Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Um Mecanismo de Auto Elasticidade com base no Tempo de Resposta para Ambientes de Computação em Nuvem baseados em Containers

Marcelo Cerqueira de Abranches

Dissertação apresentada como requisito parcial para conclusão do
Mestrado Profissional em Computação Aplicada

Orientador
Prof.a Dr.a Priscila América Solis

Brasília
2016

Ficha catalográfica elaborada automaticamente,
com os dados fornecidos pelo(a) autor(a)

Cm Cerqueira de Abranches, Marcelo
Um Mecanismo de Auto Elasticidade com base no Tempo de Resposta para Ambientes de Computação em Nuvem baseados em Containers / Marcelo Cerqueira de Abranches; orientador Priscila América Solis. -- Brasília, 2016.
120 p.

Dissertação (Mestrado - Mestrado Profissional em Computação Aplicada) -- Universidade de Brasília, 2016.

1. Computação em Nuvem. 2. Balanceamento de Carga. 3. Dimensionamento de Sistemas Computacionais. 4. Controladores PID. 5. Sistemas Web. I. América Solis, Priscila, orient. II. Título.

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Um Mecanismo de Auto Elasticidade com base no Tempo de Resposta para Ambientes de Computação em Nuvem baseados em Containers

Marcelo Cerqueira de Abranches

Dissertação apresentada como requisito parcial para conclusão do
Mestrado Profissional em Computação Aplicada

Prof.a Dr.a Priscila América Solis (Orientador)
CIC/UnB

Prof. Dr. Eduardo Alchieri
CIC/UnB

Prof. Dr. Paulo Gondim
ENE/UnB

Prof. Dr. Marcelo Ladeira
Coordenador do Programa de Pós-graduação em Computação Aplicada

Brasília, 28 de Novembro de 2016

Dedicatória

Dedico este trabalho a minha família, em especial a Paula que é minha amiga, companheira e meu amor.

Agradecimentos

Agradeço a minha família e amigos pelo apoio incondicional e carinho, a todos os meus professores que iluminaram meu caminho e principalmente ao Senhor, que é a razão de tudo.

Resumo

Este trabalho propõe uma arquitetura de computação em nuvem baseada em *containers* e um algoritmo de auto elasticidade. O algoritmo promove a otimização do processamento das requisições por meio da alocação eficiente do número de *containers* para alcançar tempos esperados de resposta no processamento de requisições. Para avaliar arquitetura proposta foi utilizada uma carga de trabalho caracterizada com base em uma série temporal real resultante das requisições de um sistema Web da Controladoria Geral da União (CGU). O sistema foi submetido a diferentes testes de carga e os resultados mostram que a solução proposta apresenta um forte potencial dado que consegue cumprir os requerimentos de desempenho alocando *containers* de forma mais eficiênte do que outras propostas.

Palavras-chave: Computação em Nuvem, balanceamento de carga, dimensionamento, elasticidade, alta disponibilidade, escalabilidade, desempenho, Controlares PID, Sistemas Web, Controladoria-Geral da União.

Abstract

This work proposes a cloud computing architecture based on containers and an auto elasticity algorithm. The algorithm promotes the optimization of request processing by efficiently allocating the number of containers to achieve expected response times in web requests. To evaluate the proposed architecture, a workload was used based on a real time series resulting from the requests of a Web System of Controladoria Geral da União. The system was tested with different loads and the results show that the proposed solution has a strong potential since it can meet the performance requirements by allocating containers more efficiently than other related proposals.

Keywords: Cloud Computing, Load Balancing, Sizing, Elasticity, High Availability, Performance, PID Controllers, Web Systems, Controladoria-Geral da União.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Neste capitulo será descrito o tema e a sua justificativa, assim como os objetivos e os benefícios a serem alcançados com a proposta desenvolvida neste trabalho de dissertação de mestrado.

1.1 Definição do Problema

Nos últimos anos, o aumento massivo da geração de conteúdo e a explosão no uso de dispositivos móveis alterou o perfil tradicional dos sistemas baseados na *web*. Assim, também no Brasil, a lei de acesso a informação [11] tem promovido uma maior e crescente demanda aos sistemas do governo, o que se traduz em maiores desafios em termos de infraestrutura de Tecnologia da Informação (TI) para garantir disponibilidade, segurança, qualidade de serviço, entre outros.

Os custos crescentes associados à manutenção de infraestrutura própria de TI, assim como a complexidade do sistema derivada do aumento das demandas de serviço, tem promovido que dentro do governo brasileiro tenham sido publicadas no mês de maio de 2016 [8] uma série de recomendações. Entre estas recomendações está a adoção de sistemas de computação em nuvem, em vez de soluções de infraestrutura de TI individuais para cada órgão.

Nesse sentido, nos próximos meses, vários órgãos deverão migrar aplicações de alta demanda para nuvens públicas, privadas ou híbridas. Essas recomendações podem ser consideradas coerentes com o observado em outros países do mundo e representam hoje uma forte tendência tecnológica.

Um caso particular de estudo neste trabalho foi o serviço oferecido pelo Portal da Transparência [6] da CGU (Controladoria-Geral da União). Considerando a atual tendência de migrar os serviços para ambientes de computação em nuvem, foi definido como caso de estudo o problema de propor uma arquitetura, que poderia ser implementada ou

adotada neste órgão (ou em outros similares) e que teria como principal objetivo atender a demandas variáveis de usuários e que ao mesmo tempo integrasse ideias e tecnologias do estado da arte. Dessa forma, a proposta de tal arquitetura teria como objetivo, além de prover o serviço ou conjunto de serviços, propiciar uma evolução técnica sólida, robusta e de baixo custo para as futuras demandas.

Com base no anterior, os objetivos deste trabalho são:

- Definir uma arquitetura de computação em nuvem que permita um uso massivo de sistemas *web*, garanta um tempo mínimo de resposta e que integre características de auto elasticidade, auto escalabilidade e disponibilidade;
- Fazer um levantamento do estado da arte de métodos e tecnologias que possam ser integradas em uma solução de computação em nuvem;
- Propor novas ideias que venham dar um caráter inovador e evolutivo à solução proposta;
- Com base em uma abordagem sistemática, avaliar o desempenho da solução proposta e indicar caminhos futuros.

1.2 Justificativa do Tema

A pesquisa nos ambientes de computação em nuvem apresenta como um dos maiores pontos de interesse algoritmos de auto escalabilidade [33] e balanceamento de carga. Como em qualquer processo de dimensionamento e planejamento, um dos pontos cruciais para esta tarefa é a correta caracterização da carga de trabalho e a otimização no uso de recursos, os quais devem atender um conjunto de requerimentos de desempenho das aplicações.

Diversos trabalhos recentes abordam o dimensionamento elástico e a auto escalabilidade em ambientes de computação em nuvem [33], [26], [15]. As soluções propostas nestes trabalhos são na sua maioria baseadas em métodos matemáticos tradicionais, tais como teoria de filas, análises de séries temporais, heurísticas, entre outros.

Este trabalho propõe um algoritmo para promover a auto elasticidade em um ambiente de nuvem baseado na alocação eficiente do número de *containers* para atender requisições de um ambiente *web*. A proposta foi avaliada em um ambiente simulado e os resultados mostram um potencial aceitável de otimização para o processo de alocação de recursos de processamento nesse ambiente.

Outros temas de interesse na pesquisa de computação em nuvem são gerenciamento de tráfego, monitoramento e análise. A arquitetura inerentemente distribuída de sistemas de computação em nuvem faz com que métodos de agregação de logs e sua análise, de forma a extrair valor destas informações, se tornem relevantes [28]. Este trabalho propõe

uma infraestrutura para agregação de *logs* de acessos de um sistema *web* e extração de dados, que permitam alimentar o sistema para melhorar a experiência do usuário com base nessas informações.

Outro tema relacionado a computação em nuvem é a eficiência energética de *datacenters* [28]. O correto dimensionamento de sistemas é importante na medida em que evita-se o uso de recursos físicos de forma desnecessária. Este trabalho propõe um método que tem o objetivo de alocar recursos de forma eficiente e de acordo com a demanda.

