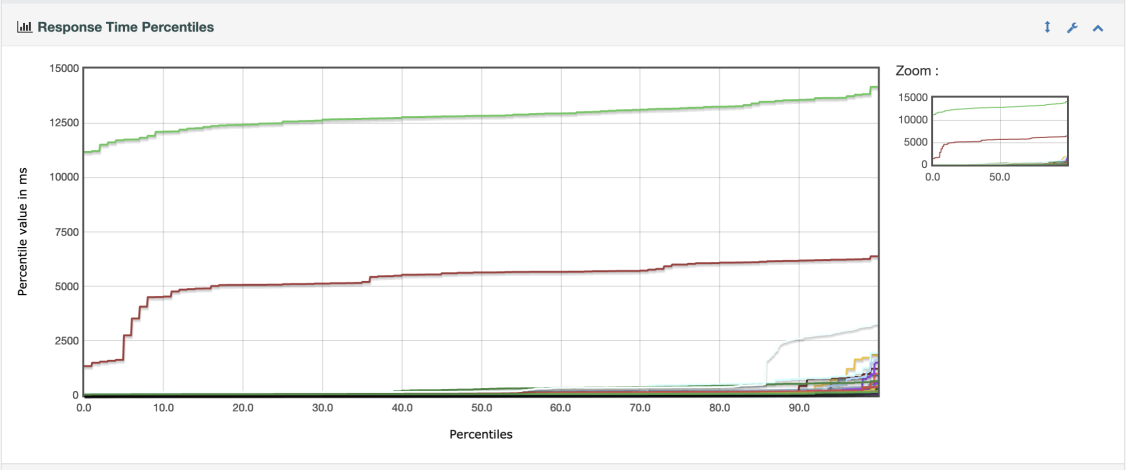

Figura 36: Teste 3 - *Multi-Machine* - 100 *threads*

Executions			Response Times (ms)							Throughput	Network (KB/sec)	
#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
11400	3869	33.94%	579.56	37	15839	168.00	746.00	2337.95	10927.73	399.54	351.13	277.66

Figura 37: Teste 3 - *Multi-Machine* - 300 *threads*

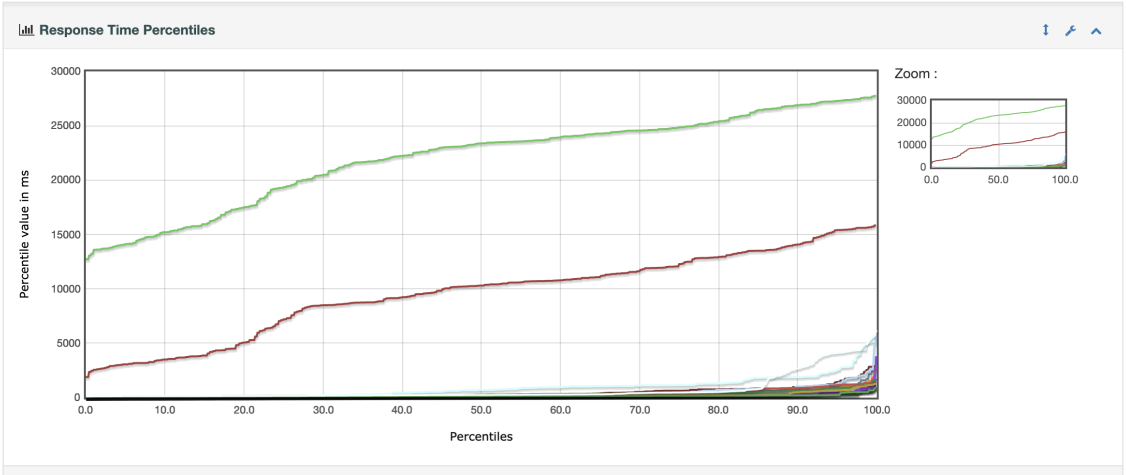

Figura 38: Teste 3 - *Multi-Machine* - 300 *threads*

Executions			Response Times (ms)							Throughput	Network (KB/sec)	
#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
19000	7746	40.77%	519.35	0	13923	249.00	796.00	1551.90	6378.62	592.75	561.72	397.82

