

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Mateus Ferreira Gonçalves

**UM ARCABOUÇO PARA EXECUÇÃO DE
TESTES DE CARGA EM AMBIENTES
CONTAINERIZADOS**

Uberlândia, Brasil

2020

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Mateus Ferreira Gonçalves

UM ARCABOUÇO PARA EXECUÇÃO DE TESTES DE CARGA EM AMBIENTES CONTEINERIZADOS

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, Minas Gerais, como requisito exigido parcial à obtenção do grau de Bacharel em Sistemas de Informação.

Orientador: Rafael Pasquini

Universidade Federal de Uberlândia – UFU

Faculdade de Computação

Bacharelado em Sistemas de Informação

Uberlândia, Brasil

2020

Mateus Ferreira Gonçalves

UM ARCABOUÇO PARA EXECUÇÃO DE TESTES DE CARGA EM AMBIENTES CONTEINERIZADOS

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, Minas Gerais, como requisito exigido parcial à obtenção do grau de Bacharel em Sistemas de Informação.

Trabalho aprovado. Uberlândia, Brasil, 11 de dezembro de 2020:

Rafael Pasquini
Orientador

Paulo Rodolfo da Silva Leite Coelho

Rodrigo Sanches Miani

Uberlândia, Brasil
2020

*Dedico este trabalho a todos aqueles que me auxiliaram, principalmente meu orientador
Rafael Pasquini.*

Agradecimentos

Agradeço aos meus pais, aos meus professores, aos meus colegas e a todos aqueles que contribuíram com a minha formação não só como Profissional, mas como pessoa.

Resumo

Este trabalho propõe e cria um arcabouço para a realização de testes de carga em qualquer tipo de ambiente, o principal objetivo deste arcabouço é conseguir degradar sistemas em *cluster* mas isso não será limitado. O arcabouço poderá ser customizado de diversas formas, desde o serviço a ser testado, o ambiente a ser testado e a aplicação que será usada para testar. Ainda neste trabalho, avalia-se este arcabouço e verifica que o mesmo pode ser usado para degradar os ambientes, atendendo a parâmetros específicos e dependendo também do ambiente. Para efeito de validação, este trabalho avaliou o comportamento de um Banco de Dados (Apache Cassandra) e de uma API Web.

Para a avaliação dos resultados, foi usado principalmente o Tempo de Resposta das requisições, e o uso de CPU das máquinas utilizadas, sendo. Ao final da avaliação, os resultados demonstraram, que a medida que se modifica os parâmetros, as métricas tendem a mudar, registrando um maior tempo de resposta ou consumo de CPU, por exemplo. Eventualmente, acredita-se que poderia chegar a uma degradação total do sistema, porém, este trabalho não cobriu este cenário.

Palavras-chave: teste, docker, computação em nuvem, teste de carga, cluster.

Lista de ilustrações

Figura 1 – Representação da Onda Senoidal	17
Figura 2 – Modelando o <i>Flashcrowd</i>	18
Figura 3 – Computador vs Cluster	19
Figura 4 – Cluster vs Cluster	20
Figura 5 – Infraestrutura dos Serviços	25
Figura 6 – Tempo de Resposta (ms) Médio a cada 10 segundos para o Teste Cassandra Constante com 5 clientes	28
Figura 7 – Uso de CPU Médio a cada minuto para o Teste Cassandra Constante com 5 clientes	29
Figura 8 – Tempo de Resposta (ms) Médio a cada 10 segundos para o Teste Cassandra Constante com 10 clientes	30
Figura 9 – Uso de CPU Médio a cada minuto para o Teste Cassandra Constante com 10 clientes	31
Figura 10 – Tempo de Resposta (ms) Médio a cada 10 segundos para o Teste Cassandra Flashcrowd com 4 clientes e <i>shock_level 2</i>	32
Figura 11 – Uso de CPU Médio a cada minuto para o Teste Cassandra Flashcrowd com 4 clientes e <i>shock_level 2</i>	33
Figura 12 – Tempo de Resposta (ms) Médio a cada 10 segundos para o Teste Cassandra Flashcrowd com 4 clientes e <i>shock_level 3</i>	34
Figura 13 – Uso de CPU Médio a cada minuto para o Teste Cassandra Flashcrowd com 4 clientes e <i>shock_level 3</i>	35
Figura 14 – Tempo de Resposta (ms) Médio a cada 10 segundos para o Teste Cassandra Sinusoid com 7 clientes e amplitude 2	36
Figura 15 – Uso de CPU Médio a cada minuto para o Teste Cassandra Sinusoid com 7 clientes e amplitude 2	37
Figura 16 – Tempo de Resposta (ms) Médio a cada 10 segundos para o Teste Cassandra Sinusoid com 10 clientes e amplitude 5	38
Figura 17 – Uso de CPU Médio a cada minuto para o Teste Cassandra Sinusoid com 10 clientes e amplitude 5	39
Figura 18 – Tempo de Resposta (ms) Médio a cada 10 segundos para o Teste WEB API Constante com 5 clientes	41
Figura 19 – Uso de CPU Médio a cada minuto para o Teste WEB API Constante com 5 clientes	42
Figura 20 – Tempo de Resposta (ms) Médio a cada 10 segundos para o Teste Cassandra Constante com 10 clientes	43

Figura 21 – Uso de CPU Médio a cada minuto para o Teste Cassandra Constante com 10 clientes	44
Figura 22 – Tempo de Resposta (ms) Médio a cada 10 segundos para o Teste WEB API Flashcrowd com 10 clientes e <i>shock_level</i> 3	45
Figura 23 – Uso de CPU Médio a cada minuto para o Teste WEB API Flashcrowd com 10 clientes e <i>shock_level</i> 3	46
Figura 24 – Tempo de Resposta (ms) Médio a cada 10 segundos para o Teste WEB API Flashcrowd com 10 clientes e <i>shock_level</i> 5	47
Figura 25 – Uso de CPU Médio a cada minuto para o Teste WEB API Flashcrowd com 10 clientes e <i>shock_level</i> 5	48
Figura 26 – Tempo de Resposta (ms) Médio a cada 10 segundos para o Teste WEB API Sinusoid com 20 clientes e amplitude 10	49
Figura 27 – Uso de CPU Médio a cada minuto para o Teste WEB API Sinusoid com 20 clientes e amplitude 10	50
Figura 28 – Tempo de Resposta (ms) Médio a cada 10 segundos para o Teste WEB API Sinusoid com 30 clientes e amplitude 20	51
Figura 29 – Uso de CPU Médio a cada minuto para o Teste WEB API Sinusoid com 30 clientes e amplitude 20	52

Sumário

1	INTRODUÇÃO	10
1.1	Motivação	11
1.2	Objetivos e Desafios da Pesquisa	12
1.3	Hipótese: Testes de Estresse em um Serviço em <i>Cluster</i>	12
1.4	Organização	12
2	FUNDAMENTAÇÃO TEÓRICA	13
2.1	Containerização de Aplicações e o <i>Docker</i>	13
2.1.1	O Funcionamento do <i>Docker</i>	13
2.1.2	Orquestração de Contêineres <i>Docker</i>	14
2.2	Testes de Carga	15
2.2.1	Métodos de Carga	16
2.2.1.1	Carga Constante	16
2.2.1.2	Carga Senoidal	16
2.2.1.3	<i>Flashcrowd</i>	16
3	PROPOSTA	19
3.1	Contextualização	19
3.2	Proposta	20
3.3	Interface de Cluster	21
3.3.1	Métodos da Interface	21
3.3.1.1	create(deployment_name, name, image, args, mounts, replicas)	21
3.3.1.2	scale(deployment_name, name, image, args, mounts, replicas)	21
3.3.1.3	remove(deployment_name)	22
3.3.2	Interfaces a Serem Implementadas	22
3.3.2.1	Kubernetes	22
3.4	Interface de Algoritmo	22
3.4.1	Métodos da Interface	23
3.4.1.1	run(cluster, args)	23
3.4.2	Interfaces a Serem Implementadas	23
3.4.2.1	Método de Carga Constante	23
3.4.2.2	Método de Carga Senoidal	23
3.4.2.3	Método de Carga FlashCrowd	23
3.5	Executor (Main)	24
4	EXPERIMENTOS E ANÁLISE DOS RESULTADOS	25

4.1	Serviços Testados	25
4.2	Serviços de Testes	26
4.3	Cluster	26
4.4	Métodos de Carga	27
4.5	Método para a Avaliação	27
4.6	Experimentos	27
4.6.1	Cassandra	28
4.6.1.1	Constante - Teste 1	28
4.6.1.2	Constante - Teste 2	29
4.6.1.3	Flashcrowd - Teste 1	31
4.6.1.4	Flashcrowd - Teste 2	33
4.6.1.5	Sinusoid - Teste 1	36
4.6.1.6	Sinusoid - Teste 2	37
4.6.1.7	Avaliação dos Testes	40
4.6.1.8	Tempo de Resposta	40
4.6.1.9	Uso de CPU	40
4.6.2	WEB API	41
4.6.2.1	Constante - Teste 1	41
4.6.2.2	Constante - Teste 2	42
4.6.2.3	Flashcrowd - Teste 1	44
4.6.2.4	Flashcrowd - Teste 2	46
4.6.2.5	Sinusoid - Teste 1	49
4.6.2.6	Sinusoid - Teste 2	50
4.6.2.7	Avaliação dos Resultados	53
4.6.2.8	Tempo de Resposta	53
4.6.2.9	Uso de CPU	53
5	CONCLUSÃO	54
5.1	Trabalhos Futuros	54
5.1.1	Padronização da coleta de métricas e geração de resultados	54
5.1.2	Criação de uma Interface Web para gestão dos cenários	55
5.1.3	Criação automática de infraestrutura para testes	55
5.1.4	Explorar outras métricas ou métodos	55
	REFERÊNCIAS	56

1 Introdução

[Taurion \(2009\)](#) descreve a Computação em Nuvem como uma imensa rede de servidores, sejam eles virtuais ou físicos. E ainda define-a como a evolução da virtualização, a virtualização do Data Center. Algumas empresas começaram na vanguarda e [Taurion \(2009\)](#) define a Amazon como a principal empresa a explorar este mercado, tornando-se uma das protagonistas.

Para [Santos \(2018\)](#) o modelo de computação em nuvem apresentou uma nova maneira de produzir, entregar e consumir os recursos de TI por meio da Internet. [Taurion \(2009\)](#) também comenta que a Computação em Nuvem melhora a utilização dos recursos computacionais, diminui o *time-to-market*, já que se ganha mais tempo diminuindo a gestão da infraestrutura e focando na geração de valor.

Atualmente, tanto serviços a aplicações completas podem ser utilizadas em nuvem, tanto [Santos \(2018\)](#), como [Veras \(2012\)](#) descrevem três modelos de computação em nuvem:

- ***Iaas - Infrastructure as a Service***: Armazenamento e processamento fornecido de forma transparente, que é o caso de instâncias virtuais. O Amazon EC2 é um exemplo deste tipo de serviço.
- ***Paas - Platform as a Service***: Pode se realizar a implementação de uma aplicação dentro da estrutura, mas não há controle sobre a infraestrutura. O Heroku é uma plataforma de Paas.
- ***Saas - Software as a Service***: Um serviço disponibilizado de forma remota no qual não se controla nenhum recurso de infraestrutura. Aplicativos de compartilhamentos de arquivo como Google Drive ou Dropbox são considerados *Saas*.

Impulsionada pela nuvem, novas tecnologias ganham força, duas delas são importantes para este trabalho, sendo os contêineres, propriamente dito e a orquestração de contêineres. [Khan \(2017\)](#) define os contêineres como o encapsulamento de um processo executando em um sistema operacional, num ambiente privado com limitação de recursos, incluindo memória e CPU. Já a Orquestração de Contêineres, [Khan \(2017\)](#) a descreve como um sistema que provê um *framework* para integrar e gerenciar contêineres de forma empresarial.

[Khan \(2017\)](#) também define alguns itens que todo orquestrador deve possuir:

- 1. Gerenciamento de Estado do Cluster

- 2. Alta disponibilidade e tolerância a falhas
- 3. Garantir segurança
- 4. Simplificar a Rede
- 5. Permitir a descoberta de serviços
- 6. Permitir a Entrega Contínua
- 7. Prover monitoramento e governança

Apesar do custo da computação em nuvem poder ser pago de forma mensal, como Santos (2018) explica, pode-se então ter a possibilidade de testar diversos ambientes e redimensioná-los de acordo com o uso, pois mesmo a cobrança sendo mensal, ainda há custo. Tendo tudo isto em vista, antes de criar ambientes deve ser feita uma avaliação mínima para definir o tamanho inicial da infraestrutura, esta avaliação poderia ser feita por um profissional que presta este tipo de serviço, porém um gerador de estresse poderia ser usado para calibrar uma infraestrutura e definir o menor tamanho viável de um Cluster. Com um gerador de carga dinâmico, as empresas e pessoas poderiam se beneficiar não só na validação do ambiente e se o mesmo é resistente num momento de estresse, mas também para verificar qual o ambiente mínimo para um número X de usuários.

1.1 Motivação

Na primeira metade da década de 2010, no mundo da computação, surge uma nova tecnologia. O *Docker* como foi chamado pela então *DotCloud*, foi rapidamente tornado código aberto (Matthias; Kane, 2015). Em vez de virtualizar um sistema operacional inteiro como as máquinas virtuais faziam, o *Docker* se propõe a virtualizar aplicações dentro de um mesmo sistema operacional utilizando os recursos do sistema operacional que o mesmo está sendo executado. (TURNBULL, 2014)

Com a popularização do *Docker*, foi possível o surgimento de ferramentas de Orquestração de Contêineres, como o *Kubernetes* da Google (KHAN, 2017). Essas ferramentas, como (KHAN, 2017) possui uma funcionalidade importante: alta disponibilidade e tolerância a falhas, além de outras citadas na Seção 1, o que facilitou a gestão de ambientes e aplicações em Cluster.

Este trabalho verificará, se mesmo estes ambientes orquestrados, que devem ser altamente disponíveis, podem ser degradados. Para tal, será utilizado os mesmos mecanismos destes *Clusters*, isto é, Orquestração de Contêineres, que neste caso, será a Orquestração das Aplicações de Teste que tentarão fazer carga em sistemas em Cluster.

1.2 Objetivos e Desafios da Pesquisa

O objetivo deste trabalho é construir um arcabouço para a realização de testes de carga, então, será conduzido um experimento, para verificar se aplicações que utilizam mecanismos de orquestração e containerização podem ser degradados, e se puderem, quais são os parâmetros necessários para que o mesmo ocorra.

1.3 Hipótese: Testes de Estresse em um Serviço em *Cluster*

Um ou mais serviços serão testados sem nenhuma restrição de banda de rede. As seguintes perguntas deverão ser verificadas:

- É possível degradar um serviço orquestrado?
- Quais os parâmetros para degradar um serviço orquestrado?
- É necessário realizar alguma intervenção para que o serviço seja estressado (Diminuição de banda de rede nos servidores, por exemplo) ?

1.4 Organização

Este documento está dividido em cinco capítulos, no Capítulo 1 descreve-se o problema, a motivação deste trabalho e os objetivos. No Capítulo 2 é feito o referencial teórico do trabalho explicando sobre contêineres, orquestração e testes de carga. No Capítulo 3 apresentamos a proposta do experimento, e no Capítulo 4, apresentamos a execução do experimento e os resultados obtidos. Por fim, no Capítulo 5, apresentamos a conclusão do Trabalho.

2 Fundamentação Teórica

O presente capítulo fornece as principais definições para o entendimento do trabalho, bem como o *Docker*, a Orquestração de Contêineres e Testes de Carga.

2.1 Containerização de Aplicações e o *Docker*

A containerização de aplicações em si, é algo que sempre foi possível nos ambientes Linux, porém, mesmo sendo possível, não era fácil de ser utilizado antes da chegada do *Docker* (Matthias; Kane, 2015). Até então, não havia uma ferramenta que se destacasse na implementação de contêineres.

O *Docker* foi criado pela empresa *DotCloud Inc.* (atualmente *Docker Inc*) como um *framework* para a sua plataforma de serviços (HOLLA, 2015). Nas primeiras semanas em que foi anunciado, recebeu bastante atenção da mídia e apenas alguns meses depois, as empresas começaram a ouvir sobre ele e a forma com que ele funcionava. (Matthias; Kane, 2015)

Para Matthias e Kane (2015), o principal objetivo de uma implementação de contêineres, seria simplificar a coordenação e escalabilidade aplicações, o *Docker* não só cumpre este objetivo, como traz uma forma padronizada de definir as aplicações.

O uso do *Docker* permite que os desenvolvedores se concentrem em construir as aplicações e executá-las dentro de um contêiner (HOLLA, 2015). Funcionando diferente de uma virtualização de máquina onde apenas recursos mínimos são compartilhados, no *Docker* os recursos são compartilhados a nível do Sistema Operacional economizando assim recursos (HOLLA, 2015).

Desde o sua criação em 2013, o *Docker* se tornou um sucesso, uma pesquisa da *Portworx* em 2019 (PORTWORX, 2019), aponta que cerca de 87% das empresas pesquisadas estavam utilizando *Docker*, e dessas que utilizavam *Docker*, 89% o faziam em ambientes de Produção.