1.3 Contribuição Esperada

A contribuição esperada com este trabalho é a proposição de uma arquitetura para computação em nuvem que seja escalável e promova alto desempenho aos sistemas nela hospedados. Esta arquitetura irá disponibilizar um ambiente elástico para a hospedagem de sistemas *web*, que serão dimensionados usando metodologia inovadora.

O ambiente deve ser escalável e o balanceamento de carga será utilizado para otimizar os recursos disponíveis com base nos requisitos de desempenho. Desta forma, espera-se a disponibilização de um ambiente escalável e de alto desempenho para hospedagem dos sistemas *web* da CGU ou de algum outro órgão federal com caraterísticas de uso semelhantes.

1.4 Estrutura do Trabalho

Os próximos capítulos serão estruturados conforme descrito a seguir:

- O capítulo 2 contém a revisão bibliográfica, onde serão apresentados os trabalhos relacionados, a descrição das tecnologias e a base teórica da solução.

- O capítulo 3 contém a solução proposta, onde serão apresentadas as principais características das ferramentas utilizadas, a arquitetura da solução e o detalhamento do algoritmo de auto elasticidade desenvolvido.

- O capítulo 4 descreve a metodologia de avaliação de desempenho, apresenta os resultados experimentais e a sua análise.

- O capítulo 5 apresenta as conclusões e trabalhos futuros desta pesquisa.

Capítulo 2

Revisão Bibliográfica

Neste capitulo serão apresentados os conceitos teóricos resultantes da revisão da literatura associada ao tema, assim como o paradigma e as tecnologias que foram definidas para configurar uma arquitetura de nuvem privada com auto elasticidade.

2.1 Conceitos de Computação em Nuvem

Segundo o NIST [29], computação em nuvem é um modelo que permite acesso "onipresente", de forma conveniente e sob-demanda a serviços de rede, por meio de um conjunto de recursos computacionais (rede, servidores, armazenamento, aplicações e serviços), que podem ser provisionados rapidamente e "liberados" com mínimo esforço de gerenciamento por parte do provedor do serviço.

Ainda segundo o NIST, o modelo de nuvem é composto por cinco características essenciais, três modelos de serviço e quatro modelos de implantação.

Nesta linha, as características essenciais que um ambiente de nuvem deve prover são:

- Serviço *self-service* sob demanda: Um consumidor pode provisionar de maneira "unilateral" recursos computacionais como servidores ou armazenamento, a medida que o recurso for necessário, e de maneira automática, sem a necessidade de interação humana.
- Amplo acesso a rede: Os recursos são disponibilizados em rede e acessados por meio de mecanismos padrão que promovem acessos a dispositivos heterogêneos como, por exemplo, celulares, *tablets* ou estações de trabalho.
- Compartilhamento de recursos: Os recursos computacionais do provedor são compartilhados entre os consumidores, com diferentes recursos físicos ou virtuais disponibilizados para um ou outro cliente de acordo com a demanda. Existe um senso

de independência de localização do recurso, de modo que o cliente não tem controle ou conhecimento do local exato onde o recurso está localizado. Porém, pode ser possível ao cliente definir a localização do serviço em um maior nível, como, por exemplo, país, estado ou *datacenter*. Exemplos destes recursos compartilhados incluem armazenamento, processamento, memória e rede.

- Rápida elasticidade: Os recursos podem ser provisionados e desprovisionados de forma elástica, em alguns casos de forma automática, de acordo com a demanda. Os recursos disponíveis aparecem para o consumidor como se fossem ilimitados e como se pudessem ser provisionados em qualquer quantidade a qualquer hora.

- Serviço contabilizado: Serviços de nuvem controlam e otimizam os recursos de forma automatizada por meio da capacidade de medição do uso dos recursos. Essa medição é feita em diferentes níveis de abstração para se adequarem a cada caso, como, por exemplo, armazenamento, processamento, banda e usuários ativos. O uso dos recursos pode ser monitorado, controlado e reportado, promovendo transparência tanto para o provedor quanto para o consumidor dos recursos.

Os modelos de provisão de serviços de computação em nuvem são separados de acordo com o nível de abstração computacional que é provida ao usuário, sendo eles:

- *Software as a Service* (SaaS): O recurso provido ao consumidor é uma aplicação sendo executada sobre uma infraestrutura de nuvem. A aplicação é acessada por vários dispositivos clientes, que podem ter interfaces leves, como, por exemplo, uma interface *web* ou uma interface cliente instalada. O cliente não gerencia nem controla a infraestrutura de nuvem que está por baixo, incluindo rede, servidores, sistemas operacionais e armazenamento. O usuário não gerencia nem mesmo os recursos da aplicação, porém, sendo possível, em alguns casos, que o usuário gerencie algumas configurações da aplicação.

- *Plataform as a Service* (PaaS): Esse modelo permite que o consumidor disponibilize uma aplicação em uma infraestrutura de nuvem. Esta aplicação pode ter sido desenvolvida pelo próprio cliente ou por terceiros. O desenvolvimento da aplicação é feito usando as linguagens de programação, bibliotecas, serviços e ferramentas suportadas e disponibilizadas pelo provedor. O consumidor não gerencia ou controla a infraestrutura de rede que está por baixo de sua aplicação, incluindo rede, servidores, sistemas operacionais ou armazenamento. Porém, o consumidor controla as suas aplicações desenvolvidas e pode ter controle sobre as configurações do ambiente da aplicação.

- *Infrastructure as a Service* (IaaS): Neste modelo é possível que o cliente provisione processamento, armazenamento, rede e outros recursos computacionais fundamentais. É possível que o consumidor decida qual sistema operacional será utilizado e quais aplicações serão instaladas. O consumidor não tem controle sobre a infraestrutura da nuvem, porém controla o sistema operacional, armazenamento e algumas configurações de rede como, por exemplo, *firewall*.

Os modelos de implantação são separados de acordo com o público para o qual o serviço estará disponível, podendo ser:

- Nuvem privada: A infraestrutura da nuvem é provisionada para uso exclusivo de uma única organização, contemplando suas diversas áreas de negócios. O próprio consumidor pode ser o proprietário dos ativos computacionais e pode também ser o operador da nuvem. Esta infraestrutura pode também ser de terceiros, e uma combinação entre os dois casos também é possível.

- Nuvem comunitária: A infraestrutura da nuvem é provisionada para uso exclusivo de uma comunidade específica de consumidores de organizações que tem interesses compartilhados (ex: Governo, organizações de segurança). Uma ou mais organizações da comunidade podem ser a proprietária e a gerente do ambiente. Assim como no caso anterior, esta infraestrutura pode também ser de terceiros, e uma combinação entre os dois casos também é possível.

- Nuvem pública: A infraestrutura da nuvem é provisionada para uso do público em geral. O proprietário e o gerente deste ambiente pode ser uma empresa, governo, universidade ou uma combinação desses.

- Nuvem híbrida: A infraestrutura da nuvem é uma composição de duas ou mais infraestruturas de nuvem distintas (privada, comunitária ou pública) que permanecem como entidades únicas, mas são integradas por tecnologias padrão ou proprietárias que permitem portabilidade entre os dados e as aplicações (ex: balanceamento de carga entre nuvens).

O uso de provedores de nuvem pública é interessante, pois o cliente não tem que se preocupar com a infraestrutura física do *datacenter*. Porém, preocupações com a segurança dos dados podem fazer com que corporações hesitem em prover serviços ou armazenar seus dados em nuvens públicas. Mesmo assim, essas corporações podem se beneficiar das funcionalidades da computação em nuvem, por meio da implantação de nuvens privadas.

2.2 Estado da Arte da Computação em Nuvem

A computação em nuvem apresenta diversos tópicos de interesse para pesquisa, com diversos problemas a serem resolvidos. Nesta seção serão apresentados alguns destes tópicos e será indicado como alguns desses tópicos estão sendo tratados neste trabalho.