Figura 39: Teste 3 - *Multi-Machine* - 500 *threads*

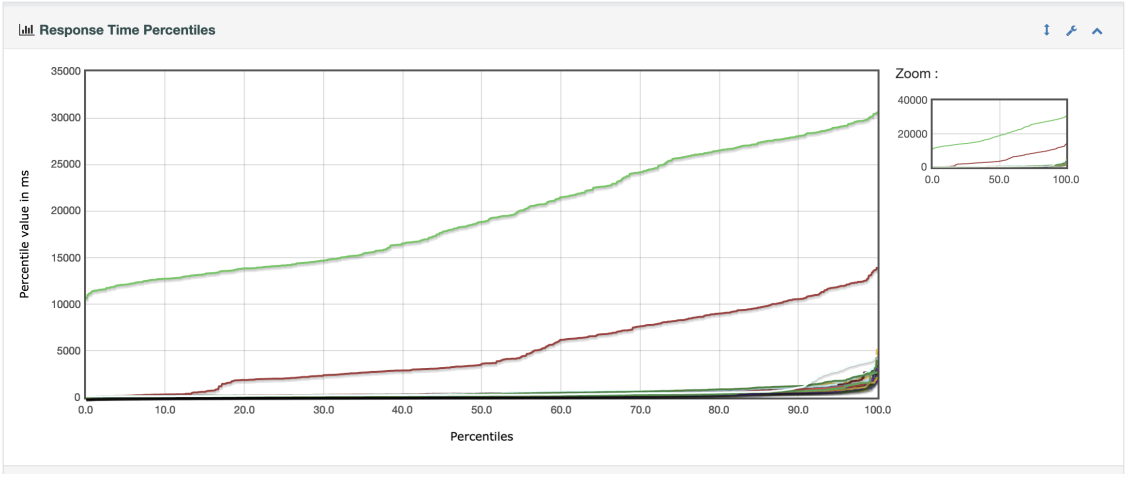

Figura 40: Teste 3 - *Multi-Machine* - 500 *threads*

Como se foi verificando anteriormente, mais uma vez o tempo de resposta aumentou significativamente em cada um dos testes de clientes. No entanto, conseguiu-se aumentar o débito de pedidos.

5.3 Monitorização Durante os Testes

A análise das métricas do sistema aquando dos testes com Jmeter, representados nas Figs. 41 e 42 para o teste mais impactante permitiram-nos perceber que o principal *bottleneck* do nosso sistema seria os servidores aplicacionais e portanto seriam esses que deviam ser replicados de modos a garantir um sistema mais confiável e tolerante a faltas.