2.1.1 O Funcionamento do *Docker*

Para o *Docker* (DOCKER, 2020b), um contêiner é uma unidade de software padronizada que junta a aplicação e todas as suas dependências para que a aplicação possa executar de forma rápida e confiável.

O *Docker* em si está pautado e três conceitos:

- **Padronização:** O *Docker* criou a padronização para a containerização, permitindo a portabilidade.
- **Leve:** Os contêineres compartilham o mesmo Sistema Operacional, levando a eficiência dos servidores e diminuindo o custo de licenças.
- **Segurança:** As aplicações são mais seguras dentro dos contêineres pois o *Docker* provê, por padrão, uma forte isolamento.

[Matthias e Kane \(2015\)](#) definem a terminologia essencial do *Docker* como:

- **Servidor *Docker*:** Um *daemon* do *Docker* que torna um sistema Linux em um servidor *Docker* que pode possuir contêineres publicados e executados via cliente *Docker*.
- **Imagens *Docker*:** Consiste em arquivos juntamente com metadados que representam todos os arquivos necessários para a containerização de uma aplicação. Um contêiner geralmente irá possuir um nome e uma *tag* que geralmente é utilizada para identificar uma versão específica da aplicação.
- **Contêiner *Docker*:** Um contêiner *Docker* é um contêiner em Linux que foi instanciado a partir de uma Imagem *Docker*, o contêiner em si é único, mas vários contêineres podem ser criados a partir de uma única imagem.

Uma imagem *Docker* pode ser armazenada em um serviço de Registro *Docker*. O *Docker Hub*, por exemplo, é um serviço ofertado pela própria *Docker Inc* que permite publicar imagens *Docker* de forma pública. Além da própria *Docker Inc*, as principais empresas de computação em nuvem, como *AWS*, *Azure*, *Google Cloud* e *Digital Ocean* oferecem esse tipo de serviço.

2.1.2 Orquestração de Contêineres *Docker*

Com pequenos pedaços de aplicações, definidas em imagens *Docker*, passamos a ter a possibilidade de gerenciar e escalar a execução destas imagens em um ambiente em *Cluster*.

Este tipo de serviço, é geralmente desempenhado pelos chamados Orquestradores. Os orquestradores podem reiniciar contêineres com falha, e aumentar ou diminuir a capacidade do sistema com operações de escalabilidade.

Dois serviços de Orquestração conhecidos são o *Kubernetes* e o *Docker Swarm*, falaremos brevemente sobre alguns deles.

- ***Docker Swarm*:** O *Swarm* ([DOCKER, 2020a](#)), diferentemente dos demais orquestradores, é embutido dentro do próprio *Docker* e ofertado pela *Docker Inc*, toda

instalação *Docker* vem com o *Swarm* instalado mas não habilitado por padrão. Um *Cluster Swarm* é formado por instâncias de servidores executando o *Docker* em modo *Swarm*. O *Swarm* funciona basicamente da seguinte forma:

- Nós: São instâncias do *Docker* em modo *Swarm* que se conectam entre si, na prática, uma máquina física poderia executar mais de uma instância, mas não é algo recomendado para um ambiente produtivo. Os nós podem ainda ser do tipo *workers*, que apenas executam as tarefas, e *managers* que coordenam a execução, mas também podem ser executores.
 - Serviços: Um serviço é a definição das tarefas que serão executadas nos *workers* e é a estrutura central de um orquestrador *Swarm*.
 - Tarefa: A tarefa define a imagem *Docker* e as instruções que serão executadas por essa imagem, como argumentos por exemplo.
 - Balanceamento de Carga: O *Swarm* pode expor externamente os serviços e trata de balancear as chamadas para as diferentes instâncias em execução.
- *Kubernetes*: O *Kubernetes* ([KUBERNETES, 2020](#)) é um serviço de código aberto para a Orquestração de Contêineres. O *Kubernetes* foi criado pelo Google, que apresenta a ferramenta como resultado de "15 anos de experimento" em orquestração de contêineres. O *Kubernetes* tende a ser mais complexo que o *Swarm*, porém alguns conceitos básicos são:
 - Nó: Um nó é uma instância do *Kubernetes*, de forma que possa executar *Pods*.
 - *Pod*: Um *Pod* é a menor unidade de um *Cluster Kubernetes*. Os *Pods* definem os contêineres que serão executados, assim como o compartilhamento de recursos entre eles.
 - *Deployment*: Um *Deployment* define o estado desejado de execução dos *Pods*, os *deployments* podem ser usados para escalar serviços.
 - AWS ECS: O *AWS Elastic Container Service* também é um serviço de orquestração de contêineres da AWS. O serviço funciona de forma similar ao *Swarm* e *Kubernetes* porém com menores níveis de customização.

Existem ainda outros serviços disponíveis no mercado para uso, mas com funcionalidades similares, sendo que o objetivo principal destes serviços, é a gestão dos contêineres de forma a permitir a escalabilidade.

2.2 Testes de Carga

O Teste de Carga é um tipo de teste de Software geralmente inserido no contexto de Testes de Performance. Testes de Carga geralmente são utilizados para avaliar o

comportamento de um sistema quando exposto a um número alto de usuários acessando a um sistema. Ao realizar este tipo de teste, podemos definir métricas para avaliar o comportamento da aplicação.

(MOLYNEAUX, 2014) apresenta quatro possíveis métricas que podem ser utilizadas:

- **Disponibilidade:** O tempo em que uma aplicação está disponível para o usuário final.
- **Tempo de Resposta:** O tempo que uma aplicação leva para responder a uma requisição de um cliente.
- **Vazão:** A periodicidade em que os eventos ocorrem.
- **Utilização:** O percentual teórico de recursos utilizados pela aplicação.

2.2.1 Métodos de Carga

Alguns softwares de teste de carga, geralmente utilizam um número constante de usuários simultâneos para medir o comportamento do sistema, o Apache JMeter, por exemplo, utiliza este modelo (JMETER, 2021). Este trabalho propõe e implementa outros métodos de carga, que podem ser baseados em trabalhos relacionados ou não. Neste contexto, a seguir é descrito os métodos que poderiam ser utilizados.

2.2.1.1 Carga Constante

Neste tipo de Método de Carga, simulamos um número de usuários constante, alto o suficiente, para expor a degradação do sistema. Este tipo de teste pode parecer simples, mas pode ser usado para definir um "comportamento base" do sistema.

2.2.1.2 Carga Senoidal

Neste método, o número de clientes muda seguindo a forma de uma curva senoidal. Neste tipo de método, um número de clientes se inicia num valor N , e toma também como argumento uma amplitude A , onde o máximo de clientes é $N+A$ e o mínimo é $N-A$. A Figura 1 apresenta o comportamento da Onda Senoidal.

2.2.1.3 Flashcrowd

O *Flashcrowd*, é um flash, isto é, um número de acessos inesperados em um site web (Ari et al., 2003), porém podemos generalizar para qualquer serviço exposto, seja um site web, uma *API Rest* ou até mesmo um Banco de Dados. No *Flashcrowd*, um número de acessos pré definido ocorre inesperadamente.

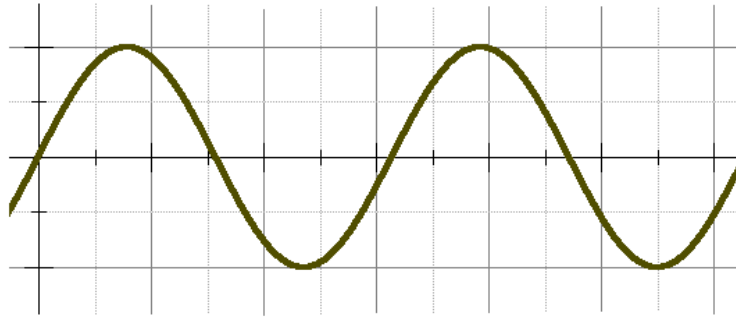


Figura 1 – Representação da Onda Senoidal

(COSTA et al., 2016)

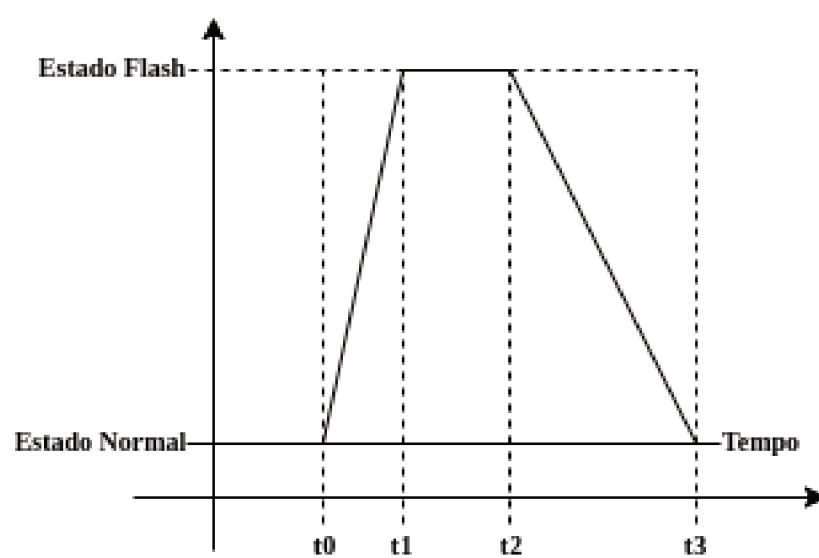
No modelo de estresse de *Flashcrowd*, há basicamente dois estados, sendo:

- **Estado Normal:** Neste estado, o número de clientes está num valor inicial que seria a média de clientes que geralmente utilizam aquele serviço.
- **Estado Flash:** Neste estado, o Flash foi atingido e o número de clientes atinge o valor máximo.

Este método, inicia-se no *Estado Normal*, e em intervalos regulares deve atingir *Estado Flash*, permanecer em *Estado Normal* por algum tempo e depois voltar a *Estado Normal*, para isso, define-se três momentos.

- **Ramp-Up:** Tempo para sair do Estado Normal e chegar no Estado Flash.
- **Sustained-Time:** Tempo em que o número de clientes permanece no Estado Flash.
- **Ramp-Down:** Tempo para sair do Estado Flash e voltar ao Estado Normal.

A Figura 2 detalha os momentos t que podemos ter no Modelo de *Flashcrowd*. Sendo que entre t_0 e t_1 estaremos no momento de *Ramp-Up*, de t_1 a t_2 em *Sustained-Time*, e de t_2 a t_3 em *Ramp-Down*.

Figura 2 – Modelando o *Flashcrowd*

(Ari et al., 2003)

3 Proposta

3.1 Contextualização

Atualmente existem tecnologias que permitem a execução de testes de carga em sistemas, sendo que alguns podem funcionar de forma distribuída, que é o caso do [Locust](#), e outros não, como o [Apache JMeter](#).

O objetivo deste trabalho não é comparar estas ferramentas, mas como já dito, construir um Arcabouço totalmente extensível que possa executar testes de carga em quaisquer tipos de sistema.

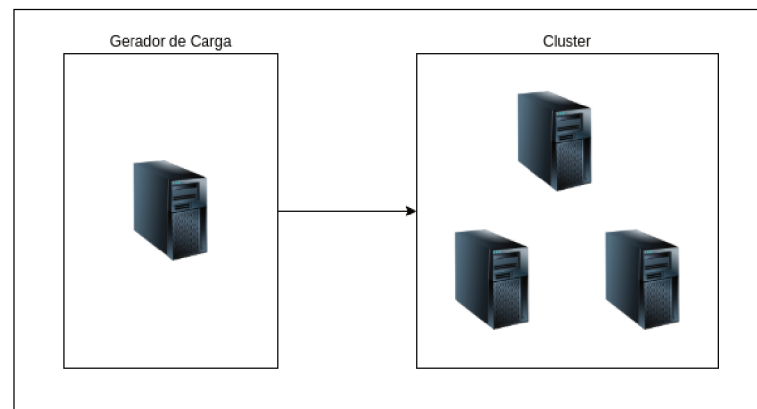


Figura 3 – Computador vs Cluster

Na Figura 3 podemos ver um primeiro cenário no qual apenas uma máquina geraria carga para um Cluster, este cenário não foi tratado neste trabalho e apenas partimos de um teste deste tipo para construir o arcabouço. Para que a carga seja efetiva, geralmente será necessário que os recursos computacionais desta máquina sejam iguais ou superior aos recursos do Cluster somado.

Caso tivéssemos poder computacional suficiente, talvez poderíamos estressar o ambiente, por ser uma única máquina seria também mais simples executar várias instâncias da aplicação testadora, mas ainda assim teríamos a limitação de tráfego, além de que possuir uma máquina poderia incorrer em custos desnecessários já que provavelmente esta máquina não seria usada a todo momento já que os testes são executados de forma controlada.

Já num segundo cenário, descrito na Figura 4, teríamos um conjunto de máquinas, isto é, um Cluster, executando as aplicações de testes e gerando estresse no Cluster de Serviços. Este cenário é o objetivo deste trabalho e foi implementado.

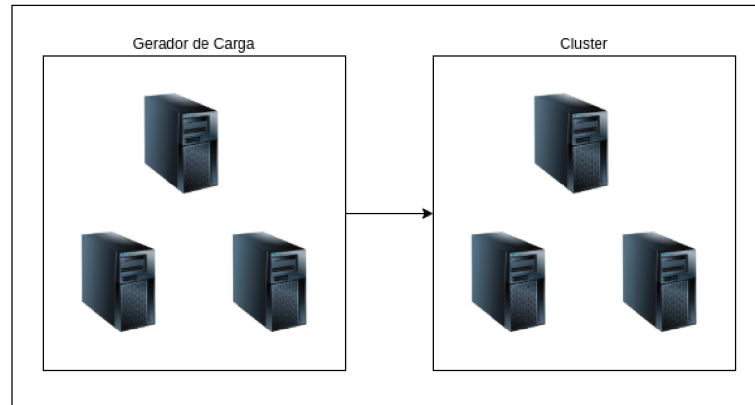


Figura 4 – Cluster vs Cluster

Neste tipo de cenário, há um custo inicial de se ter o software do Cluster em si executando, já que numa máquina normal o processo poderia executar diretamente, enquanto que num Cluster, exige-se as ferramentas mínimas que compõem o Cluster. Porém, num ambiente os recursos computacionais são virtualmente ilimitados, já que se necessário mais recursos, podemos apenas adicionar mais nós ao Cluster, além de se não existir mais o problema de rede já que a carga é gerada não só por uma máquina.

3.2 Proposta

Para este trabalho, foi criado um arcabouço para execução de testes em um ambiente em *Cluster*. A biblioteca permite que qualquer tipo de ambiente seja testado e estressado utilizando métodos de carga.

Para a construção desta biblioteca foi utilizada a linguagem de Programação *Python* na versão 3, sendo implementado em sua maior parte de forma procedural. A biblioteca faz uso de Imagens *Docker* que são publicadas em um servidor de Orquestração e poderão ser executados os métodos de carga.

No arcabouço do projeto, foram implementadas três entidades principais:

- **Cluster:** esta entidade implementa os métodos necessários para a execução num determinado tipo de Cluster, os detalhes de implementação serão descritos abaixo.
- **Método:** esta entidade implementa o método de carga, este algoritmo pode ser desde o mais simples como o Carga Constante ou um mais complexo como o *Flash-crowd*.
- **Main:** a principal entidade do arcabouço, esta entidade é o ponto de entrada da biblioteca e a partir dos parâmetros, será definida por ela, qual *Cluster* e método de carga serão utilizados.

3.3 Interface de Cluster

A interface de *Cluster* deverá ser implementada para permitir que um determinado tipo Cluster, *Docker Swarm*, por exemplo, possa ser estressado, ou seja, com este arcabouço, temos a possibilidade de estressar qualquer ambiente em Cluster que possua funções de escala.

3.3.1 Métodos da Interface

3.3.1.1 `create(deployment_name, name, image, args, mounts, replicas)`

Este método define a criação do serviço de teste orquestrado no servidor em *Cluster*. Este método será chamado no início do Teste de Carga e os parâmetros representam o estado inicial do servidor.

A seguir, uma breve descrição dos parâmetros deste método:

- **deployment_name**: define o nome do serviço no ambiente em Cluster. Num *Cluster* como o *Kubernetes* seria o nome de uma *Deployment*, por exemplo, já em um *Cluster* de *Docker Swarm* refletiria um *Service*.
- **name**: Define o nome ou alias da imagem Docker.
- **image**: nome da Imagem *Docker* que será utilizada para os testes.
- **args**: quaisquer argumentos que serão passados no momento da execução para os contêineres.
- **mounts**: define os pontos de montagem de Sistema de Arquivos, entre o contêiner em execução e o servidor hospedeiro.
- **replicas**: define o número de instâncias iniciais em que o serviço de testes será iniciado.