2.2.1 Provisionamento automático de recursos

Provedores de computação em nuvem devem ser capazes de prover meios para alocação/desalocação de recursos conforme a demanda, de forma a atingir os níveis de serviço estabelecidos em contrato ou *Service Level Agreement (SLA)* [51]. Porém, não é óbvio como um provedor pode mapear os requisitos de *SLA*, como ,por exemplo, requisitos de qualidade de serviço (*QoS*), em requisitos de baixo nível, como, por exemplo, uso máximo de memória e processamento. Além disso, para que os serviços providos possam atender a rápidas flutuações de demanda, as decisões de provisionamento devem ser feitas de forma *on-line*.

A proposta deste trabalho apresenta uma infraestrutura que permite a elasticidade de recursos de forma *on-line*, por meio da alocação/desalocação de recursos, conforme a demanda. Para endereçar a dificuldade de tradução de requisitos de CPU e memória em requisitos de *QoS*, propõe-se elasticidade baseada em variações no tempo de resposta de uma aplicação.

2.2.2 Consolidação de servidores

A consolidação de servidores é uma abordagem para maximizar a utilização de recursos e minimizar o consumo de energia em um ambiente de computação em nuvem. A movimentação automática de recursos de servidores subutilizados de modo a consolidá-los e a configuração destes servidores em modo econômico de energia permitem corte de gastos. Porém, essa consolidação deve ocorrer de modo que não atrapalhe o desempenho das aplicações [51].

Dentro deste tópico, este trabalho propõe uma infraestrutura de processamento que é capaz de aumentar a consolidação de aplicações heterogêneas no mesmo conjunto de recursos, otimizando a competição por CPU e memória; e permitindo a coexistência de bibliotecas e serviços passíveis de incompatibilidade. A tecnologia que permite esta consolidação são os *containers*, que serão descritos na Seção 2.3.3.

2.2.3 Gerenciamento de tráfego, monitoramento e análise

A arquitetura distribuída dos sistemas de computação em nuvem faz com que seja importante a provisão de meios de agregação de *logs*, de forma que estas informações possam ser utilizadas proativamente na otimização da experiência dos usuários. Devem estar disponíveis tanto informações de serviço fim-a-fim, como informações de serviços individuais, ativos de computação e rede [51].

Além disso, é necessária a provisão de meios para monitoramento que permitam [1]:

- Manter os ativos funcionando de forma otimizada e com máximo desempenho;
- Detectar variações de desempenho nas aplicações e recursos;
- Verificar questões de *SLA* (*Service Leve Agreement*) e violações de parâmetors de qualidade de serviço (*QoS*);

Este trabalho realiza análise de *logs* de um balanceador de carga usando ferramentas de consolidação de *logs* e *Big Data*, de forma a permitir que se observe o desempenho atual de uma aplicação e seja possível o ajuste no provisionamento de recursos, com o objetivo de se obter os requisitos de desempenho desejados.

2.2.4 Segurança de dados

Provedores de computação em nuvem devem prover mecanismos para que seja possível o atendimento de recursos de segurança como, por exemplo, confidencialidade para acesso e transferência de dados e, também, auditabilidade, possibilitando a verificação de possíveis atentados a segurança da infraestrutura e das aplicações [51]. As várias camadas de abstração presentes nas arquiteturas da computação em nuvem, além da possibilidade de compartilhamento de recursos entre diversos clientes, fazem com que o atendimento dos requisitos de segurança devam ser observados. Este trabalho não aborda a segurança de dados.

2.2.5 *Frameworks* de *Software*

A computação em nuvem provê os recursos necessários para a hospedagem de aplicações de larga escala e com utilização massiva de dados. Normalmente, o processamento massivo de dados é feito via *frameworks* de *MapReduce* como, por exemplo, o *Hadoop* ou *Spark*. Esses *Frameworks* são construídos de forma distribuída e, normalmente, fazem uso intenso de CPU, memória, leituras e escritas em disco (*I/O*) [51]. Aplicações em larga escala estão comumente presentes em ambientes de nuvem. Desta forma, o provedor de nuvem de ser

capaz de realizar alocação eficiente de recursos para estas cargas de trabalho, de modo a diminuir o impacto da possível competição por recursos em um *datacenter*.

Este trabalho propõe o uso de um orquestrador de recursos de computação (orquestrador de *containers*) para, dentre outras funções, realizar a distribuição de cargas de trabalho de forma igualitária entre os membros de um *cluster*.

2.2.6 Tecnologias de Armazenamento e Gerenciamento de Dados

Os *frameworks* de *MapReduce* utilizam sistemas de arquivos distribuídos para a manipulação de dados em alta escala de forma distribuída. Alguns exemplos de sistemas de arquivos distribuídos são o *HDFS* e o *GFS*. Esses dois sistemas não implementam a interface POSIX padrão para sistemas de arquivo, o que leva ao risco de incompatibilidade no acesso a dados para sistemas legados [51] . Portanto, o provedor de nuvem deve ser capaz de prover *APIs* para acesso de dados que abstraiam essas limitações e permitam maior interoperabilidade entre sistemas.

O uso massivo de dados em ambientes de nuvem faz com que seja necessária a redução dos custos de armazenamento, mantendo bons níveis de serviço para os usuários. O uso de técnicas de *tiering* permite que dados que não são acessados a determinado tempo sejam movidos para camadas de armazenamento mais baratas como,por exemplo, fitas e discos *SATA*; enquanto dados com acessos mais recentes permaneçam em discos com melhor desempenho, porém mais caros, como *SSD* e *SAS*. Isto de forma transparente para o usuário [50]. Outra tecnologia de armazenamento que permite redução de custos em ambientes de nuvem é a deduplicação de dados, onde blocos comuns de dados são armazenados uma única vez e passam a ser referenciados via ponteiros no sistema de arquivos [31].

2.2.7 Eficiência Energética

Outro tópico de interesse na pesquisa sobre computação em nuvem é a eficiência energética. Segundo [28], atualmente, o consumo de energia gerado por *datacenters* no mundo representam de 1.1% a 1.5% do total do consumo de energia mundial. Isso faz com que seja necessário que se repense as questões de eficiência energética nestes *datacenters*.

A realização de dimensionamento correto de sistemas sem superdimensionamentos é importante para a eficiência energética de *datacenters*, na medida em que menos entidades físicas são necessárias para suportar a demanda, o que se traduz em menos energia para alimentação de equipamentos, assim como menor energia necessária para resfriamento dos mesmos [28].

Este trabalho propõe uma infraestrutura capaz de melhorar a eficiência energética de um *datacenters*, por meio da alocação de recursos computacionais que escalam de acordo com a demanda, evitando dimensionamentos idealizados para suportar picos de demanda.

2.2.8 Integração de Internet das Coisas (*IoT*) com Computação em Nuvem

A crescente popularização e barateamento de dispositivos móveis, como *smart phones*, assim como o crescente uso de dispositivos embarcados e sensores, gera uma crescente demanda por meios de processar os dados gerados por estes dispositivos, assim como estender as suas funcionalidades. Normalmente, dispositivos de *IoT* trabalham de forma descentralizada, algumas vezes com conectividade e capacidade de processamento e armazenamento limitadas [4].

Por outro lado, a computação em nuvem provê meios para disponibilização de recursos computacionais massivos e conectividade ubíqua. Isso faz com que a integração de *IoT* e computação em nuvem tenham um crescente interesse, na medida em que se pode obter meios para agregação, análise e geração de valor dos dados obtidos, assim como a disponibilização de aplicações que possam prover serviços para estes equipamentos. Assim, a grande carga de trabalho a ser gerada por esses inúmeros dispositivos provoca preocupações em muitos dos itens mencionados anteriormente.

2.3 Tecnologias para Computação em Nuvem

Nesta seção serão descritas algumas tecnologias para provisão de ambientes de computação em nuvem e será justificada a adoção de determinadas tecnologias para compor a arquitetura da solução proposta neste trabalho.