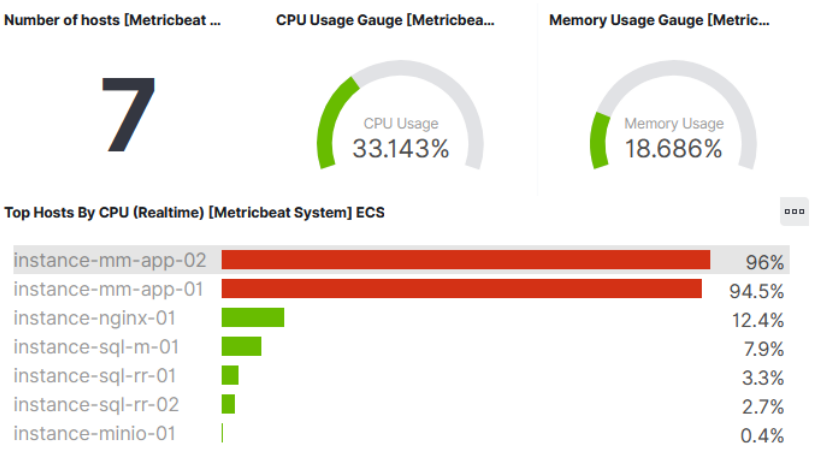

Figura 41: Dashboard Geral Kibana - CPU Utilizado durante os teste com Jmeter

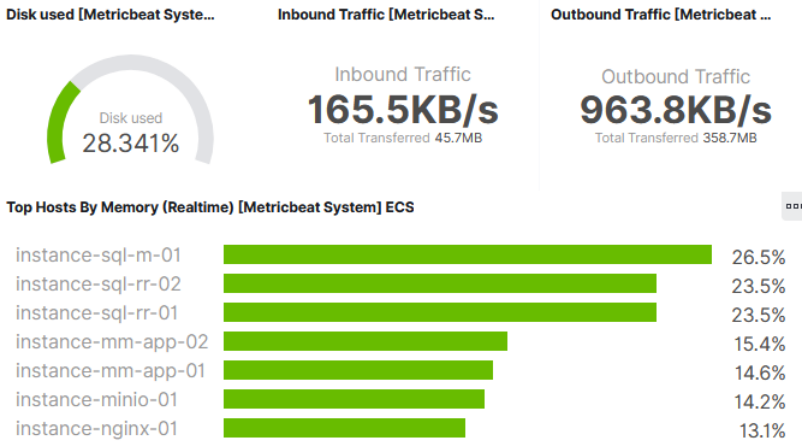

Figura 42: Dashboard Geral Kibana - Memória Utilizada durante os teste com Jmeter

6 Conclusão

O trabalho prático teve como objectivo final a criação de uma instalação automática da aplicação *Mattermost*, sendo que existiu uma fase posterior de análise onde foram identificados os pontos os únicos de falha de uma arquitectura *single-machine*, estudados os pontos de configuração do sistema e, por fim, analisadas várias arquitecturas possíveis, cada uma com diferentes níveis de disponibilidade.

Em relação a esta fase do trabalho, inicialmente foi apresentada a arquitectura a implementar, nomeadamente, *Multi-Machine*. Esta arquitectura, tal como foi explicado na primeira parte do trabalho prático, elimina os pontos únicos de falha acrescentando redundância a nível de dados e máquinas. No entanto, e dada a impossibilidade de criar mais do que oito máquinas no *Google Cloud Platform*, implementou-se apenas uma parte da arquitectura, onde só foram replicadas as máquinas que hospedam a base de dados e os servidores aplicacionais.

Numa segunda fase do trabalho, foi apresentada a lógica de implementação e automatização de cada camada implementada. O *Ansible* foi uma ferramenta crucial nesta fase uma vez que fornece um método de automatização bastante flexível e ágil. Para além disso, o facto de existir uma comunidade enorme à volta do *Ansible* fez com que certas configurações fossem mais rápidas de se realizar uma vez que foram usados módulos já feitos, como por exemplo, os módulos usados na replicação e na criação de utilizadores da base de dados. Um ponto menos positivo em relação a esta ferramenta foi o tempo de espera durante a fase de teste dos *scripts* de automatização.

De seguida, procedeu-se à análise de desempenho do sistema onde foi feita uma instalação básica do *Mattermost* para ter um nível de comparação com a aplicação de alta disponibilidade. As ferramentas usadas para os testes foram o *JMeter* e o *BlazeMeter*, onde esta última foi bastante útil na criação dos testes.

Relativamente aos resultados obtidos pode-se constatar que em alguns casos o response time subiu, no entanto temos de ter em conta que no caso single machine, todo o tráfego era direcionado à máquina local, não existindo outras camadas que agora existem. No caso do multi machine existem diversas camadas aos quais necessitamos de contactar para obter informação ou até para enviar informação, como é o caso dos *beats* presentes em cada máquina, o que causa mais um overhead extra. A base de dados também é replicada e é necessário garantir que a informação inserida está efectivamente escrita pelos slaves, sendo portanto outro factor que prejudica o response time.

Por fim, pode-se concluir que os resultados obtidos são satisfatórios ainda que não se tenha conseguido implementar a arquitectura objectivada.