3.3.1.2 `scale(deployment_name, name, image, args, mounts, replicas)`

Este método define a escalabilidade do serviço de teste orquestrado no servidor em *Cluster*. A utilização deste método presume que o serviço já existe no *Cluster* e pode ser utilizando tanto para aumentar o número de clientes (*Scale-Up*), quanto diminuir (*Scale-Down*)

A seguir, uma breve descrição dos parâmetros deste método:

- **deployment_name**: define o nome do serviço no ambiente em Cluster.
- **name**: define o nome ou alias da imagem *Docker*.

- **image**: nome da Imagem *Docker* que será utilizada para os testes.
- **args**: quaisquer argumentos que serão passados no momento da execução para os contêineres.
- **mounts**: define os pontos de montagem de Sistema de Arquivos, entre o contêiner em execução e o servidor hospedeiro.
- **replicas**: define o número de clientes para que o serviço possa ser escalado.

3.3.1.3 remove(deployment_name)

Este método define a remoção serviço de teste orquestrado no servidor em *Cluster*. Este método será utilizado ao fim do teste para remover as instâncias em execução.

A seguir, uma breve descrição dos parâmetros deste método:

- **deployment_name**: define o nome do serviço no ambiente em *Cluster*.

3.3.2 Interfaces a Serem Implementadas

3.3.2.1 Kubernetes

Para o escopo deste experimento, apenas o *Cluster Kubernetes* será implementado, ele foi escolhido dentre os demais possíveis orquestradores, abaixo alguns dos motivos para sua escolha:

- Ser o Orquestrador mais utilizado atualmente e o que possui a maior comunidade online.
- Possuir um alto nível de customização.
- Possuir uma *API* já implementada e difundida em Python.

3.4 Interface de Algoritmo

Esta interface contém a implementação de um determinado método de teste, num método de teste Senoidal, por exemplo, o cálculo da curva senoidal, bem como o número de clientes em cada momento da curva será definido por esta interface. Ela também será responsável por fazer as operações de criação e escala no *cluster*. Com esta Interface sendo genérica, a biblioteca permite que métodos de carga distintos sejam implementados e utilizados.

3.4.1 Métodos da Interface

3.4.1.1 `run(cluster, args)`

Este método define a execução do algoritmo de teste, o cálculo da quantidade de clientes e os momentos em que serão efetuadas as escalas é de responsabilidade deste método.

A seguir, uma breve descrição dos parâmetros deste método:

- **cluster**: a interface de *Cluster* que será utilizada para as operações de escala.
- **args**: Quaisquer argumentos necessários para os cálculos do algoritmo e ainda os argumentos para os contêineres que serão executados.

3.4.2 Interfaces a Serem Implementadas

3.4.2.1 Método de Carga Constante

O método de carga constante, apenas mantém um número de clientes constantes, do início ao fim do teste. Este método não requer nenhum parâmetro adicional.

3.4.2.2 Método de Carga Senoidal

O método de carga senoidal simula uma onda senoide. Este método requer os seguintes parâmetros:

- **amplitude**: define a amplitude da onda senoidal. A amplitude define o quanto o valor máximo e mínimo que curva chegará a partir de um valor de clientes *N*.
- **period**: define a periodicidade da senoide, isto é, o tempo que se leva, no início da senoide até o retorno do estado inicial da mesma.

3.4.2.3 Método de Carga FlashCrowd

O método de carga *Flashcrowd*, simula uma onda de acessos repentino em um serviço. Este método requer os seguintes parâmetros especiais:

- **period**: define todo o tempo de um flash no *Flashcrowd*. Ou seja, é todo o tempo entre sair de R-Normal, ir para R-Flash e voltar para R-Normal novamente.
- **shock_level**: define o nível de choque que será dado em cada Flash, este parâmetro é um fator multiplicador para o número de clientes, por exemplo, se **shock_level** for igual a 2, e o número de clientes inicial for 50, o número de clientes em *R-Flash* será 100.

- **occurrences**: o número de eventos de Flash que ocorrerão na duração do teste.

3.5 Executor (Main)

Este será o *Script* principal da biblioteca, a partir dele que serão coordenadas as execuções do *Cluster* e dos métodos de carga.

Como entidade principal, todos os parâmetros de entrada necessários para as demais entidades deverão ser passados, a seguir uma lista de parâmetros e uma breve descrição sobre cada um.

- **cluster**: a implementação da Interface de *Cluster* a ser utilizada. Exemplo: *kubernetes*, *Swarm*.
- **algorithm**: a implementação do método de teste a ser utilizado. Exemplo: *sinusoid*, *flashcrowd*.
- **image**: a imagem *Docker* a ser utilizada para a execução dos testes.
- **name**: o nome do teste que será executado, geralmente será o mesmo da aplicação a ser testada.
- **duration**: duração da execução dos testes.
- **dockerargs**: os argumentos que serão passados aos contêineres a serem executados.
- **args**: Os argumentos que serão passados ao algoritmo de teste.

4 Experimentos e Análise dos Resultados

Após a criação do Arcabouço, alguns testes foram conduzidos para avaliar o funcionamento dos métodos de carga em diferentes ambientes, com diferentes parâmetros.

Para o escopo deste experimento, dois serviços foram testados: Um Banco de Dados não relacional (Apache Cassandra) e uma Web *API* simples implementada em Python.

4.1 Serviços Testados

Apache Cassandra: O Cassandra é um banco de dados de Código aberto que executa de forma distribuída, ele utiliza um modelo de particionamento em colunas, com consistência eventual.

Web API: Uma Web *API* simples, construída em Python utilizando a biblioteca Flask, com apenas um endpoint.

O serviços foram orquestrados em um *Cluster Kubernetes* utilizando a nuvem do Google, GCP (Google Cloud Computing).

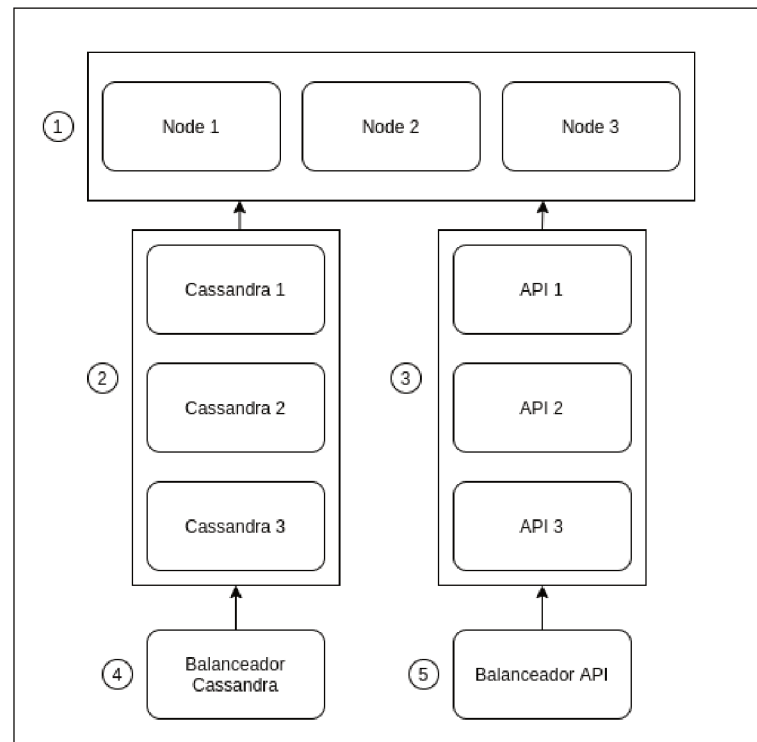


Figura 5 – Infraestrutura dos Serviços

- 1. Os nós do cluster: O *Cluster* foi construído utilizando um pool de 3 máquinas

do tipo *n1-standard-2*, estas máquinas possuem 2 CPUS virtualizadas e 7,5 GB de Memória RAM.

- 2. Serviço Cassandra: O Serviço do Cassandra, foi publicado no Kubernetes, utilizando 3 réplicas, que serão executadas nos nós do *Cluster*.
- 3. Serviço *API*: O Serviço de Web API, também foi publicado utilizando 3 réplicas.
- 4. Balanceador de Carga Cassandra: Um Balanceador de Carga para redirecionar as requisições para uma das 3 réplicas em execução. Há de notar, que mesmo balanceado, o serviço do Cassandra trabalha de forma distribuída.
- 5. Balanceador de Carga *API*: Um Balanceador de Carga para redirecionar as requisições para uma das 3 réplicas em execução.

O diagrama da infraestrutura ilustrado na Figura 5 foi simplificado, apresentando apenas os recursos pertinentes ao experimento, existem outros recursos presentes na infraestrutura como *Deployments* e *ReplicaSets*, mas estes foram retirados para evitar confusão. A seguir uma descrição de cada elemento apresentado no diagrama:

4.2 Serviços de Testes

Para cada serviço testado, o Arcabouço desenvolvido exige uma aplicação testadora. Esta aplicação testadora deverá ser transformada em um contêiner, para que seja possível a execução dos testes. Para os serviços testados no escopo deste trabalho, utilizamos as seguintes aplicações testadoras:

Cassandra Stress Tool: O [Cassandra Stress Tool](#) é um utilitário construído em Java que permite realizar benchmarking de modelos de dados no Cassandra, para este trabalho, o utilitário foi *dockerizado* de forma a permitir a sua execução em *Cluster* utilizando o Arcabouço construído.

Apache JMeter: O [JMeter](#) é uma ferramenta de código aberto construída em Java, que permite realizar testes de carga, originalmente foi desenvolvido com o foco em aplicações Web, mas nos dias atuais é possível testar diversos serviços com ele. O JMeter também foi *dockerizado* e um plano de testes básico foi criado, de forma a permitir o teste de carga na Web API.

4.3 Cluster

O *Cluster* utilizado neste experimento foi o Kubernetes, já foi evidenciado no Capítulo 3 os motivos do seu uso, e para construir a infraestrutura foi utilizado serviço em nuvem da Google: Google Cloud Platform.

4.4 Métodos de Carga

Serão utilizados para os testes os algoritmos:

Constante: Neste método, o numero de clientes permanece estático durante a execução.

Sinusoid: Neste método, o numero de clientes aumenta ou diminui de acordo com uma onda senoidal.

Flashcrowd: Neste método, o número de clientes aumenta de forma elevada em ondas de *flash*.

4.5 Método para a Avaliação

Para avaliar o comportamento dos serviços, em cada aplicação a ser testada, foram realizadas três etapas de testes.

- **Etapla 1:** dois testes foram executados utilizando o método **Constante** para gerar valores de comparação.
- **Etapla 2:** dois testes foram executados utilizando o método **Flashcrowd**.
- **Etapla 3:** dois testes foram executados utilizando o método **Sinusoid**.

Os resultados foram coletados a partir de uma instância adicional de execução, esta instância foi criada a parte pois em alguns métodos, as instâncias podem ser removidas durante a a execução do teste, enquanto que ela ficará executando a todo momento. Além disto, também foi coletada métricas relacionadas a infraestrutura.

Para a avaliação dos resultados, foram utilizadas as métricas que não precisavam de implementações muito complexas para realizar a sua coleta, sendo elas:

- **Tempo de Resposta:** o tempo de resposta entre a requisição e a resposta do servidor;
- **Uso de CPU:** o uso de CPU dos servidores do *Cluster*.

4.6 Experimentos

A seguir iremos apresentar os experimentos realizados com cada serviço, e os algoritmos utilizados.

4.6.1 Cassandra

4.6.1.1 Constante - Teste 1

Este primeiro teste, coleta os dados iniciais da plataforma, sem estresse e sem utilizar métodos específicos que fazem carga. Para o primeiro teste do Cassandra, após alguns testes, decidiu-se que seria utilizado cinco clientes de forma constante. Este teste também terá um tempo de trinta minutos, valor também decidido após a realização de testes anteriores.

A seguir, está sumarizada os parâmetros de execução:

Duração	30 minutos
Número de Clientes	5

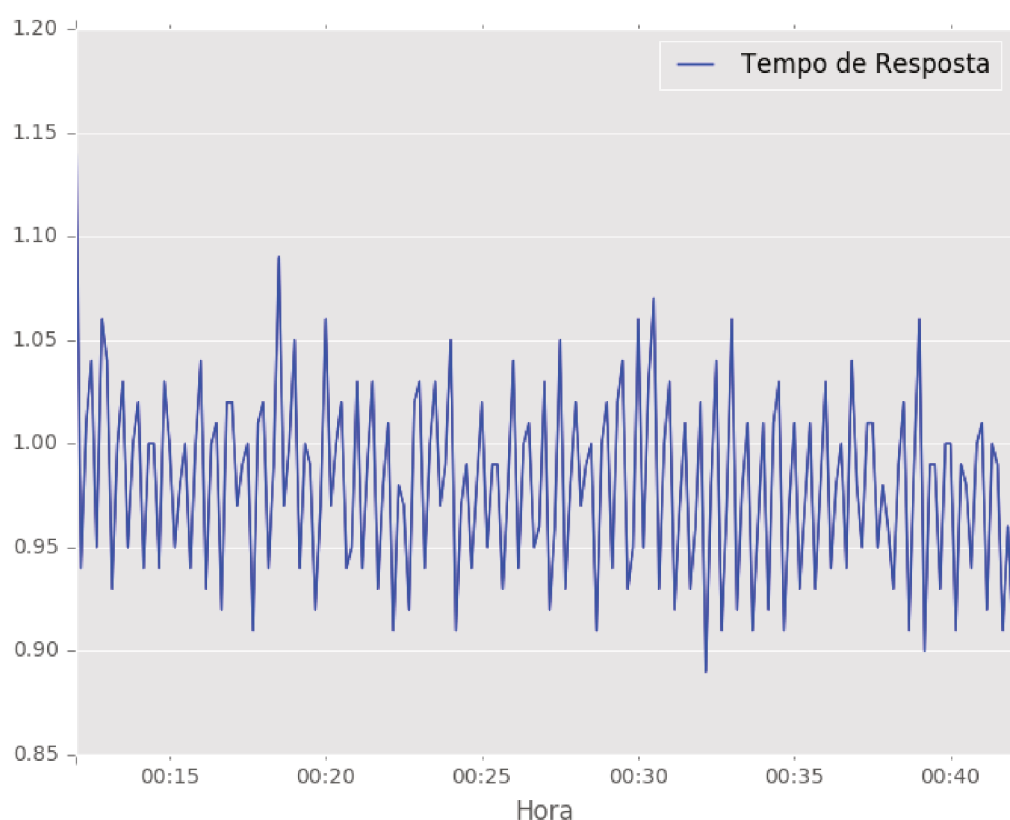


Figura 6 – Tempo de Resposta (ms) Médio a cada 10 segundos para o Teste Cassandra Constante com 5 clientes

A Figura 6 apresenta o tempo de resposta durante a execução no qual foram coletados os seguintes valores:

- Tempo de Resposta (média): 0,98

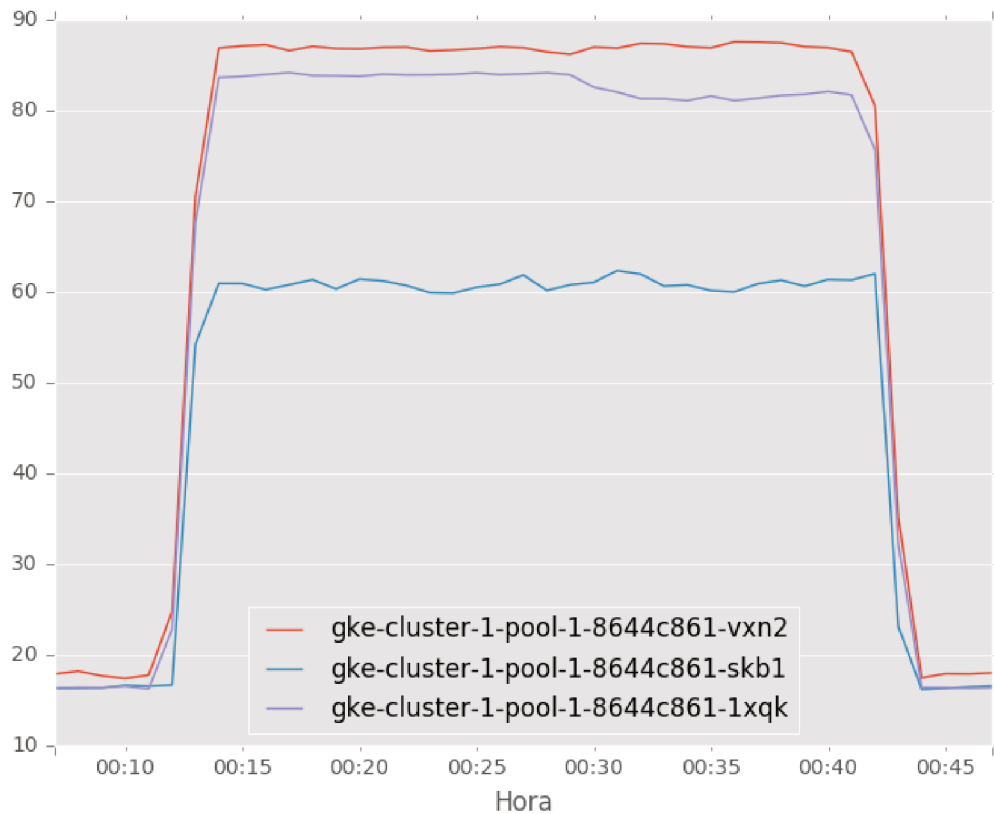


Figura 7 – Uso de CPU Médio a cada minuto para o Teste Cassandra Constante com 5 clientes

- Tempo de Resposta (desvio padrão): 0,10

A Figura 7 demonstra o uso de CPU durante a execução no qual foram coletados os seguintes valores:

- CPU 1 (média): 84,21
- CPU 1 (desvio padrão): 11,48
- CPU 2 (média): 59,27
- CPU 2 (desvio padrão): 8,02
- CPU 3 (média): 80,28
- CPU 3 (desvio padrão): 11,15

4.6.1.2 Constante - Teste 2

Este segundo teste, coleta os dados iniciais da plataforma, sem estresse e sem utilizar métodos específicos que fazem carga. Este segundo teste do método constante,

também foi realizado para verificar se ocorrem diferenças nas métricas, se o número de clientes é aumentado. Para este teste, foi definido o uso de 10 clientes.