2.3.1 Arquitetura de Serviços *Web* de Grande Demanda

Uma abordagem comum para disponibilização de alta performance, escalabilidade e disponibilidade é o uso de arquiteturas distribuídas, que podem rotear as requisições para diversos servidores, de forma transparente ao usuário. Essa abordagem, além de melhorar a performance e a disponibilidade dos serviços, pode aumentar sua escalabilidade ao se adicionar e remover membros no *cluster*.

A Figura 2.1 mostra uma arquitetura baseada na proposta de [16]. Os clientes fazem acesso aos recursos providos pelo conjunto de servidores *web* (*cluster*), passando pelo balanceador de carga. O balanceador de carga distribui as requisições para os membros do *cluster web* de forma transparente para o usuário. Nesta figura, o conjunto de servidores

web utiliza uma camada de banco de dados para armazenamento de dados persistentes da aplicação. Para aumentar o desempenho, assim como a escalabilidade da aplicação, pode-se integrar um serviço de *cache*, que armazena dados comumente acessados pela aplicação em memória RAM. Isso diminui a latência de acesso a dados e pode diminuir a carga nos servidores de bancos de dados [32]. Além disso, deve estar disponível armazenamento de rede para armazenamento de arquivos comuns aos servidores de aplicação, como, por exemplo, arquivos de configuração, códigos da aplicação e arquivos de dados.

Uma arquitetura como esta pode aumentar a disponibilidade e o desempenho de uma aplicação, pois a arquitetura distribuída permite que se eliminem pontos únicos de falha, além de aumentar o número de servidores disponíveis para atendimento das requisições em paralelo. É importante ressaltar que deve-se preocupar com a redundância de todos os componentes apresentados nessa arquitetura. Por exemplo, o balanceador de carga deve ter redundância na forma ativo-passivo, na qual, se o balanceador principal apresentar problemas, o passivo passa a atender as requisições. A mesma preocupação deve estar presente para os demais componentes da solução, sejam eles os dispositivos de armazenamento em rede, bancos de dados, serviço de *cache* ou qualquer outro componente que esteja provendo serviços dentro da arquitetura.

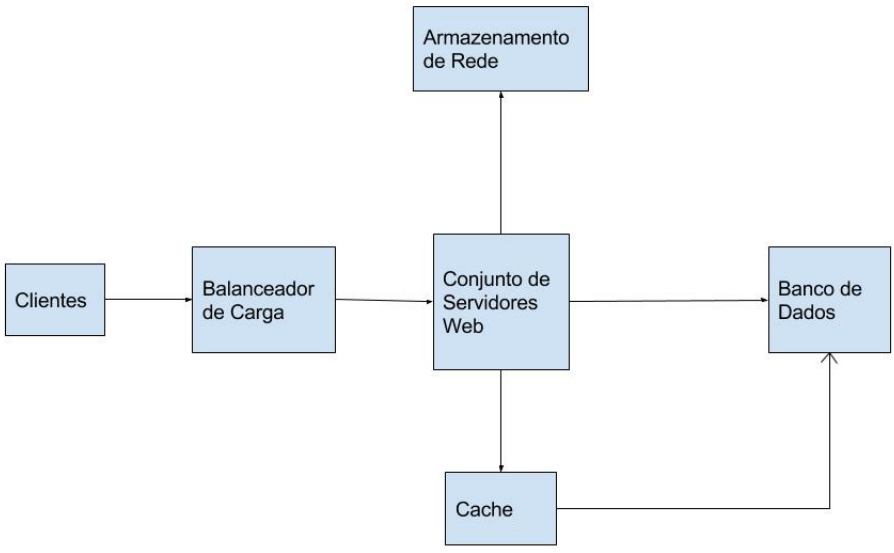

Figura 2.1: Arquitetura para serviços web de grande demanda

Porém, vários desafios se impõem para que um *cluster* distribuído de servidores possa funcionar de forma eficiente como se fosse um único servidor. Esses desafios vão desde o

roteamento das requisições para os membros do *cluster*, métodos para escolha do membro que receberá a demanda, até a manutenção do estado da conexão, entre outros. Além disso, os métodos de balanceamento de carga podem estar nas camadas 4 ou 7 do modelo OSI.

No caso do balanceamento de camada 4, as requisições são distribuídas entre os membros do *cluster*, a partir de atributos da camada de transporte, como TCP e UDP. O balanceador distribui as conexões de clientes que conhecem o endereço IP do *cluster*, entre os diversos servidores (*pool*), que efetivamente respondem as requisições. Neste caso, como a distribuição é feita com foco na camada 4, o balanceador escolhe um servidor, sem considerar o conteúdo ou o tipo da requisição.

O balanceamento de camada 7 é um balanceamento de carga no nível da aplicação e é capaz de distribuir as requisições para os servidores baseado em diferentes tipos de requisições e conteúdo, de modo que se possa prover requisitos de qualidade de serviço para diferentes tipos de conteúdo. Apesar da grande flexibilidade deste modelo, existe uma sobrecarga para realização da inspeção do conteúdo da requisição, o que pode se traduzir em menor escalabilidade.

Os métodos de escolha do servidor do *cluster* que receberá a requisição podem ser baseados em heurísticas como *hash* de IP de origem, em que o balanceador escolhe o destino de acordo com um *hash* do *IP* de origem de cada pacote. Pode ser utilizado também heurísticas, como enviar a requisição para o servidor que tem menos conexões. Outro método é a utilização de *round robin*, em que o servidor é escolhido de acordo com um peso designado a ele. Outros métodos podem utilizar dados como consumo de CPU e memória e escolher o membro que apresenta menor sobrecarga. É interessante que haja uma checagem de saúde dos membros do *cluster* para permitir que não sejam encaminhadas requisições para membros incapazes de servir a conexão.

Outra questão importante é a persistência da conexão. Uma vez que um cliente se conecta em uma instância da aplicação, é necessário que ele continue conectado a ela para garantir que o cliente tenha acesso ao estado da aplicação armazenado na instância. Diversos métodos podem ser utilizados, como, por exemplo:

- Persistência simples: Neste caso a persistência é baseada em características de rede, como, por exemplo, IP de origem e porta de destino;
- Persistência baseada no ID da sessão SSL: Para evitar que seja necessária a renegociação da sessão SSL, o ID da sessão SSL é usado no balanceador para garantir que as sessões sejam roteadas para a instância da aplicação que ela se conectou inicialmente;

- Persistência baseada em *cookies*: O balanceador usa o cabeçalho do *cookie* HTTP para persistir as conexões em uma sessão;
- Persistência baseada em *hash*: Esse método utiliza diversos valores de uma requisição para realizar a persistência. O IP de origem, IP de destino e porta de destino são utilizados para geração do Hash.

2.3.2 Virtualização, Gerenciadores de Nuvem e Automação de Configuração

Virtualização usa *software* para simular as funcionalidades de *hardware* e assim criar um sistema computacional virtual. Isso permite que vários sistemas operacionais e aplicações distintas sejam executados em um mesmo servidor [47].

Uma máquina virtual é uma porção de *software* isolada com um sistema operacional que pode ter uma aplicação sendo executada nele. Desta forma, cada máquina virtual é independente de outras máquinas virtuais. Para prover este desacoplamento, é utilizada uma camada de *software* chamada *hypervisor*, que é responsável pela execução das máquinas virtuais e pelo gerenciamento do uso dos recursos físicos do servidor (CPU, memória, interfaces de rede, dispositivos de armazenamento, gerenciamento de interrupções de entrada e saída, entre outros), assim como sua alocação para cada uma das máquinas virtuais sob seu controle.

Desta forma, o uso de máquinas virtuais traz diversos benefícios para uma infraestrutura de computação, como, por exemplo:

- Execução de múltiplos sistemas operacionais na mesma máquina física;
- Divisão de recursos entre máquinas virtuais;
- Provisão de isolamento de falhas e segurança entre as máquinas virtuais;
- Preservação de desempenho por meio de controle de recursos;
- Geração de imagens de sistema, clones e *backups* de estados na forma de arquivos;
- Restauração rápida de sistemas;
- Provisionamento rápido e facilitado de máquinas virtuais;
- Ajustes de recursos sob demanda.

Para provisão de funcionalidades desses ambientes, gerenciadores de nuvem podem ser utilizados [38]. Esses orquestradores normalmente se comunicam com os *hypervisors*, ativos de rede e equipamentos de armazenamento por meio de *APIs* (*Application Program Interfaces*).

Ferramentas de automação de configuração permitem que se defina a configuração desejada para um conjunto de servidores e essas configurações sejam aplicadas de forma automática [48]. Isso agiliza o provisionamento de servidores, assim como o de serviços. Além disso, é possível que se garanta que a configuração dos ambientes está em conformidade com o que se deseja e obtenha-se potencial diminuição de falhas de configuração devido a falhas humanas.

Desta forma, a integração de máquinas virtuais com ferramentas de gerenciamento de nuvem e ferramentas de automação de configuração provê funcionalidades de ambientes de computação em nuvem, como, por exemplo:

- Provisionamento de ambiente *self-service* de computação;
- Automação do ciclo de vida de máquinas virtuais e aplicações;
- Gerenciamento de identidade, perfis de gerenciamento e uso;
- Telemetria e bilhetagem do uso de recursos computacionais;
- Estabelecimento de cotas de uso de recursos.

É importante citar que é possível a provisão de serviços de computação em nuvem sem o uso de virtualização, porém este tipo de cenário apresenta menor flexibilidade para realização de automações, devido as funcionalidades citadas nesta seção.

2.3.3 Containers

Container é uma tecnologia para a criação de instâncias de processamento separadas ou isoladas que permitem a virtualização no nível do sistema operacional ao disponibilizar porções protegidas de processamento. Dois *containers* executando no mesmo sistema não sabem que estão compartilhando recursos, pois tem sua própria abstração de camada de rede, memória e processos [9].

Esse isolamento é feito via *Kernel namespaces*. Um *namespace* cobre um recurso global do sistema em uma abstração que faz com que pareça que processos dentro do *namespace* tem sua própria instância isolada do recurso global. Mudanças em um *namespace* são vistas apenas por processos que são membros daquele *namespace* [25]. O *Linux* implementa *namespaces* para sistema de arquivos, processos, rede e usuários [12]. Desta forma, processos sendo executados dentro de um *container* parecem estar sendo executados em um sistema *Linux* individual, apesar de estar compartilhando o *kernel* com outros *containers*. Diferentemente das máquinas virtuais que executam um sistema operacional completo, um *container* pode ter, por exemplo, apenas um processo.

A limitação e contabilização de recursos (CPU, memória, espaço em disco e E/S) dos *containers* é feita por meio dos *cgroups*. Os *cgroups* permitem a alocação desses recursos por meio de grupos de processos definidos pelo usuário em um sistema. Um *container* pode ser redimensionado simplesmente alterando-se os limites do seu *cgroup* correspondente [35].

Os *containers* apresentam uma maior portabilidade que as máquinas virtuais, ao serem configurados de forma genérica para qualquer sistema operacional baseado em Linux. A virtualização via *hypervisors* consome mais recursos do que a virtualização por *containers*, dado que os últimos são executados em sistemas operacionais que executam em espaços isolados entre si. Se um *container* não está executando nenhuma tarefa, ele não está consumindo recursos no servidor [9]. Além disso, os *containers* apresentam um grande dinamismo para serem criados e destruídos, dado que apenas tem que iniciar ou destruir processos em seu espaço isolado. Portanto, verifica-se que existem vantagens em se utilizar a tecnologia de *containers*. Essa foi a tecnologia escolhida para atender o processamento das cargas de trabalho *http* nesta solução.

2.4 Carga de Trabalho

Esta seção descreve a importância da caracterização da carga de trabalho para avaliação de sistemas computacionais e o método utilizado neste trabalho para realização dessa caracterização.

2.4.1 Caracterização da Carga de Trabalho

O correto dimensionamento de um sistema computacional depende do conhecimento do comportamento da carga a qual o sistema será submetido. Neste sentido, é importante a caracterização estatística da carga. Essa caracterização é feita por meio da análise dos atributos estatísticos presentes nas séries temporais que representam a carga, de modo que se possa obter modelos matemáticos que permitam:

- A aplicação de modelos matemáticos providas pela definição das capacidades que os componentes do sistema devem ter para atendimento das demandas de forma adequada;
- Permitir a construção de geradores de carga que auxiliem a verificação do correto dimensionamento e funcionamento do sistema por meio de testes de carga e simulação.

Vários modelos matemáticos foram produzidos baseados nos modelos de Markov, que consideram que as fontes de carga possuem comportamento Poissoniano [3]. Porém ,[7] demonstrou que os acessos *web* possuem comportamento fractal.

Os processos fractais podem ser do tipo monofractais ou multifractais. Os processos multifractais têm aplicação para a caracterização de tráfegos em alta frequência, observados em pequenas escalas de tempo, na faixa de milissegundos [34]. Esses processos não serão abordados neste trabalho, pois a caracterização da chegada de requisições oriundas do *http* se baseia na coleta de dados de acesso obtidos em *logs* de servidor *web*, que fornecem valores na escala de segundos. Portanto, este trabalho utilizará o processo monofractal, mais especificamente os processos autossimilares para caracterização da chegada de requisições *web*.

2.4.2 Autossimilaridade

Dada uma série temporal estacionária, com média zero $X = (X_t; t = 1, 2, 3, ...)$, define-se uma série m-agregada $X^m = (X_k^m; k = 1, 2, 3, ...)$ como a soma da série original X pelos m blocos não sobrepostos. Diz-se que X é H-autossimilar se para todos m positivos, X^m tem a mesma distribuição que X na escala de m^H [7]. Isto é,

$$X_t = m^{-H} \sum_{i=(t-1)m+1}^{tm} X_i, \forall m \in N \tag{2.1}$$

Se X é H-autossimilar, ele tem a mesma função de autocorrelação $r(k) = E(X_t - \mu)(X_{t+k} - \mu])/\sigma^2$, assim como a série X^m para todo m. Isso significa que a distribuição da série agregada é a mesma (exceto pela diferença na escala) da série original.

Como consequência, os processos autossimilares podem apresentar dependência de longa duração, fato que não é verificado em séries que possuem características poissonianas. Um processo com dependência de longa duração tem função de autocorrelação $r(k) \sim k^{-\beta}$ com $k \to \infty$, onde $0 < \beta < 1$. Portanto, a função de correlação desse processo segue a lei das potências, que se difere das distribuições comumente usadas na modelagem para cargas de rede, que tem decaimento exponencial. Como estas funções apresentam decaimento lento, a soma dos valores de autocorrelação deste tipo de série se aproxima do infinito. Isso tem algumas implicações. Primeiro, a variância da média de n amostras desse tipo de série não diminui proporcionalmente a $1/n$, mas sim proporcionalmente a $n^{-\beta}$. Segundo, o espectro de potência desse tipo de série é hiperbólico, tendendo para o infinito na frequência zero, o que reflete a influência infinita da dependência de longa duração nos dados

Um dos atributos mais interessantes das séries auto similares é o parâmetro de Hurst, que demonstra o grau de autossimilaridade de uma série, sendo este definido como: $H = 1-\frac{\beta}{2}$. Portanto, uma série autossimilar com longa dependência tem $\frac{1}{2} < H < 1$. Com $H \to 1$, tanto o grau de autossimilaridade quanto a dependência de longa duração aumentam. [7]

2.4.3 Estimativa do Parâmetro H

Existem vários métodos para a estimativa do parâmetro de Hurst (H). Neste trabalho serão apresentados os métodos R/S e Kettani-Gubner, que foram utilizados para caracterizar a carga de trabalho.