A seguir, está sumarizada os parâmetros de execução:

Duração	30 minutos
Número de Clientes	10

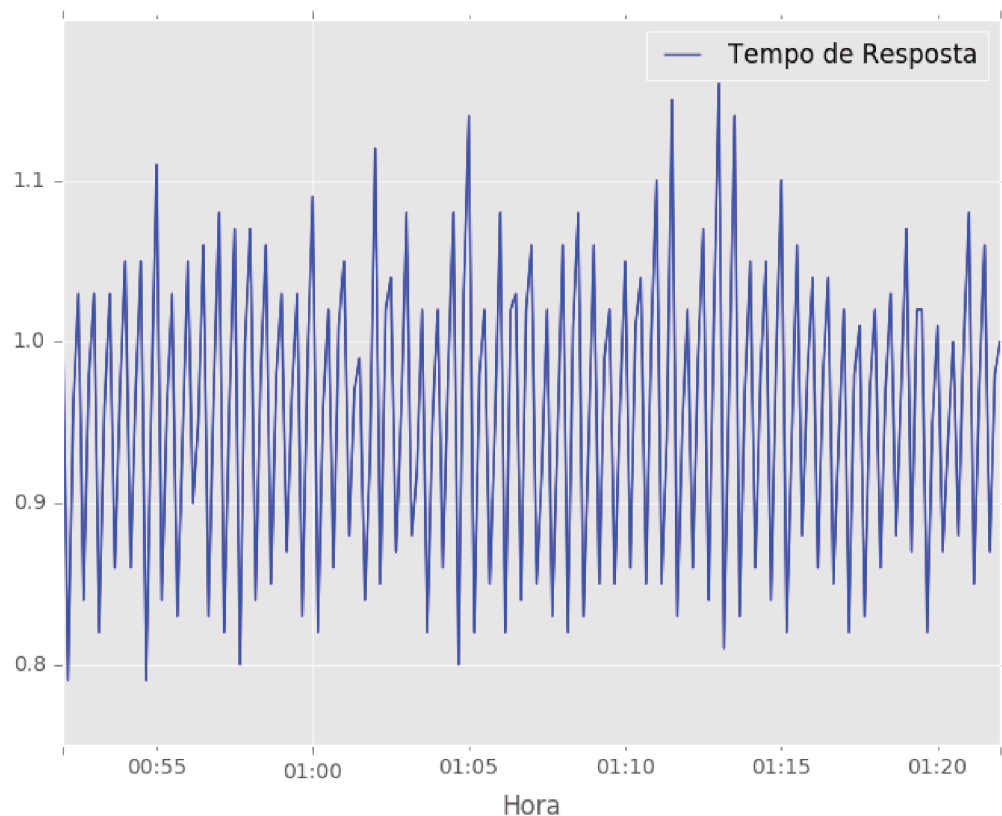


Figura 8 – Tempo de Resposta (ms) Médio a cada 10 segundos para o Teste Cassandra Constante com 10 clientes

A Figura 8 demonstra o tempo de resposta durante a execução no qual foram coletados os seguintes valores:

- Tempo de Resposta (média): 0,96
- Tempo de Resposta (desvio padrão): 0,20

A Figura 9 apresenta o uso de CPU durante a execução no qual foram coletados os seguintes valores:

- CPU 1 (média): 89,60

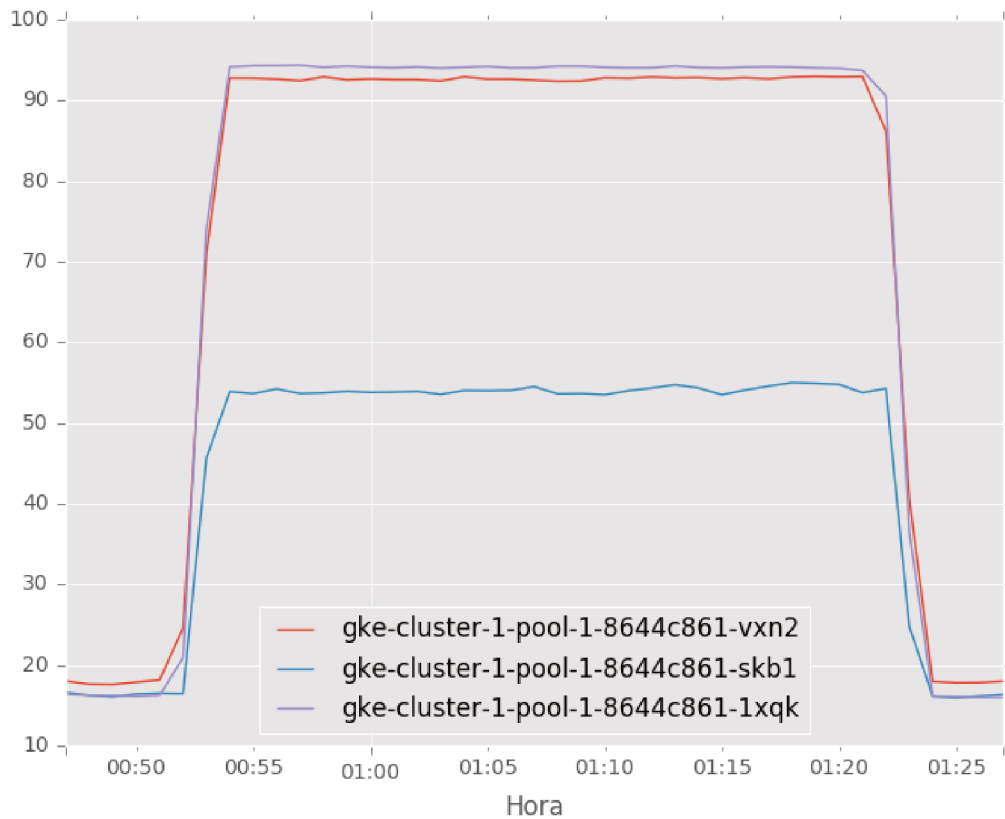


Figura 9 – Uso de CPU Médio a cada minuto para o Teste Cassandra Constante com 10 clientes

- CPU 1 (desvio padrão): 12,73
- CPU 2 (média): 52,59
- CPU 2 (desvio padrão): 6,89
- CPU 3 (média): 91,00
- CPU 3 (desvio padrão): 13,52

Em relação ao teste descrito na Seção 4.6.1.1, apesar de não observado um aumento no Tempo de Resposta, inclusive foi menor, observa-se um aumento explícito no uso da CPU 3, aumentando em quase 11%. Outro fator, observado foi que os dados coletados neste Teste (desvio padrão = 0,20) são mais dispersos do que o teste apresentado na Seção 4.6.1.1 (desvio padrão = 0,1).

4.6.1.3 Flashcrowd - Teste 1

Este teste foi executado para avaliar o comportamento do sistema utilizando o método de carga Flashcrowd em vez da carga constante. Para o primeiro teste foi definido

o número de clientes como 4 e o valor da *Shock Level* em 2, ou seja, nos momentos de Flash, haviam 8 clientes. Na duração, diferentemente do teste Constante, este teste teve duração de 60 minutos, pois os 30 minutos não eram suficientes para as várias Ondas de Flash.

A seguir, está sumarizada os parâmetros de execução:

Duração	60 minutos
Clientes	4
Shock Level	2
Ocorrências	3
Período	15

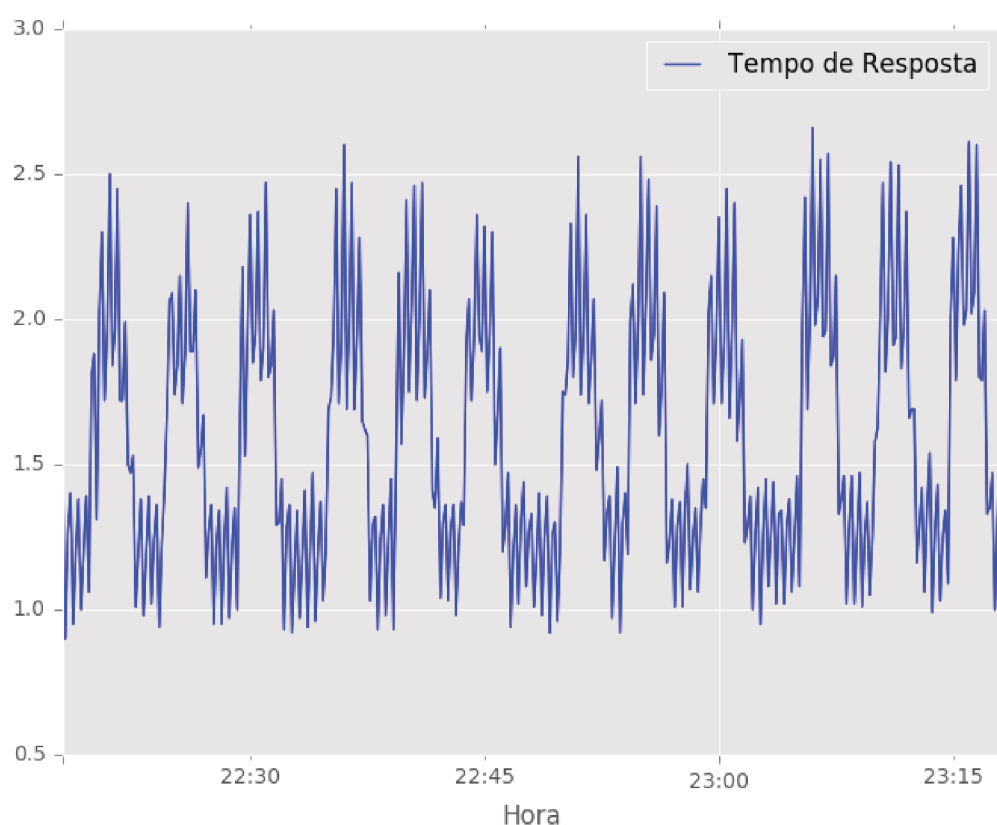


Figura 10 – Tempo de Resposta (ms) Médio a cada 10 segundos para o Teste Cassandra Flashcrowd com 4 clientes e *shock_level* 2

A Figura 10 demonstra o tempo de resposta durante a execução no qual foram coletados os seguintes valores:

- Tempo de Resposta (média): 1,59

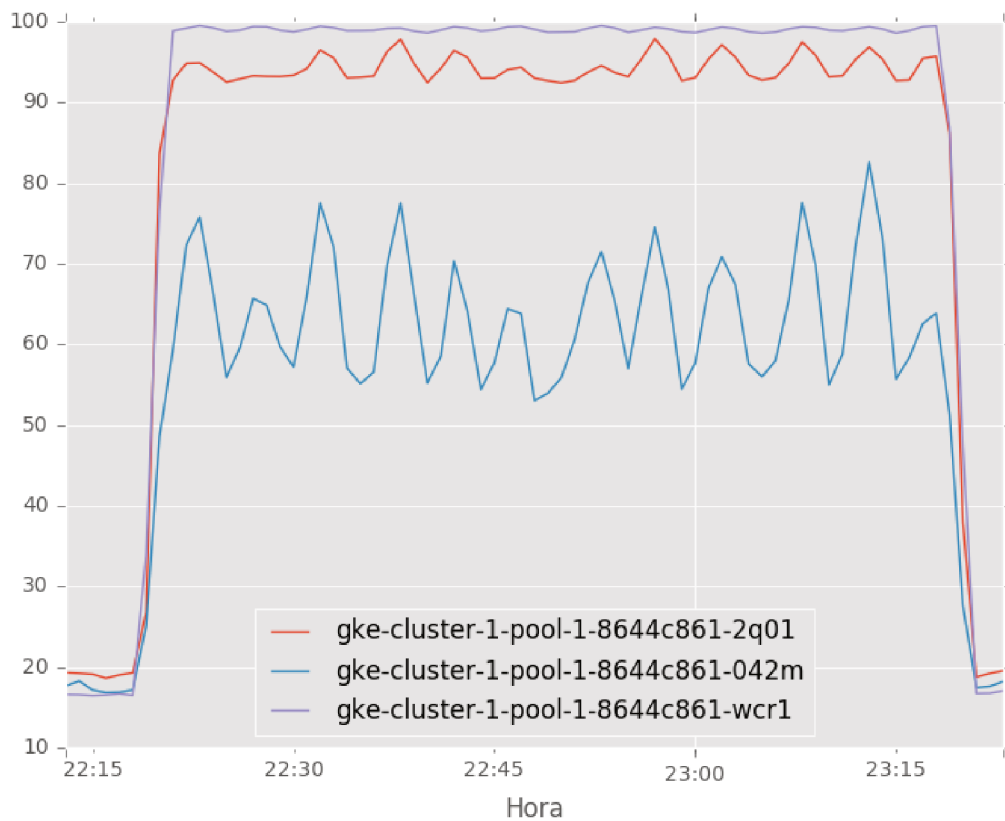


Figura 11 – Uso de CPU Médio a cada minuto para o Teste Cassandra Flashcrowd com 4 clientes e *shock_level* 2

- Tempo de Resposta (desvio padrão): 0,53

A Figura 11 apresenta o uso de CPU durante a execução no qual foram coletados os seguintes valores:

- CPU 1 (média): 91,80
- CPU 1 (desvio padrão): 12,94
- CPU 2 (média): 62,14
- CPU 2 (desvio padrão): 10,70
- CPU 3 (média): 96,28
- CPU 3 (desvio padrão): 13,59

4.6.1.4 Flashcrowd - Teste 2

Este teste foi executado avaliar o comportamento do sistema utilizando o método de carga Flashcrowd em vez da carga constante, além de ser comparado com o teste

descrito na Seção 4.6.1.3. Para o segundo teste apenas foi modificado o valor da *Onda de Flash* para 3.