Análise R/S

Seja uma série infinita $X_k, k = 1, 2, ..., n$, com média $Y(n)$ e variância das amostras dadas por $S^2(n)$, então a amplitude reescalonada será dada por [34]:

$$\frac{R(n)}{S(n)} = \frac{1}{S(n)}[max_{0\leq t\leq n}(Y(t) - tY(n)) - min_{0\leq t\leq n}(Y(t) - tY(n)) \tag{2.2}$$

Segundo Hurst, várias séries temporais satisfazem a seguinte relação empírica:

$$E[\frac{R(n)}{S(n)}] \approx cn^H, n \to \infty \tag{2.3}$$

onde c é uma constante finita, independente de n, e S(n) é o desvio padrão. Desse modo, a declividade da reta ajustada para *(log(n), log(R(n)/S(n)))* fornece uma estimativa para o parâmetro de Hurst.

2.4.4 Método Kettani-Gubner

Seja um processo exatamente autossimilar de segunda ordem. Neste caso, o coeficiente de autocorrelação é descrito pela equação,

$$R(k) = 1/2(|k+1|^{2H}) - 2|k|^{2H} + |k-1|^{2H} \tag{2.4}$$

Para k=1 tem-se $R(1) = 2^{2H-1}$, isolando H, têm-se: $\hat{H} = \frac{1}{2}[1 + \log(1 + \hat{R_n}(1))]$. Sob a suposição de que o processo seja ergódico, é possível calcular o parâmetro H com [3]:

$$\hat{H_n} = \frac{1}{2}[1 + log(1 + \hat{R_n}(1))] \tag{2.5}$$

A vantagem do método Kettani-Gubner é que ele permite o cálculo do parâmetro de H, utilizando séries menores do que as necessárias para o método R/S, além de ter menor custo computacional.

2.5 Controladores PID

Esta seção descreve os controladores PID (proporcional-integral-derivativo), que foram utilizados neste trabalho como parte integrante da arquitetura para prover um ambiente elástico.

2.5.1 Descrição

Controladores PID (Proporcional-Integral-Derivativo) são um dos algoritmos de controle mais utilizados na industria. Suas aplicações vão desde controle de temperatura em ambientes até a manutenção de estabilidade em voo e controle de trajetória para *drones* [21].

Os controladores PID possuem três coeficientes: Proporcional, integral e derivativo. Esses coeficientes são variados de forma a se obter a resposta de controle ideal desejada para um processo.

Um controlador PID trabalha dentro de um sistema em malha fechada, onde é possível a leitura do estado atual de determinada variável que se deseja controlar e, de acordo com seu valor, uma ação é executada (atuador) de modo que a variável chegue e tente permanecer no nível desejado (apesar de distúrbios externos), nas próximas iterações de tempo [21].

Sendo assim, o PID deve ler o estado atual da variável e calcular a resposta da saída do atuador por meio do cálculo dos componentes proporcional, integral e derivativo, e então somar os três para calcular a saída.

A Figura 2.2 mostra um sistema de malha fechada, onde é possível ver um ciclo de realimentação em um circuito fechado. Um exemplo de sistema operando, conforme esta figura, seria um sistema projetado para controle de temperatura de um determinado ambiente. Digamos que se deseje manter a temperatura de um ambiente em 20 graus *celsius*. Nesse caso, o *setpoint* seria 20. Porém, digamos que a temperatura lida pelo sensor indique que o ambiente está com 30 graus *celsius*. Neste caso, o controlador verificaria a temperatura atual do ambiente e deveria acionar o atuador (ar condicionado, por exemplo), de modo que este enviasse uma resposta ao ambiente que fizesse diminuir a sua temperatura (ar frio), mesmo na presença de distúrbios, como, por exemplo, a abertura e fechamento de janelas.

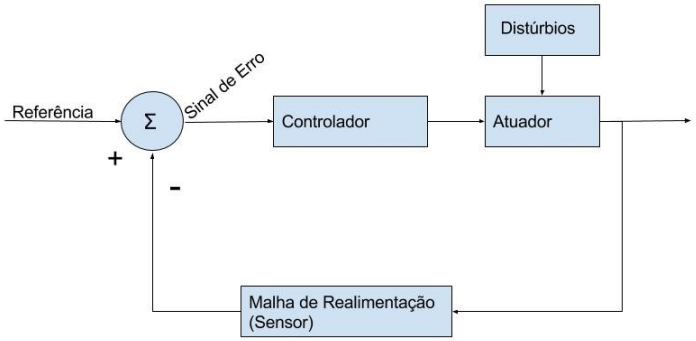

Figura 2.2: Exemplo de Malha de Controle Fechada

A Figura 2.3 mostra o esquema do controlador PID. O valor atual da variável de controle é lido do sistema *S*. Esse valor é comparado com o valor desejado *setpoint* e é gerado o termo de erro *e(t)* com esta diferença. O termo de erro é combinado nos componentes proporcional, integral e derivativo e é gerado um sinal de controle que atua no sistema *S*, de modo a controlar o valor da variável. Estes componentes são descritos nas subseções 2.5.2, 2.5.3 e 2.5.4.

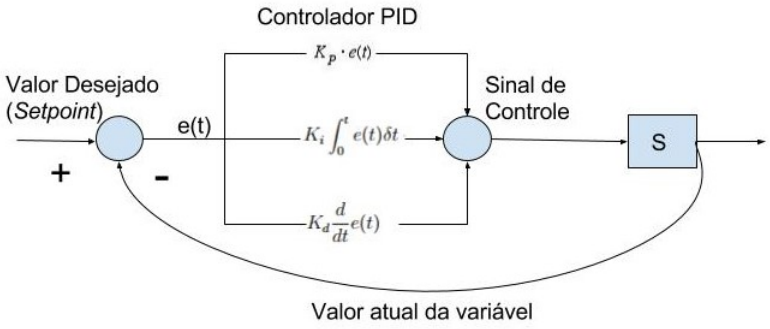

Figura 2.3: Esquema controle PID

2.5.2 Resposta Proporcional

O componente proporcional depende da diferença entre o valor desejado (*setpoint*) e o valor atual da variável. Essa diferença é referida como erro. O ganho proporcional (*Kp*) determina a taxa de resposta de saída para um sinal de erro.

2.5.3 Resposta Integral

O componente integral soma o termo de erro ao longo do tempo. Sendo assim, mesmo um pequeno erro fará com que o componente integral aumente lentamente. Isto é, a

resposta integral aumenta ao longo do tempo, a menos que o erro seja zero. Portanto, este componente tem o poder de conduzir o erro de estado estacionário para zero.

2.5.4 Resposta Derivativa

O componente derivativo faz com que a saída diminua, caso a variável do processo esteja aumentando rapidamente. A resposta derivativa é proporcional a taxa de variação da variável do processo.

2.5.5 Ajustes dos Parâmetros do PID

Para que o controlador PID produza os ajustes necessários ao sistema, os parâmetros de ganho K_p, K_i e K_d devem ser ajustados. Existem diversos métodos de ajuste desses parâmetros, por exemplo, o método manual, o método Ziegler-Nichols [21] e o ajuste utilizando o algoritmo *Coordinate Ascent* para otimização de parâmetros. O método manual e o *Coordinate Ascent* foram os métodos utilizados neste trabalho.

No método manual, os ganhos de cada um dos componentes são ajustados usando tentativa e erro. Para isso, ao ajustar os parâmetros, deve-se conhecer os efeitos que cada parâmetro provoca na saída do controlador. Neste método, os termos K_i e K_d são ajustados para zero e o termo K_p é aumentado até que a saída do ciclo comece a oscilar. A partir daí aumenta-se lentamente o termo K_i para reduzir o erro estacionário. Neste ponto, inicia-se o incremento do termo K_d, de modo a diminuir as oscilações na saída do ciclo [21].

No método Zigler-Nichols, no início, os termos K_i e K_d são ajustados para zero e o termo K_p é aumentado até que a saída do *loop* comece a oscilar. Nesse ponto, registra-se o valor de K_p como a variável K_u, o período das oscilações T_u também é registrado e os parâmetros são ajustados conforme a Tabela 2.1.

Tabela 2.1: Método Ziegler-Nichols

Tipo de Controlador	K_p	K_i	K_d
P	$0.5K_u$	-	-
PI	$0.45K_u$	$1.2\frac{K_p}{T_u}$	-
PID	$0.6K_u$	$2\frac{K_p}{T_u}$	$\frac{K_pTu}{8}$

O método *Coordinate Ascent* (CA) foi utilizado pela equipe vencedora do *"Darpa Grand Challenge"* no desenvolvimento do carro autônomo "Stanley" para otimização de parâmetros do modelo probabilístico de estimativa de seu posicionamento citeth-run2006stanley.

Este algoritmo consiste em um laço que visa maximizar ou minimizar uma função de pontuação. Essa função define quão próximo do objetivo uma variável permaneceu dentro de uma iteração do algoritmo. Assim, dada uma estimativa inicial de um conjunto de parâmetros que afeta o valor da função de pontuação, o algoritmo modifica cada um desses parâmetros com base em um valor fixo (passo). A partir desta modificação, o algoritmo verifica se essa alteração representou uma melhoria na função de pontuação. Se houver a melhoria, esse parâmetro é registrado, assim como o novo valor da função de pontuação e aumenta-se o passo de forma gradual (ex: aumenta-se em 10 %). Caso não haja melhoria, o passo é diminuído de forma gradual (ex: diminui-se em 10 %), até que o intervalo de busca do parâmetro seja menor do que um valor mínimo pré-fixado. O pseudo código do Algoritmo 1 descreve o CA.

A implementação que realiza o ajuste dos parâmetros do PID usando o *Coordinate Ascent*, está descrita na Seção 3.1.6.

Algoritmo 1: Coordinate Ascent

Entrada: Parâmetros a serem otimizados
Saída: Parâmetros otimizados

```
início
    Enquanto a soma dos passos para cada parâmetro > limiar mínimo
      Para cada parâmetro
        aumente o parâmetro com o valor do passo
        calcule o valor da pontuação atual
      Se pontuação atual > pontuação anterior
        mantenha o valor de parâmetro
        aumente o passo
        registre o valor da pontuação
      Se pontuação atual < pontuação anterior
        diminua o parâmetro com o valor do passo
        calcule o valor da pontuação atual
        Se pontuação atual > pontuação anterior
          mantenha o valor de parâmetro
          aumente o passo
          registre o valor da pontuação
        Se pontuação atual < pontuação anterior
          diminua o passo
fim
retorna Parâmetros otimizados
```

2.6 Trabalhos Relacionados

Sobre balanceadores de carga, [5] realiza uma avaliação de diversos mecanismos de balanceamento para serviços *web* (*round robin*, escolha aleatória, entre outros), considerando os impactos de cada método no desempenho do sistema. Se avaliados também os ganhos de se considerar o estado (conteúdo do *cache* e carga) de cada um dos servidores do *pool* nas decisões de balanceamento de carga. Esse estudo é interessante pois alerta para a possibilidade de diminuição de taxa de acerto de *cache*, caso o estado de *cache* de cada servidor não seja considerado nas decisões de balanceamento. O uso de afinidade de *cache* nas decisões de balanceamento, enquanto pode aumentar a taxa de acerto de *cache*, pode trazer balanceamento desigual entre os membros do *pool*.

O trabalho [14] mostra uma pesquisa do estado da arte dos métodos de balanceamento de carga em sistemas *Web*. São apresentadas diversas classificações de balanceadores. Essas classificações se dão a partir de características, como balanceamento baseado em conteúdo ou não, uso de IPs virtuais (VIP) para acesso de clientes ou disponibilização dos próprios IPs dos membros do *pool* para acesso dos clientes, balanceamentos de carga no nível de *hardware* ou no nível de *software*. Outra classificação apresentada diz respeito ao caminho do fluxo de volta ao cliente, se esse passa novamente no balanceador ou se vai direto do servidor para o cliente. No caso do balanceamento não consciente do conteúdo (camada 4 do modelo OSI), são apresentados diversos algoritmos para escolha de qual servidor receberá determinada requisição. Nesse caso, apresentam os algoritmos *Round Robin, Weighted Round Robin, Least Connection, Weighted Least-Connection, Least Loaded e Random Server*[14].

Os balanceadores conscientes do conteúdo são apresentados como balanceadores no nível da aplicação (camada 7 do modelo OSI). Nesse caso, o conteúdo da requisição *http* deve ser avaliado pelo balanceador, que decidirá para onde enviar as requisições. Os autores comentam que, apesar de já existirem produtos de mercado atuando desta forma, esse ainda é um assunto sendo investigado. Apesar disso, os autores mostram diversos trabalhos que demonstram o potencial dessa abordagem para produção de algoritmos de distribuição mais flexíveis e inteligentes [14]. É interessante comentar que, para que esse tipo de balanceador apresente ganhos, a arquitetura dos servidores *web* tem que ser desenhada neste sentido, isto é, deve-se ter servidores especializados em diversos tipos de conteúdo. Por exemplo, poderiam existir servidores otimizados para servir conteúdos estáticos, outros para conteúdos dinâmicos, outros para conteúdos de vídeo e assim por diante. Por fim, na seção de balanceadores no nível da aplicação são listados os diversos métodos de estado da arte para as políticas de distribuição das requisições (políticas com considerações de *QoS*, localidade, controles de admissão, granularidade de requisição com HTTP/1.1, considerações de cache, etc). O interessante deste trabalho é que ele

apresenta as diversas opções para realização de balanceamento de carga, demonstrando as implicações e complexidades que cada método pode trazer ao ambiente.

O trabalho [26] realiza uma comparação entre os diversos métodos de se obter auto-escalabilidade e elasticidade em um ambiente de nuvem. Esses métodos são separados nas categorias reativos e preditivos. São descritos métodos baseados em aprendizado de máquina, teoria de filas, teoria de controle, análises de séries temporais e baseados em limiares. Esse estudo mostra a variedade de técnicas que estão sendo propostas para provisão de auto-escalabilidade e elasticidade, sendo que essas técnicas estão em diferentes áreas de conhecimento.

Existem outros trabalhos com dimensionamento elástico (auto-escalável) para ambientes de nuvem. No trabalho PRESS [15], os autores propõem algoritmo de predição de carga de CPU, que extrai padrões de consumo e ajusta a alocação de recursos. A abordagem dos autores utiliza dois métodos para realização de previsões em linha. O primeiro se baseia em uso de processamento de sinais (FFT) para extração de frequências dominantes. Com essas frequências são geradas séries temporais e diversas janelas de tempo são comparadas, sendo gerado o índice de correlação de Pearson para as várias janelas comparadas. Caso se obtenha índice de correlação maior que 0.85, o valor médio do uso de recursos dentro de cada posição da série temporal é utilizado para gerar uma previsão para a próxima janela e os recursos das máquinas virtuais são ajustados. Caso um padrão não seja identificado, os autores propõem uma abordagem que utiliza uma cadeia de Markov com número finito de estados para a realização de previsões.

Outro trabalho que realiza dimensionamento elástico (auto escalável) para ambientes de nuvem é o do Haven [33]. Essa proposta utiliza ferramentas de sistema operacional de nuvem para monitoramento de cargas de CPU e memória a que cada máquina virtual de um *pool* de balanceamento está submetida. A partir de limiares previamente estabelecidos para consumo de CPU e memória, o Haven pode instanciar novas máquinas virtuais e realizar a sua inserção em um *pool* de balanceamento de carga. Além disso, o balanceador de carga, implementado via SDN (*Software Defined Network*), tem inteligência para realizar o encaminhamento da requisição para o membro com melhores condições de carga.

O trabalho [19] propõe e disponibiliza uma ferramenta nativa de auto escalabilidade chamada de *Horizontal Pod Autoscaler* (HPA). Essa ferramenta trabalha escalando o ambiente a partir de limiares médios de consumo de CPU dos *containers* que atendem a determinada aplicação.