A seguir, está sumarizada os parâmetros de execução:

Duração	60 minutos
Clientes	4
Shock Level	3
Ocorrências	3
Período	15

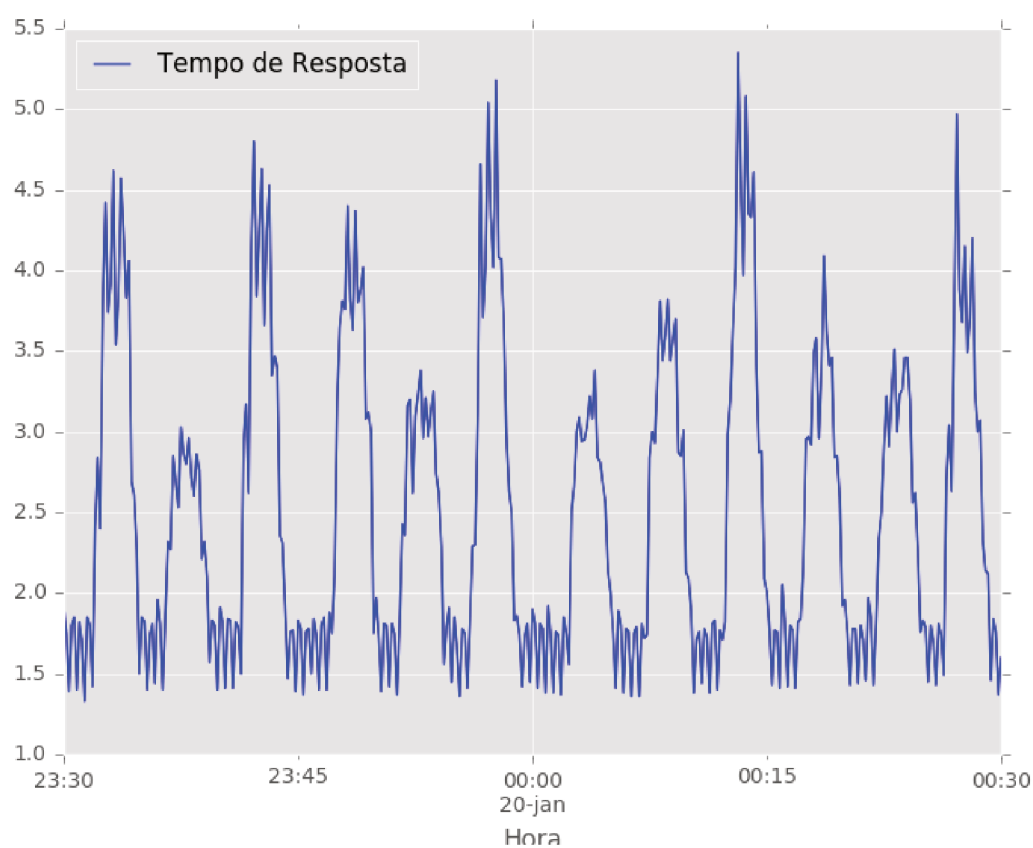


Figura 12 – Tempo de Resposta (ms) Médio a cada 10 segundos para o Teste Cassandra Flashcrowd com 4 clientes e *shock_level* 3

A Figura 12 demonstra o tempo de resposta durante a execução no qual foram coletados os seguintes valores:

- Tempo de Resposta (média): 2,52
- Tempo de Resposta (desvio padrão): 1,05

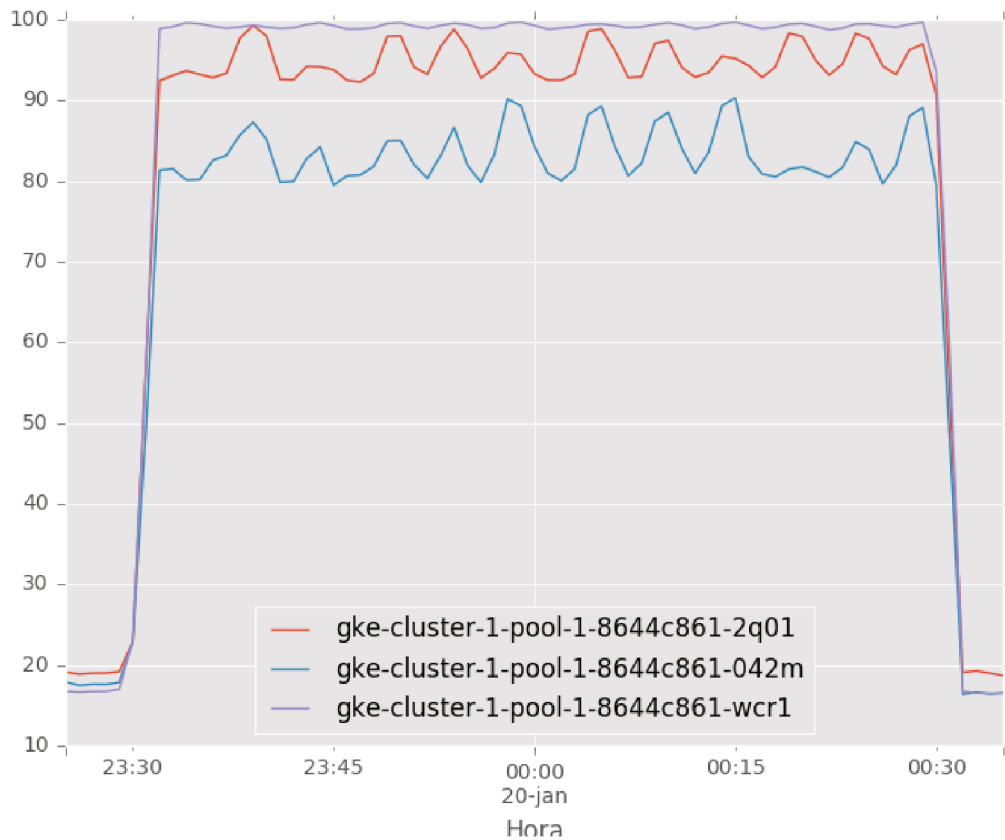


Figura 13 – Uso de CPU Médio a cada minuto para o Teste Cassandra Flashcrowd com 4 clientes e *shock_level* 3

A Figura 13 demonstra o uso de CPU durante a execução no qual foram coletados os seguintes valores:

- CPU 1 (média): 93,08
- CPU 1 (desvio padrão): 10,53
- CPU 2 (média): 81,80
- CPU 2 (desvio padrão): 9,33
- CPU 3 (média): 97,22
- CPU 3 (desvio padrão): 11,06

Foi possível observar que em relação ao teste apresentado na Seção 4.6.1.3 aumentou-se a média do Tempo de Resposta (2,52 contra 1,59), além de uma maior dispersão dos dados (1,05 contra 0,52). Também foi observado que o uso da CPU 2 aumentou consideravelmente, em quase 20%.

4.6.1.5 Sinusoid - Teste 1

Este teste foi executado para avaliar o comportamento do sistema utilizando o método de carga Sinusoid. Para o primeiro teste foi definido o número de clientes como 7 e o valor da amplitude em 3. Na duração, diferentemente do teste Constante, este teste teve duração de 60 minutos, pois os 30 minutos não eram suficientes para observar as ondas senoidais.

A seguir, está sumarizada os parâmetros de execução:

Duração	60 minutos
Cientes	7
Amplitude	2
Período	20

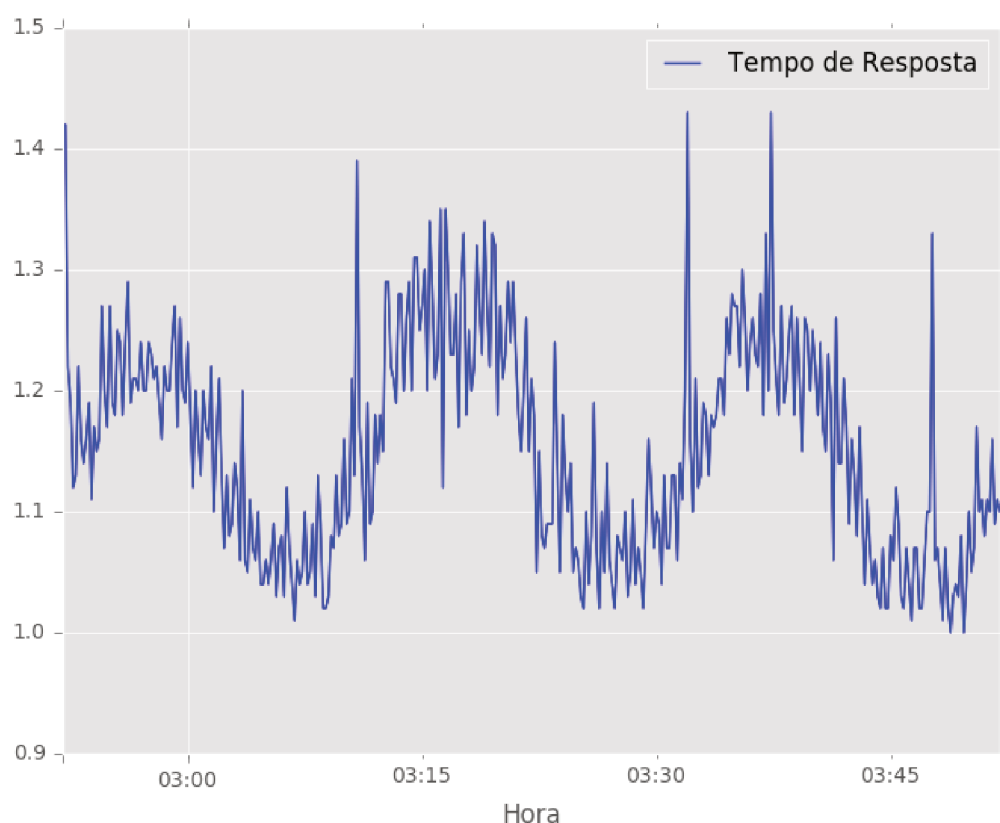


Figura 14 – Tempo de Resposta (ms) Médio a cada 10 segundos para o Teste Cassandra Sinusoid com 7 clientes e amplitude 2

A Figura 14 demonstra o tempo de resposta durante a execução no qual foram coletados os seguintes valores:

- Tempo de Resposta (média): 1,15

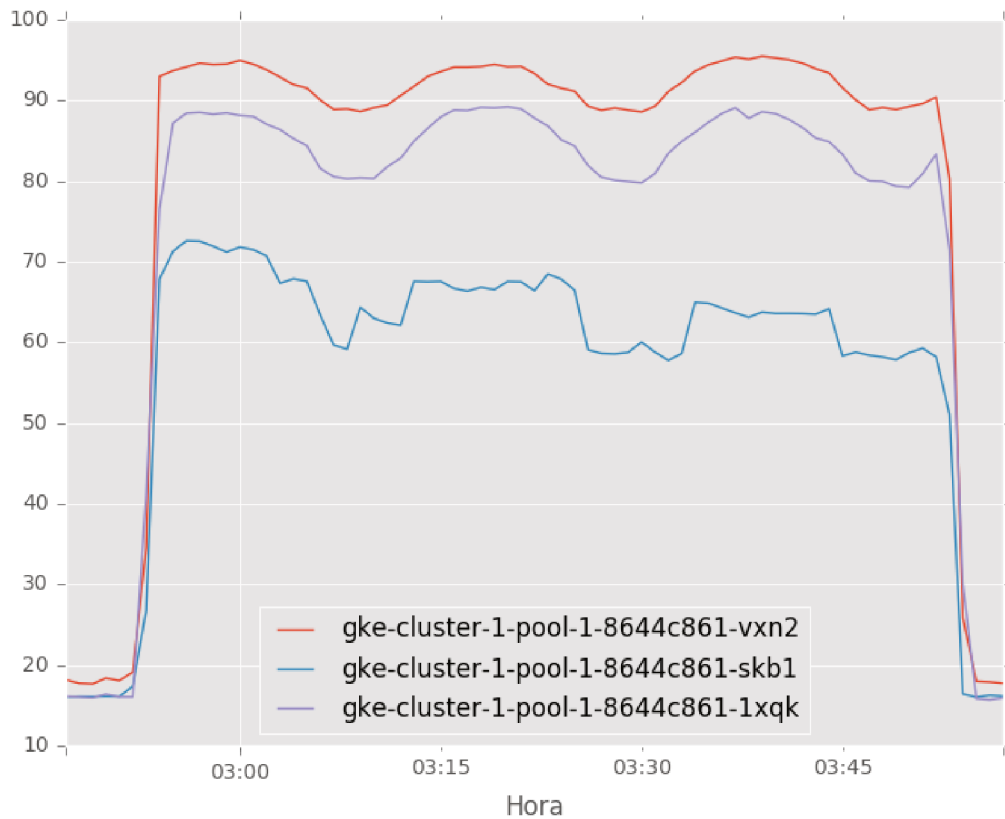


Figura 15 – Uso de CPU Médio a cada minuto para o Teste Cassandra Sinusoid com 7 clientes e amplitude 2

- Tempo de Resposta (desvio padrão): 0,16

A Figura 15 apresenta o uso de CPU durante a execução no qual foram coletados os seguintes valores:

- CPU 1 (média): 90,02
- CPU 1 (desvio padrão): 12,94
- CPU 2 (média): 62,90
- CPU 2 (desvio padrão): 10,70
- CPU 3 (média): 82,91
- CPU 3 (desvio padrão): 13,59

4.6.1.6 Sinusoid - Teste 2

Este teste foi executado para avaliar o comportamento do sistema utilizando o método de carga Sinusoid e também para comparar com o teste descrito na Seção 4.6.1.6.

Para o segundo teste foi definido o número de clientes foi aumentado para 10 e o valor da amplitude para 5.

A seguir, está sumarizada os parâmetros de execução:

Duração	60 minutos
Cientes	10
Amplitude	5
Período	20

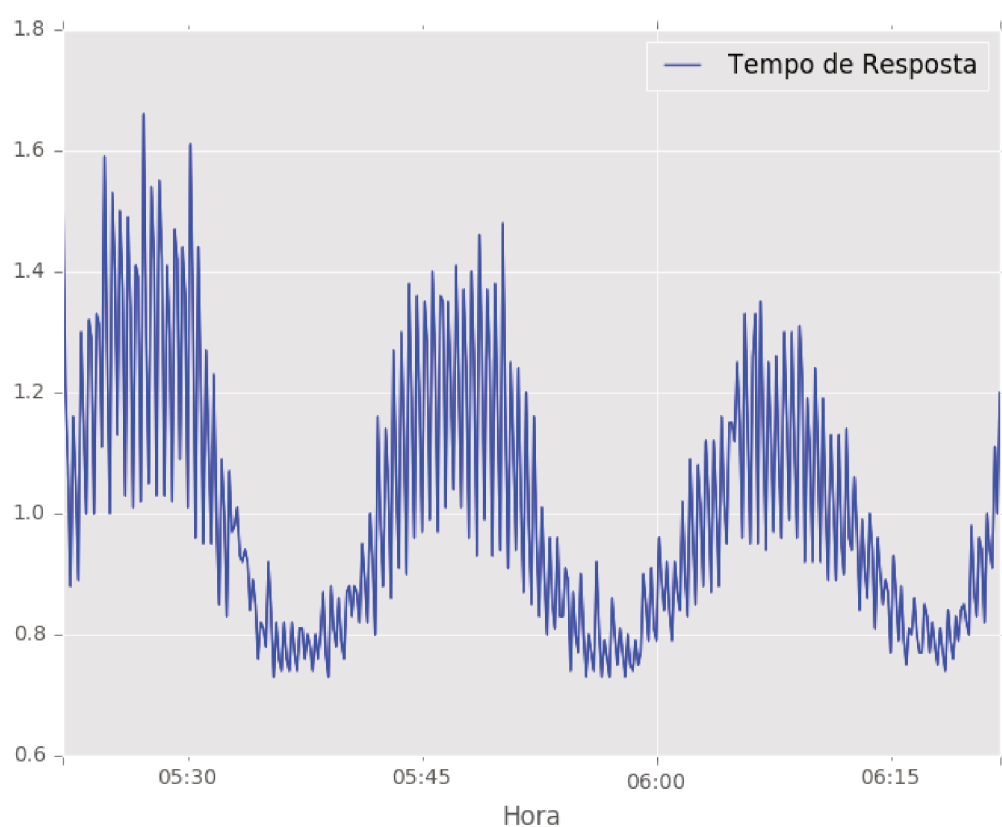


Figura 16 – Tempo de Resposta (ms) Médio a cada 10 segundos para o Teste Cassandra Sinusoid com 10 clientes e amplitude 5

A Figura 16 demonstra o tempo de resposta durante a execução no qual foram coletados os seguintes valores:

- Tempo de Resposta (média): 1,00
- Tempo de Resposta (desvio padrão): 0,28

A Figura 17 apresenta o uso de CPU durante a execução no qual foram coletados os seguintes valores:

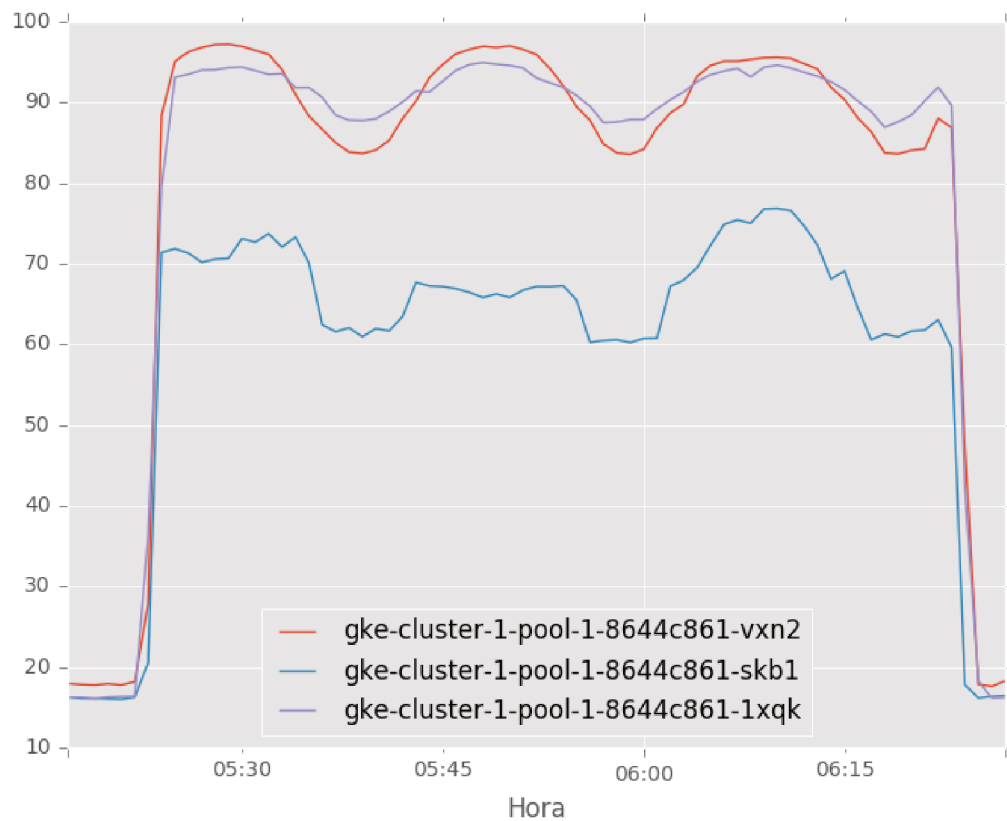


Figura 17 – Uso de CPU Médio a cada minuto para o Teste Cassandra Sinusoid com 10 clientes e amplitude 5

- CPU 1 (média): 88,93
- CPU 1 (desvio padrão): 13,18
- CPU 2 (média): 65,78
- CPU 2 (desvio padrão): 10,14
- CPU 3 (média): 89,33
- CPU 3 (desvio padrão): 12,18

Foi observado que em relação ao teste 4.6.1.5 houve uma diminuição no Tempo de Resposta (1,00 contra 1,15), porém com uma maior dispersão dos dados (0,28 contra 0,16). Outro fator observado foi que o consumo de CPU diminuiu na pouco na CPU 1 (88,93 contra 90,02), porém aumentou na CPU 2 (65,78 contra 62,90) e na CPU 3 (89,33 contra 82,91).

4.6.1.7 Avaliação dos Testes

Foi observado que nos Testes Constante descrito nas Seções 4.6.1.1 e 4.6.1.2 e Sinusoid descrito nas Seções 4.6.1.5 e 4.6.1.6 há uma maior homogeneidade do Tempo de Resposta, em relação aos testes do Flashcrowd apresentados nas Seções 4.6.1.3 e 4.6.1.4. Também é possível inferir que o serviço do Cassandra utiliza algum mecanismo para manter o tempo de resposta num limiar aumentando o consumo de CPU, o que parece ter funcionado nos testes Constante e Sinusoid onde o Tempo de Resposta medido quase não difere entre os testes, ou é até menor mas há maior consumo de CPU, isto não é observado nos testes do Flashcrowd, provavelmente porquê o a Onda de Flash é rápida de mais para que o Cassandra possa ativar o mecanismo. No fim, este teste foi efetivo em demonstrar que ao calibrar-se os parâmetros, podemos observar diferenças nas métricas escolhidas.

4.6.1.8 Tempo de Resposta

Teste	Média	Desvio Padrão
Constante 1	0,98	0,10
Constante 2	0,96	0,20
Flashcrowd 1	1,59	0,53
Flashcrowd 2	2,52	1,05
Sinusoid 1	1,15	0,16
Sinusoid 2	1,00	0,28

4.6.1.9 Uso de CPU

Teste	M* (1)***	DP** (1)	M (2)	DP (2)	M (3)	DP (3)
Constante 1	84,21	11,48	59,27	8,02	80,28	11,15
Constante 2	89,60	12,73	52,59	6,89	91,00	13,52
Flashcrowd 1	91,80	12,94	62,14	10,70	96,28	13,59
Flashcrowd 2	93,08	10,53	81,80	9,33	97,22	11,06
Sinusoid 1	90,02	12,04	62,90	8,83	82,91	10,91
Sinusoid 2	88,93	13,18	65,78	10,14	89,33	12,18

*M se refere a média.

**DP Se refere ao Desvio Padrão.

***Os valores em parênteses representam as CPUs.

4.6.2 WEB API

4.6.2.1 Constante - Teste 1

Este primeiro teste, coleta os dados iniciais da plataforma, sem estresse e sem utilizar métodos específicos que fazem carga. Para o primeiro teste da WEB API, após alguns testes, decidiu-se que seria utilizado cinco clientes de forma constante. Este teste também terá um tempo de trinta minutos, valor também decidido após a realização de testes anteriores.

A seguir, está sumarizada os parâmetros de execução:

Duração	30 minutos
Número de Clientes	5

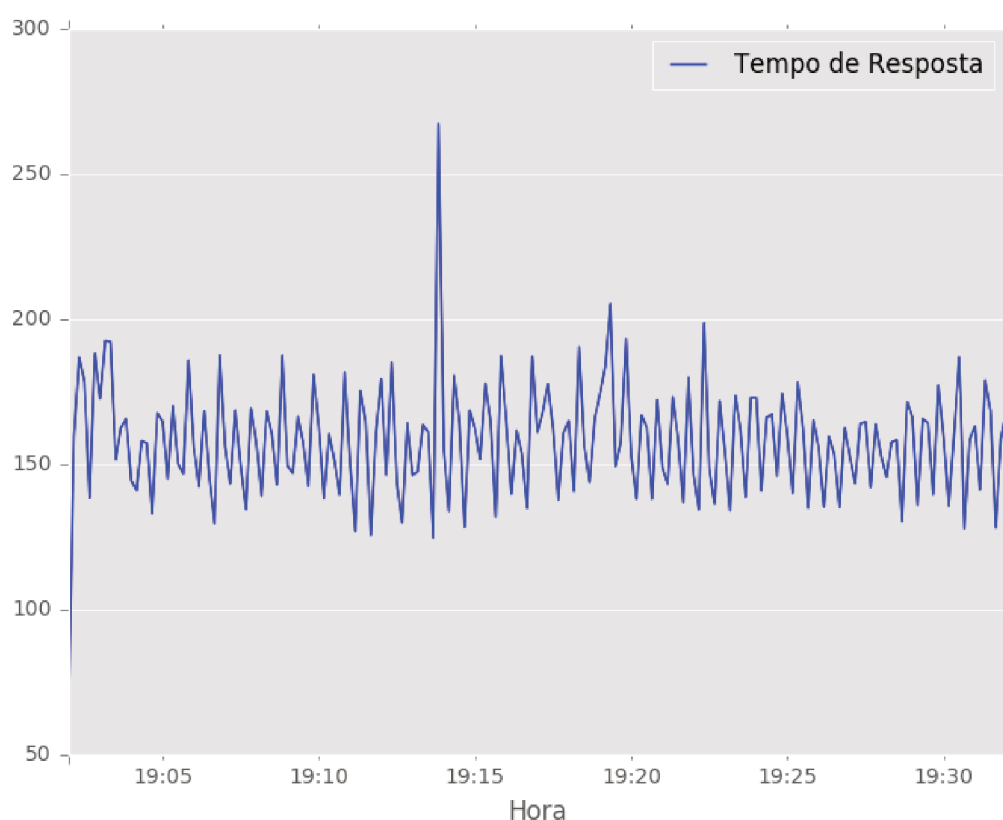


Figura 18 – Tempo de Resposta (ms) Médio a cada 10 segundos para o Teste WEB API Constante com 5 clientes

A Figura 18 descreve o tempo de resposta durante a execução no qual foram coletados os seguintes valores:

- Tempo de Resposta (média): 156,18

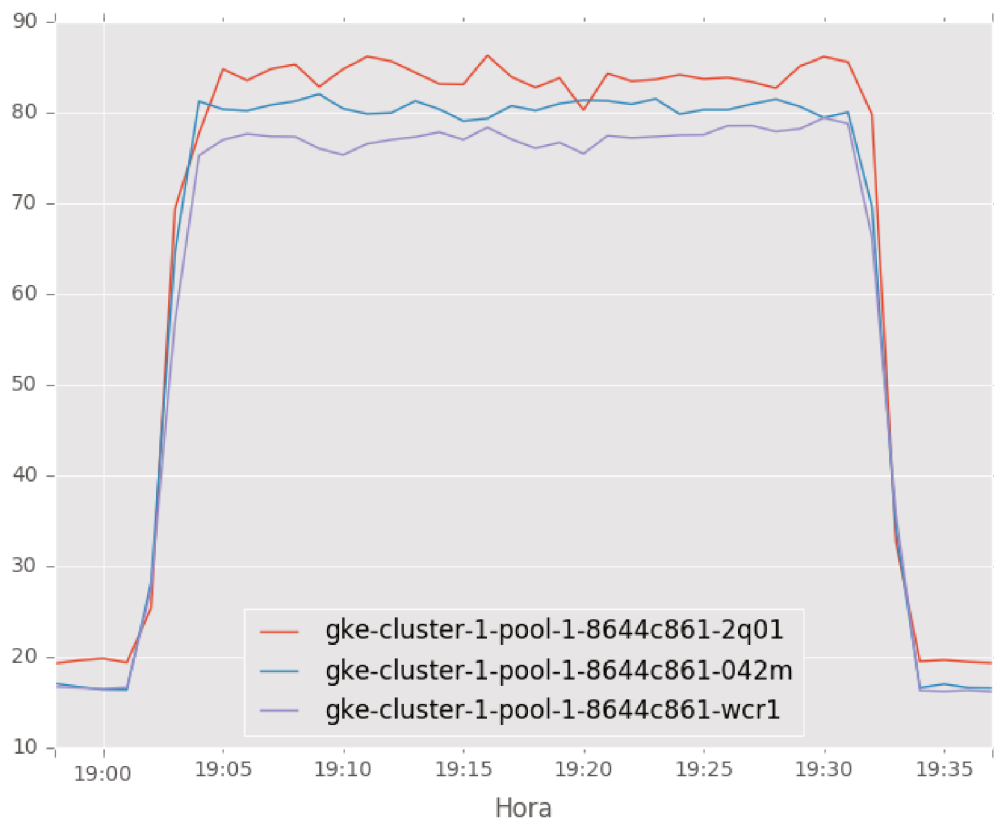


Figura 19 – Uso de CPU Médio a cada minuto para o Teste WEB API Constante com 5 clientes

- Tempo de Resposta (desvio padrão): 303,02

A Figura 19 apresenta o uso de CPU durante a execução no qual foram coletados os seguintes valores:

- CPU 1 (média): 83,28
- CPU 1 (desvio padrão): 3,22
- CPU 2 (média): 79,68
- CPU 2 (desvio padrão): 3,54
- CPU 3 (média): 76,23
- CPU 3 (desvio padrão): 4,27

4.6.2.2 Constante - Teste 2

Este segundo teste, coleta os dados iniciais da plataforma, sem estresse e sem utilizar métodos específicos que fazem carga. Este segundo teste do método constante,

também foi realizado para verificar se ocorrem diferenças nas métricas, se o número de clientes é aumentado. Para este teste, foi definido o uso de 10 clientes.

A seguir, está sumarizada os parâmetros de execução:

Duração	30 minutos
Número de Clientes	10

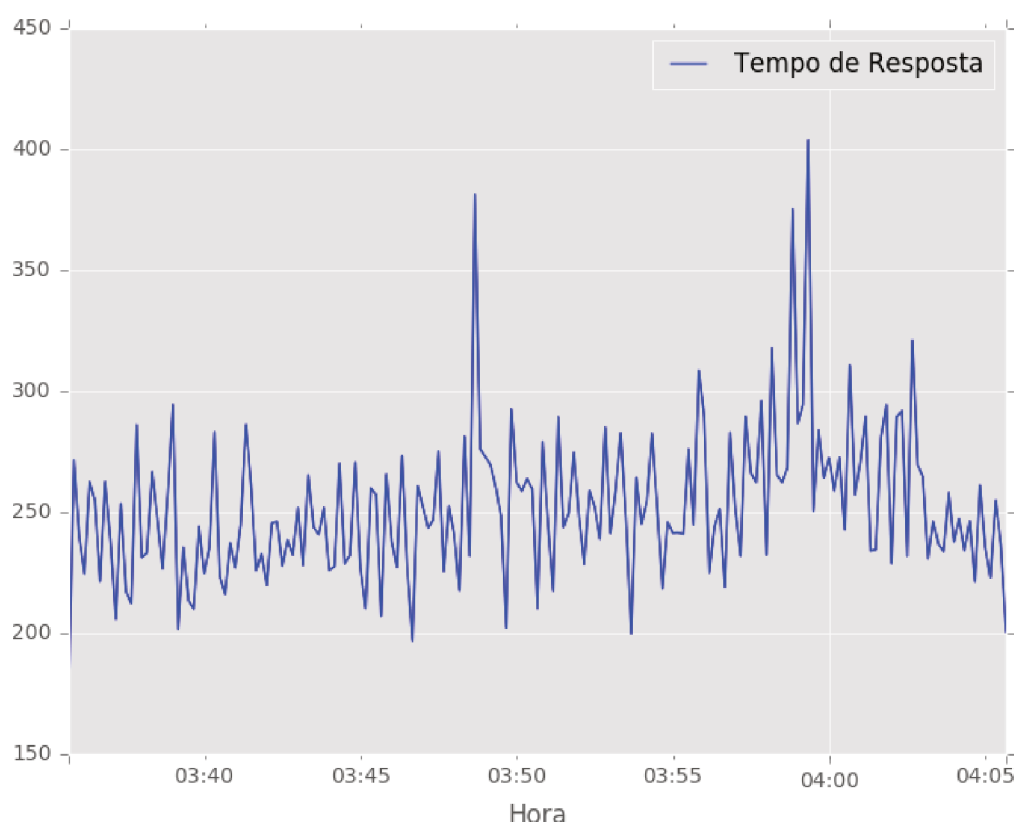


Figura 20 – Tempo de Resposta (ms) Médio a cada 10 segundos para o Teste Cassandra Constante com 10 clientes

A Figura 20 descreve o tempo de resposta durante a execução no qual foram coletados os seguintes valores:

- Tempo de Resposta (média): 0,96
- Tempo de Resposta (desvio padrão): 0,20

A Figura 21 apresenta o uso de CPU durante a execução no qual foram coletados os seguintes valores:

- CPU 1 (média): 89,60

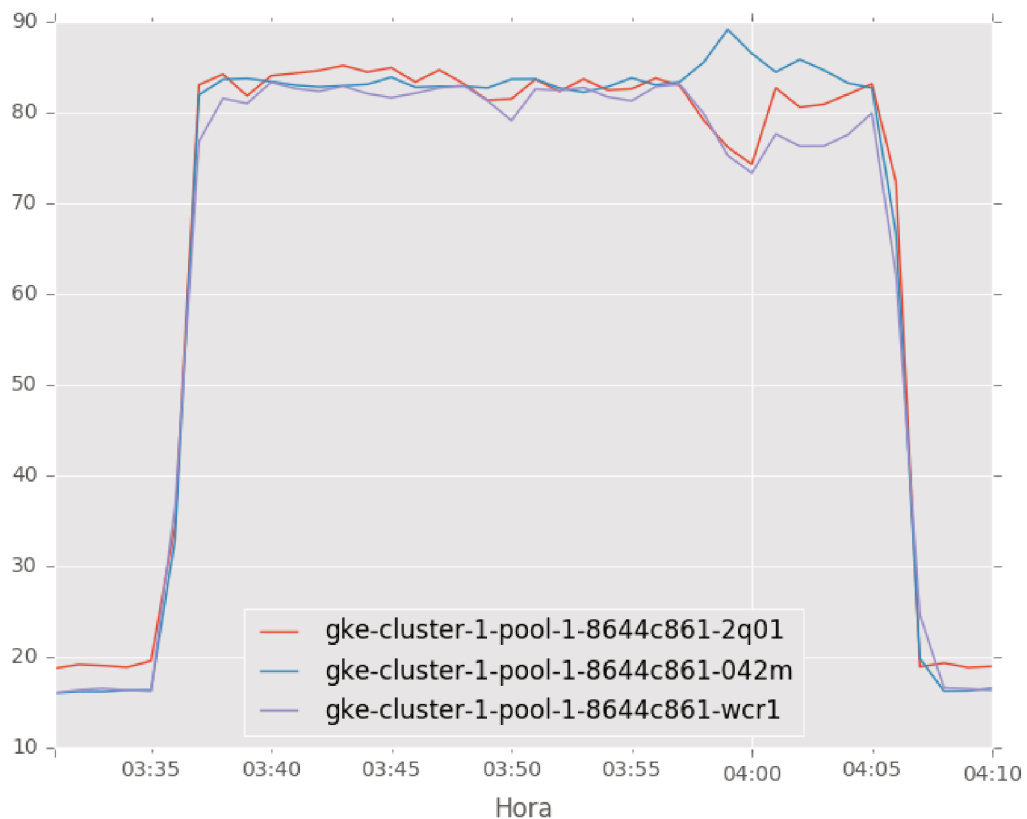


Figura 21 – Uso de CPU Médio a cada minuto para o Teste Cassandra Constante com 10 clientes

- CPU 1 (desvio padrão): 12,73
- CPU 2 (média): 52,59
- CPU 2 (desvio padrão): 6,89
- CPU 3 (média): 91,00
- CPU 3 (desvio padrão): 13,52

Em relação ao teste apresentado na Seção 4.6.2.1, foi observado um aumento no tempo de resposta, em cerca de 100ms além também de uma maior dispersão dos dados (600,53 contra 303,02). Quanto ao consumo de CPU, apenas foi observado que há uma maior dispersão dos dados neste teste em relação ao teste apresentado na Seção 4.6.2.1.

4.6.2.3 Flashcrowd - Teste 1

Este teste foi executado para avaliar o comportamento do sistema utilizando o método de carga Flashcrowd em vez da carga constante. Para o primeiro teste foi definido

o número de clientes como 10 e o valor da *Shock Level* em 3, ou seja, nos momentos de Flash, haviam 30 clientes. Na duração, diferentemente do teste Constante, este teste teve duração de 60 minutos, pois os 30 minutos não eram suficientes para as várias Ondas de Flash.

A seguir, está sumarizada os parâmetros de execução:

Duração	60 minutos
Clientes	10
Shock Level	3
Ocorrências	3
Período	15

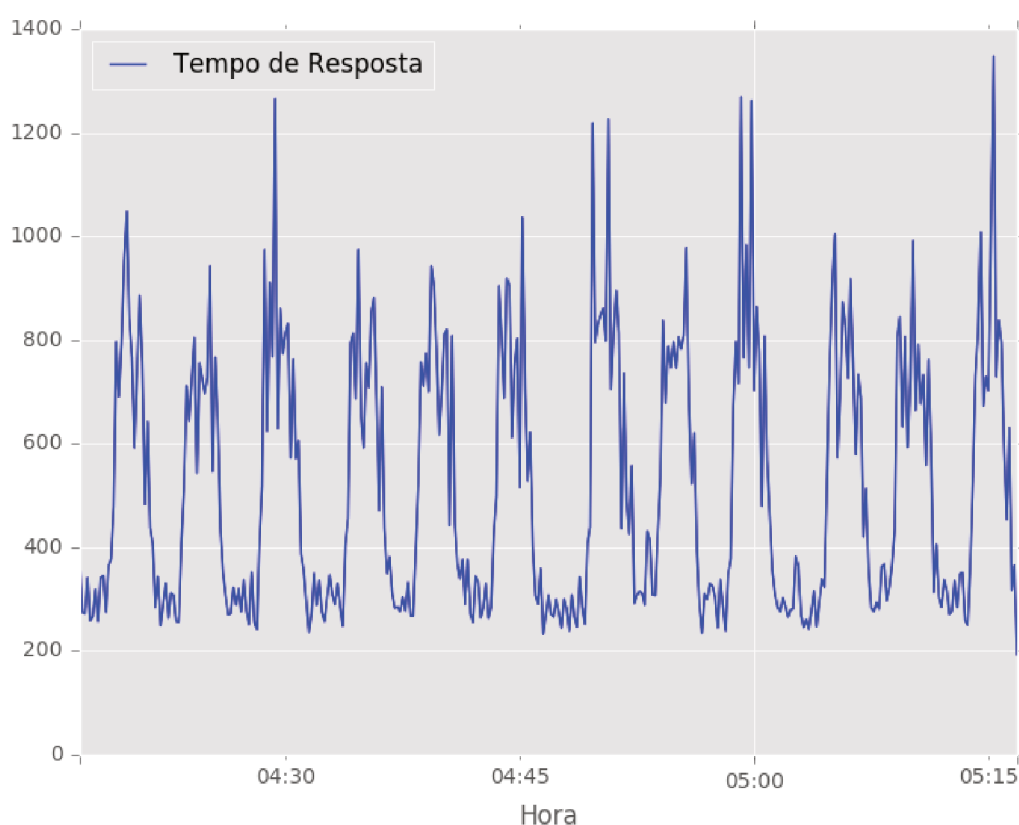


Figura 22 – Tempo de Resposta (ms) Médio a cada 10 segundos para o Teste WEB API Flashcrowd com 10 clientes e *shock_level* 3

A Figura 22 descreve o tempo de resposta durante a execução no qual foram coletados os seguintes valores:

- Tempo de Resposta (média): 422,34

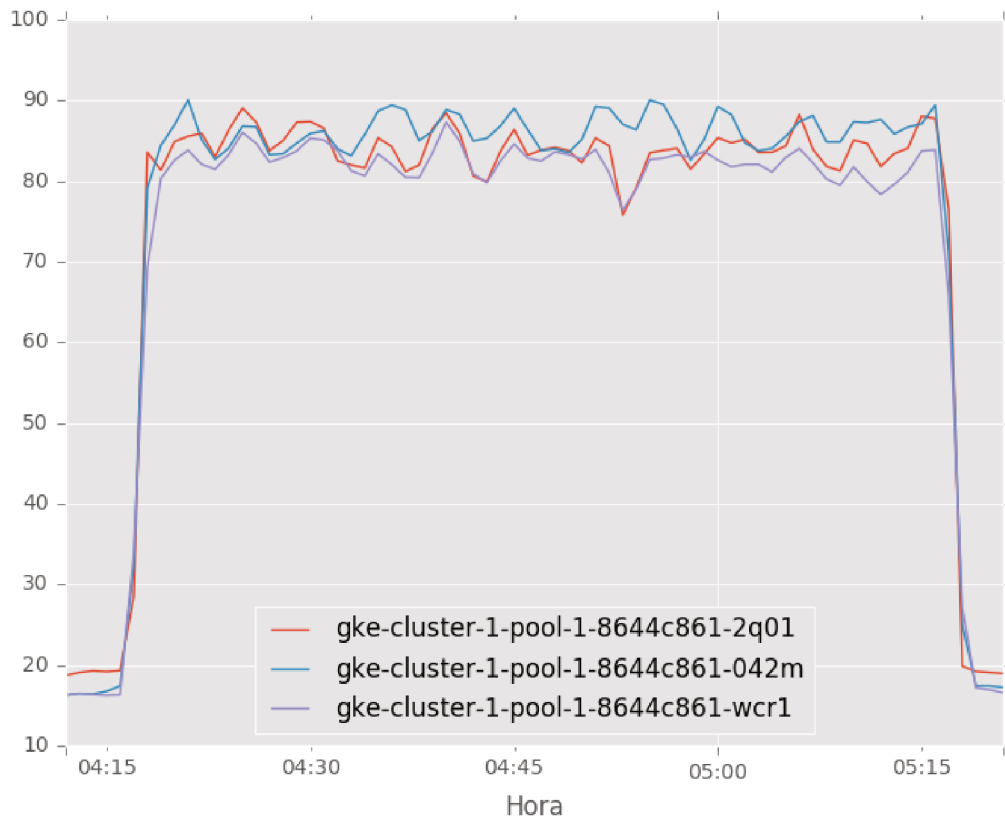


Figura 23 – Uso de CPU Médio a cada minuto para o Teste WEB API Flashcrowd com 10 clientes e *shock_level* 3

- Tempo de Resposta (desvio padrão): 1.337,46

A Figura 23 apresenta o uso de CPU durante a execução no qual foram coletados os seguintes valores:

- CPU 1 (média): 83,19
- CPU 1 (desvio padrão): 7,59
- CPU 2 (média): 85,28
- CPU 2 (desvio padrão): 7,33
- CPU 3 (média): 81,34
- CPU 3 (desvio padrão): 6,72

4.6.2.4 Flashcrowd - Teste 2

Este teste foi executado avaliar o comportamento do sistema utilizando o método de carga Flashcrowd em vez da carga constante, além de ser comparado com o teste

descrito na Seção 4.6.2.3. Para o segundo teste apenas foi modificado o valor da *Onda de Flash* para 5.

A seguir, está sumarizada os parâmetros de execução:

Duração	60 minutos
Cientes	10
Shock Level	5
Ocorrências	3
Período	15

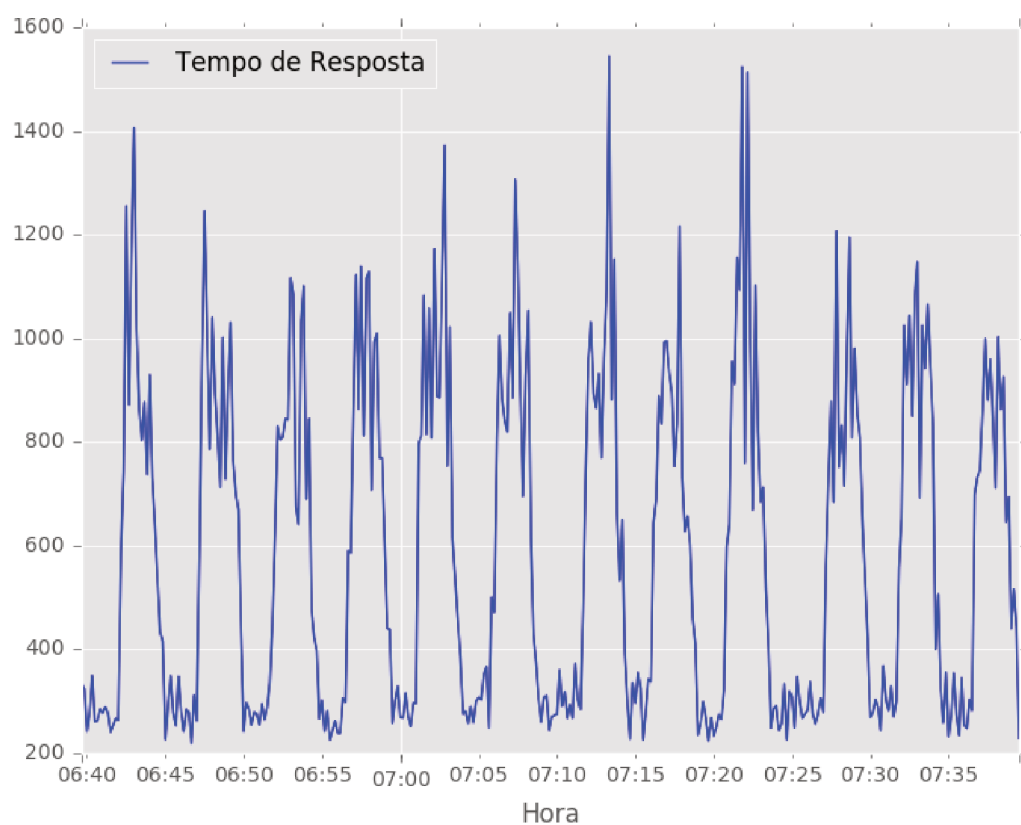


Figura 24 – Tempo de Resposta (ms) Médio a cada 10 segundos para o Teste WEB API Flashcrowd com 10 clientes e *shock_level* 5

A Figura 24 descreve o tempo de resposta durante a execução no qual foram coletados os seguintes valores:

- Tempo de Resposta (média): 430,57
- Tempo de Resposta (desvio padrão): 1.341,15

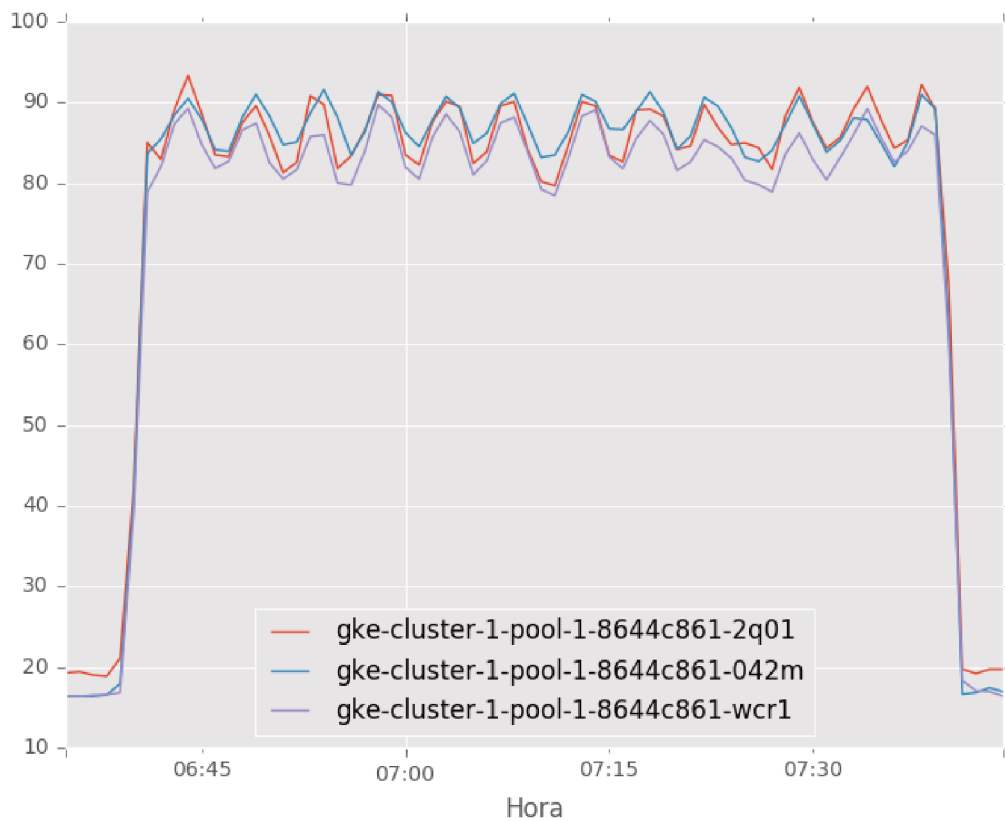


Figura 25 – Uso de CPU Médio a cada minuto para o Teste WEB API Flashcrowd com 10 clientes e *shock_level* 5

A Figura 25 apresenta o uso de CPU durante a execução no qual foram coletados os seguintes valores:

- CPU 1 (média): 85,81
- CPU 1 (desvio padrão): 6,63
- CPU 2 (média): 86,44
- CPU 2 (desvio padrão): 6,55
- CPU 3 (média): 83,34
- CPU 3 (desvio padrão): 6,55

Neste teste, comparando-o com o teste apresentado na Seção 4.6.2.3 pode-se notar pouca diferença no Tempo de Resposta, o que pode significar que o impacto deste tipo de teste não é tão significativo. O uso de CPU também sofreu pouca variação, variando entre 1% e 2%.

4.6.2.5 Sinusoid - Teste 1

Este teste foi executado para avaliar o comportamento do sistema utilizando o método de carga Sinusoid. Para o primeiro teste foi definido o número de clientes como 20 e o valor da amplitude em 10. Na duração, diferentemente do teste Constante, este teste teve duração de 60 minutos, pois os 30 minutos não eram suficientes para observar as ondas senoidais.

A seguir, está sumarizada os parâmetros de execução:

Duração	60 minutos
Cientes	20
Amplitude	10
Período	20

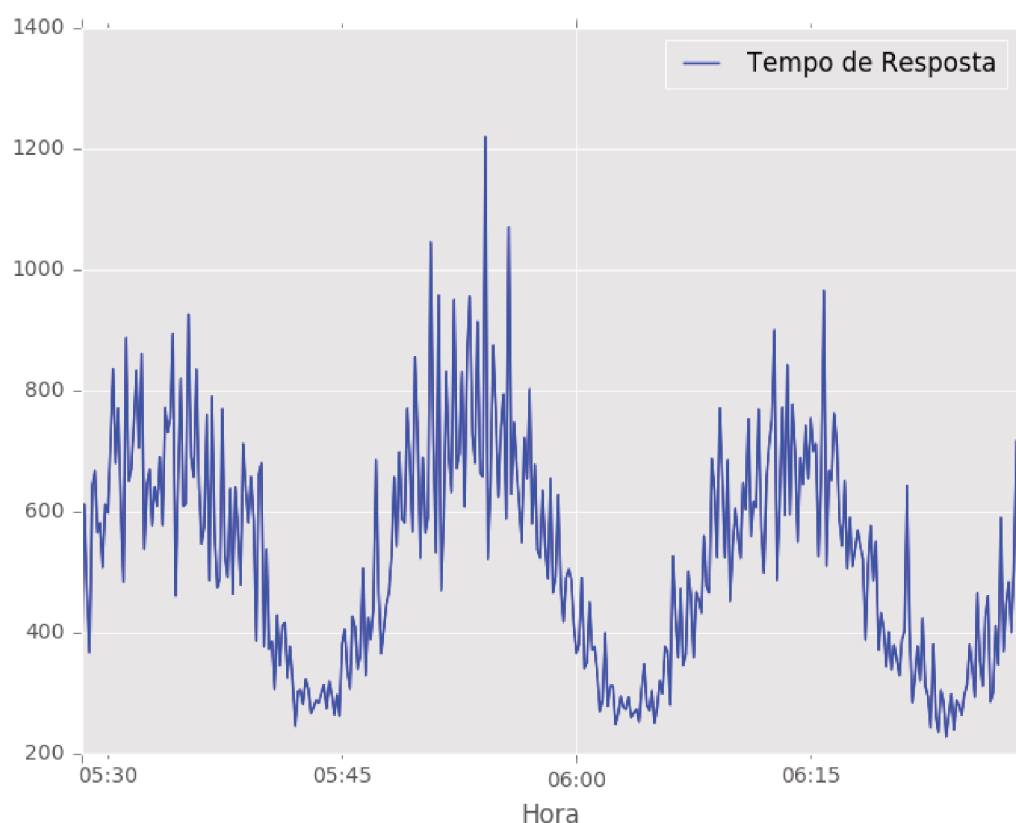


Figura 26 – Tempo de Resposta (ms) Médio a cada 10 segundos para o Teste WEB API Sinusoid com 20 clientes e amplitude 10

A Figura 26 descreve o tempo de resposta durante a execução no qual foram coletados os seguintes valores:

- Tempo de Resposta (média): 463,54

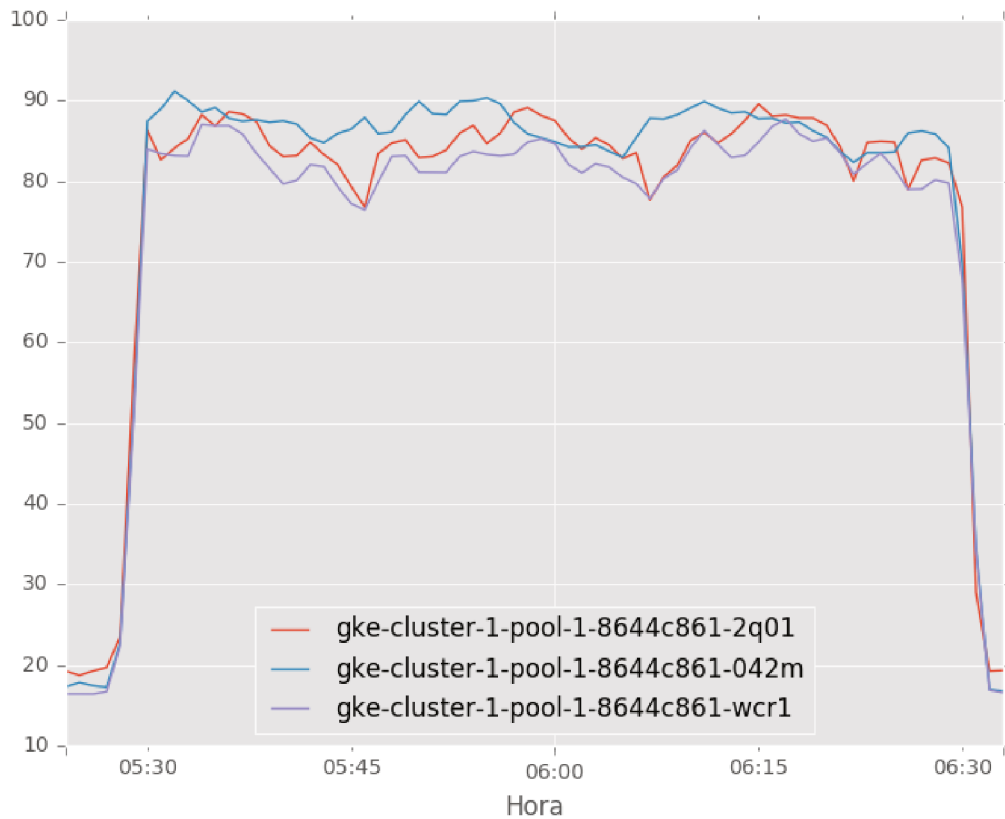


Figura 27 – Uso de CPU Médio a cada minuto para o Teste WEB API Sinusoid com 20 clientes e amplitude 10

- Tempo de Resposta (desvio padrão): 1.339,85

A Figura 27 apresenta o uso de CPU durante a execução no qual foram coletados os seguintes valores:

- CPU 1 (média): 84,29
- CPU 1 (desvio padrão): 4,57
- CPU 2 (média): 86,36
- CPU 2 (desvio padrão): 5,03
- CPU 3 (média): 82,10
- CPU 3 (desvio padrão): 4,84

4.6.2.6 Sinusoid - Teste 2

Este teste foi executado para avaliar o comportamento do sistema utilizando o método de carga Sinusoid e também para comparar com o teste descrito na Seção 4.6.2.5.

Para o segundo teste foi definido o número de clientes foi aumentado para 30 e o valor da amplitude para 20.

A seguir, está sumarizada os parâmetros de execução:

Duração	60 minutos
Cientes	30
Amplitude	20
Período	20

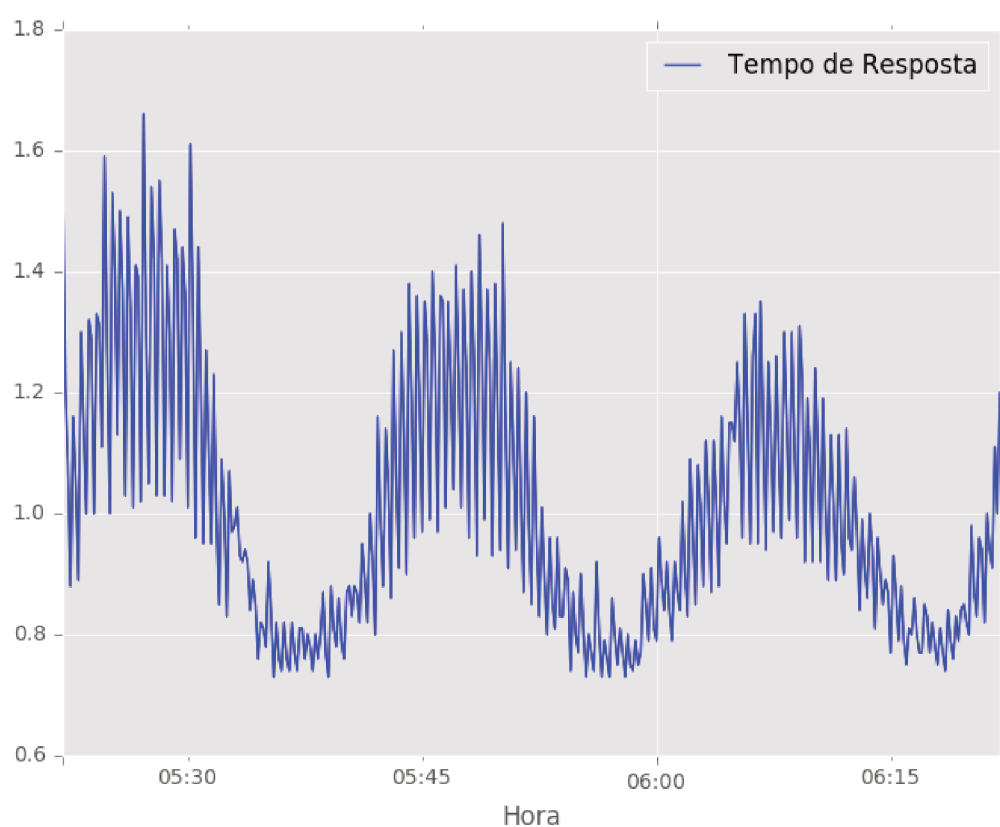


Figura 28 – Tempo de Resposta (ms) Médio a cada 10 segundos para o Teste WEB API Sinusoid com 30 clientes e amplitude 20

A Figura 28 descreve o tempo de resposta durante a execução no qual foram coletados os seguintes valores:

- Tempo de Resposta (média): 542,53
- Tempo de Resposta (desvio padrão): 1.545,38

A Figura 29 apresenta o uso de CPU durante a execução no qual foram coletados os seguintes valores:

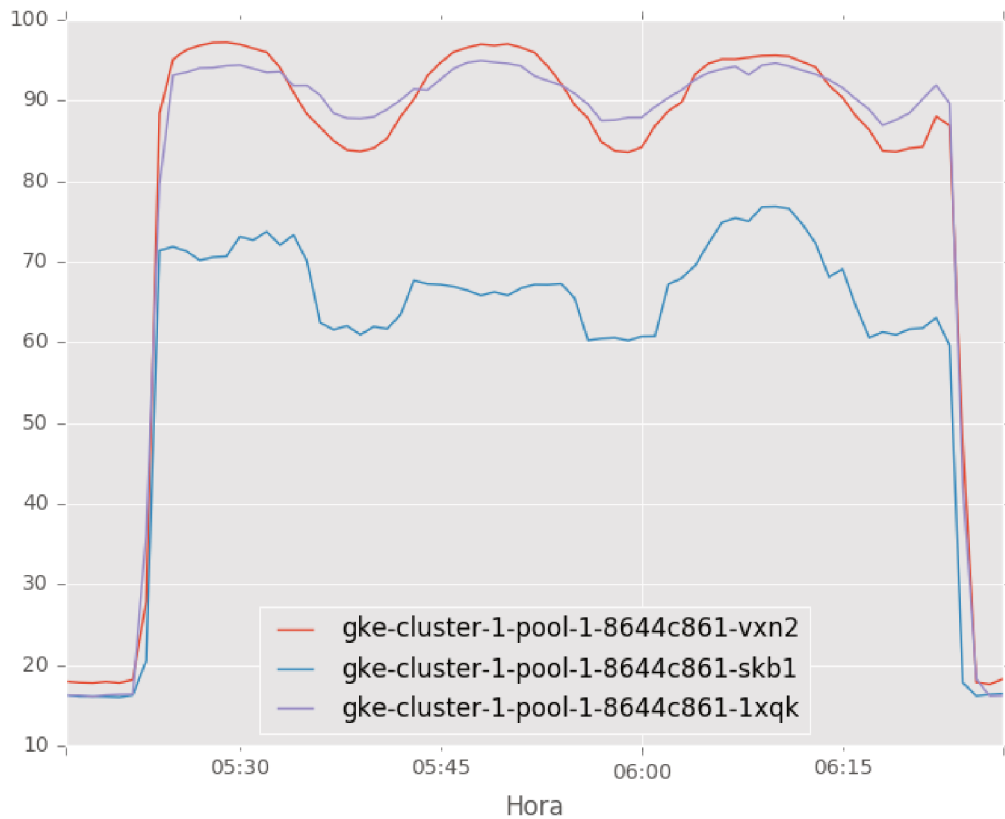


Figura 29 – Uso de CPU Médio a cada minuto para o Teste WEB API Sinusoid com 30 clientes e amplitude 20

- CPU 1 (média): 87,24
- CPU 1 (desvio padrão): 4,16
- CPU 2 (média): 87,60
- CPU 2 (desvio padrão): 4,80
- CPU 3 (média): 83,70
- CPU 3 (desvio padrão): 4,93

Foi observado que em relação ao teste apresentado na Seção 4.6.2.5 houve um aumento no tempo de resposta em aproximadamente 80ms (542,53 contra 463,54), e também um aumento na dispersão dos dados (1.545,38 contra 1.339,85). Já no consumo de CPU, em todas as métricas, mas o mais expressivo foi na CPU 3 com um aumento de cerca de 3%.

4.6.2.7 Avaliação dos Resultados

Foi observado que entre os Testes Constante descrito nas Seções 4.6.2.1 e 4.6.2.2, há uma maior diferença e dispersão no Tempo de Resposta se comparado aos Testes Sinusoid apresentados nas Seções 4.6.2.5 e 4.6.2.6 e Flashcrowd apresentados nas Seções 4.6.2.3 e 4.6.2.4. Quanto ao consumo de CPU, também não nota-se diferença se os testes forem comparados entre si, porém se compararmos o Teste Sinusoid detalhado nas Seções 4.6.2.6 e 4.6.2.1 podemos ver uma maior diferença no uso de CPU, porém há um indicativo que essa diferença ocorre não por causa do método em si, mas por causa de um maior número de clientes no primeiro em relação ao segundo. No fim, pode-se notar, que modificando os parâmetros, é possível modificar as métricas, porém não é possível concluir se os métodos podem influenciar nelas.

4.6.2.8 Tempo de Resposta

Teste	Média	Desvio Padrão
Constante 1	156,18	303,02
Constante 2	249,38	600,53
Flashcrowd 1	422,34	1.337,46
Flashcrowd 2	430,57	1.341,15
Sinusoid 1	463,54	1.339,85
Sinusoid 2	542,53	1.545,38

4.6.2.9 Uso de CPU

Teste	M* (1)***	DP** (1)	M (2)	DP (2)	M (3)	DP (3)
Constante 1	83,28	3,22	79,68	3,54	76,23	4,27
Constante 2	80,86	9,08	82,00	9,39	79,07	8,48
Flashcrowd 1	83,19	7,59	85,28	7,33	81,34	6,72
Flashcrowd 2	85,81	6,63	86,44	6,55	83,34	6,55
Sinusoid 1	84,29	4,57	86,36	5,03	82,10	4,84
Sinusoid 2	87,24	4,16	87,60	4,80	83,70	4,93

*M se refere a média.

**DP Se refere ao Desvio Padrão.

***Os valores em parênteses representam as CPUs.

5 Conclusão

O trabalho se propôs a construir um arcabouço para a realização de testes em ambientes em Cluster e testá-lo para verificar a efetividade do mesmo. O propósito deste arcabouço era não só executar testes, mas ser extensível e permitir a utilização de diferentes ambientes e diferentes métodos de carga. Após a criação deste arcabouço, diversos testes foram executados para verificar a hipótese inicial e saber se era possível degradar um ambiente utilizando-o. Diferentes métodos e aplicações foram usadas para tentar comprovar o funcionamento do Arcabouço.

A partir dos testes analisados, foi possível observar que ao modificar os parâmetros, em alguns deles, observa-se também uma diferença nas métricas coletadas, tanto no serviço do Cassandra, quanto no serviço do Web API, esta mudança ocorreu em alguns dos testes como evidenciado no Capítulo 4. Contudo, o trabalho deixa algumas incertezas, que poderiam ser sanados com a realização de mais testes e a modificação de parâmetros, porém, pode haver um grande número de combinações e o foco não era testar todas elas. Enfim, dado tudo que foi observado, é possível concluir que os ambientes podem ser degradados, utilizando o arcabouço construído e calibrando-se os parâmetros dependendo da aplicação.

5.1 Trabalhos Futuros

A proposta do trabalho era construir um Arcabouço que pudesse ser extensível, e inclusive o Apache JMeter, uma ferramenta de testes já consolidada, pode ser usada. Porém, ao longo dos experimentos e da escrita deste documento, percebeu-se que outras práticas podem ser utilizadas para aprimorar a ferramenta e avaliação dos resultados. As próximas seções definem algumas melhorias que podem ser implementadas em trabalhos futuros.

5.1.1 Padronização da coleta de métricas e geração de resultados

Durante os experimentos, houve dificuldade em coletar e gerar resultados a partir das métricas, portanto, uma evolução da aplicação poderia ser a coleta automática de métricas e subentendo-as para algum servidor de arquivos (Dropbox, Google Drive) e ainda fazendo a geração de gráficos.

5.1.2 Criação de uma Interface Web para gestão dos cenários

Nesta evolução, a proposta é criar uma Interface Web na qual possa-se criar planos de execução de diversos testes, realizar a execução e verificar os resultados.

5.1.3 Criação automática de infraestrutura para testes

Nos testes realizados, foi utilizado uma Infraestrutura criada no Google Cloud, porém esta infraestrutura teve que ser criada para que fosse possível a execução. Esta melhoria poderia fazer o uso de ferramentas como o [Terraform](#) e quando os testes fosse executados, o arcabouço criaria a infraestrutura necessária e no fim dos testes, destruiria a infraestrutura.

5.1.4 Explorar outras métricas ou métodos

Neste trabalho, apenas o Tempo de Resposta e uso de CPU foram analisados, porém, outras métricas poderiam ser exploradas. Outros métodos de carga poderiam ser implementados, ou ainda novos experimentos poderiam ser realizados com a calibração de novos parâmetros.

Referências

- Ari, I. et al. Managing flash crowds on the internet. In: . [s.n.], 2003. p. 246–249. Disponível em: <<https://doi.org/10.1109/MASCOT.2003.1240667>>. Citado 2 vezes nas páginas 16 e 18.
- COSTA, B. et al. Desenvolvimento de software educacional para representação e reconhecimento de som aplicado à ausculta cardiovascular. *Revista Eletrônica TECCEN*, v. 2, p. 17, 10 2016. Disponível em: <<https://doi.org/10.21727/teccen.v2i1.226>>. Citado na página 17.
- DOCKER. *Swarm Mode Overview*. 2020. Disponível em: <<https://docs.docker.com/engine/swarm/>>. Citado na página 14.
- DOCKER. *What is a container?* 2020. Disponível em: <<https://www.docker.com/resources/what-container>>. Citado na página 13.
- HOLLA, S. *Orchestrating Docker*. [S.l.]: Packt Publishing, 2015. Citado na página 13.
- JMETER, A. *Elements of a Test Plan*. 2021. Disponível em: <https://jmeter.apache.org/usermanual/test_plan.html#thread_group>. Acesso em: 12 jan. 2021. Citado na página 16.
- KHAN, A. Key characteristics of a container orchestration platform to enable a modern application. *IEEE Cloud Computing*, v. 4, 9 2017. Disponível em: <<https://doi.org/10.1109/MCC.2017.4250933>>. Citado 2 vezes nas páginas 10 e 11.
- KUBERNETES. *Conceitos do Kubernetes*. 2020. Disponível em: <<https://kubernetes.io/pt/docs/concepts/>>. Citado na página 15.
- Matthias, K.; Kane, S. P. *Docker Up & Running*. [S.l.]: O'Reilly, 2015. Citado 3 vezes nas páginas 11, 13 e 14.
- MOLYNEAUX, I. *The Art of Application Performance Testing: From Strategy to Tools*. O'Reilly Media, 2014. (Theory in practice). ISBN 9781491900529. Disponível em: <<https://books.google.com.br/books?id=187UBQAAQBAJ>>. Citado na página 16.
- PORTWORX. *2019 Container Adoption Survey*. [S.l.], 2019. Disponível em: <<https://portworx.com/wp-content/uploads/2019/05/2019-container-adoption-survey.pdf>>. Citado na página 13.
- SANTOS, T. *Fundamentos da computação em nuvem*. [S.l.]: Senac, 2018. Citado 2 vezes nas páginas 10 e 11.
- TAURION, C. *Cloud computing-computação em nuvem*. [S.l.]: Brasport, 2009. Citado na página 10.
- TURNBULL, J. *The Docker Book: Containerization is the new virtualization*. [S.l.]: James Turnbull, 2014. Citado na página 11.
- VERAS, M. *Cloud Computing: nova arquitetura da TI*. [S.l.]: Brasport, 2012. Citado na página 10.