A proposta desse trabalho se diferencia da abordagem do PRESS na medida que não realiza predição de carga, mas sim um dimensionamento de recursos, baseado na observação do tempo de resposta de uma aplicação atrás de um balanceador de carga. Outra diferença é que o PRESS realiza escalabilidade vertical, ou seja, realiza aumento de

recursos de memória e CPU para se ajustar a carga de trabalho, enquanto que na proposta desse trabalho se aborda a escalabilidade horizontal, ou seja, novas instâncias capazes de atender a carga de trabalho são alocadas atrás de um balanceador de carga, de modo a se ajustar às variações na demanda. A escalabilidade horizontal tem a vantagem de que os recursos alocados para atender a carga de trabalho não são limitados aos recursos físicos de uma máquina. Além disso, essa abordagem facilita a alta disponibilidade, uma vez que as demandas são atendidas por um conjunto de instâncias em paralelo.

Em relação à abordagem do Haven, a proposta desse trabalho apresenta um método de dimensionamento do sistema a partir da observação do tempo de resposta das requisições. Isso permite uma visibilidade global do comportamento do sistema, pois nos casos em que não haja excessos de consumo de processamento e memória, o ambiente pode se beneficiar do aumento do paralelismo no atendimento das requisições. O Haven também realiza escalabilidade horizontal.

Outra diferença na proposta desse trabalho é a infraestrutura utilizada. Os trabalhos PRESS e HAVEN trabalham com a tecnologia de máquinas virtuais e a proposta desse trabalho utiliza *containers* que são considerados nas publicações recentes [9] como mais leves, por não fazerem virtualização de *hardware* e, para tal, é utilizado o *Kubernetes* como orquestrador de *containers*.

Finalmente, outro diferencial na proposta desse trabalho é que as medidas de tempo de resposta de uma aplicação permitem decidir se o ambiente deve ser escalado. Enquanto o HPA realiza medidas de consumo de CPU nos próprios *containers*, a proposta desse trabalho realiza medidas externas aos *containers*, no caso, são realizadas medidas do tempo médio de resposta das aplicações no balanceador de carga. Essa abordagem traz a vantagem de poder observar o desempenho do ambiente, independente do nível de utilização dos recursos dos *containers*.

2.7 Conclusão deste Capítulo

Este capítulo apresentou a revisão da literatura e descreveu os componentes tecnológicos que serão usados para a proposta da solução deste trabalho. A Computação em Nuvem foi descrita, assim como os outros componentes da solução: *Containers*, sistemas Web e balanceadores de carga; caracterização das requisições como processo autossimilar e controladores PID. Foi também feita uma revisão bibliográfica de temas relacionados a este trabalho, assim como um posicionamento da proposta deste trabalho em relação aos que também propõem ambientes elásticos auto escaláveis.

Capítulo 3

Solução Proposta

Este trabalho tem como objetivo propor uma arquitetura de computação inovadora em nuvem para provisão de auto elasticidade. A solução propõe um ambiente de *cluster* baseado em *containers*, e um método de auto elasticidade que reage a aumentos no tempo de resposta do sistema, de modo a mantê-lo dentro de um limite. Para isso, é utilizado um sistema de malha fechada com um controlador PID, que reage às variações no tempo de resposta do sistema. Para avaliação da proposta, serão utilizadas cargas de trabalho sintéticas que foram parametrizadas a partir da caracterização de requisições de um sistema *web* de uso massivo.

3.1 Ferramentas Utilizadas para Implementação da Solução

3.1.1 Docker

O *Docker* começou como projeto da empresa de PaaS (*Platform as a Service*) dotCloud em 2013 [27], propondo ser um integrador e facilitador para adoção da tecnologia de *containers* em produção e em larga escala. São utilizadas as tecnologias de *cgroups* e *namespaces*, descritas na Seção 2.3.3, para isolamento e controle de recursos.

O *Docker* permite o empacotamento de aplicações com todas as suas dependências por meio da criação de imagens de *containers*. Isso permite, por exemplo, que aplicações com diferentes versões de bibliotecas e binários sejam executadas no mesmo servidor sem que hajam conflitos. Além disso, é possível encontrar, baixar e iniciar imagens de *containers* que foram criadas por outros programadores de forma muito rápida e prática.

A definição de uma imagem do *Docker* é feita por meio de *scripts* simples (similares a um *Makefile*), onde é definida uma imagem base, da qual serão herdadas todos os binários e arquivos. Após isso, são definidas as customizações, como instalação de pacotes adicionais,

cópias de arquivos, definições de pontos de montagem, entre outros. Essas atualizações são feitas de forma eficiente, pois o sistema de arquivos destas imagens se baseia em *copy-on-write*, que permite que as alterações em um *container* sejam simplesmente uma atualização diferencial da imagem anterior. Essas imagens podem ser armazenadas em repositórios (*registry*) públicos ou privados.

Os repositórios públicos do *Docker* provêm imagens de *containers* com aplicações prontas para executar, tais como bancos de dados, servidores *web* entre outros. Os *registrys* privados tem a vantagem de que as imagens são armazenadas de forma privada. Os *containers docker*, ao terem uma versão instanciada pela primeira vez em um servidor, tem que ser baixado a partir de uma *registry*. Portanto, se a *registry* privada estiver em um mesmo *datacenter* que o servidor, o *container* pode, neste caso, ser instanciado de forma mais rápida.

Portanto, o *Docker* traz funcionalidades que foram consideradas interessantes para o contexto deste trabalho, razão pela qual é utilizado nesta solução.

3.1.2 Kubernetes

Kubernetes é um sistema desenvolvido pelo Google [17] e disponibilizado para a comunidade, que visa gerenciar o ciclo de vida de *containers* nos nós de um *cluster*. O *Kubernetes* foi baseado no sistema *Borg*, usado pelo Google para execução de cargas de trabalho de produção desde meados dos anos 2000 [46]. Atualmente, o Google executa diversas cargas de trabalho em *containers*, sejam serviços *web* (ex: *Gmail* e *Google Docs*), serviços de infraestrutura ou tarefas de *MapReduce*.

Segundo [46], o *Borg* traz três benefícios principais:

- Esconde detalhes do gerenciamento de recursos e tratamento de falhas, de modo que os usuários possam focar no desenvolvimento das aplicações;
- Opera de forma confiável e disponível, permitindo o mesmo para as aplicações;
- Permite que sejam executadas cargas de trabalho, distribuídas em milhares de máquinas de forma eficiente.

Portanto, o *Kubernetes* é um gerenciador de recursos computacionais, que pode ser utilizado em larga escala na provisão e gerenciamento de sistemas distribuídos.

Neste contexto, o *Kubernetes* é um orquestrador de *containers*, sendo capaz de agendar a execução de *containers* entre os nós do *cluster*, assim como fazer o controle de admissão, balanceamento de recursos e escalabilidade dos mesmos. Também provê funcionalidades como descoberta de serviços entre os *containers*, publicação do serviço para acessos a partir de entidades fora do *cluster* e balanceamento de carga entre os *containers* [17].

A infraestrutura de um *cluster Kubernetes* é composta por nós do tipo *Master*, que controlam os nós do tipo *Workers*, e esses executam os *containers* . Todas as configurações do *cluster* ficam armazenadas em um repositório de configurações distribuído, o *Etcd*. Os *PODs* são a unidade básica com a qual o *Kubernetes* trabalha. Os *containers* são agrupados em *PODs*, e esses geralmente representam uma aplicação. Esses são criados por meio dos Replication Controllers, que são utilizados para definir *PODs* e podem ser escalados horizontalmente. Os Replication Controllers também são responsáveis por manter o número desejado de PODs ativos no *cluster*.

A seguir são definidos os componentes do *Kubernetes*:

- Servidor *Master*: Controla os nós *workers* e fornece ao administrador as ferramentas necessárias para gerenciamento do cluster.
- *Etcd*: É o repositório de configurações distribuído (banco chave-valor), sendo acessado por todos os nós do *cluster*, que buscam e gravam configurações para descoberta de serviços e informações de estado do *cluster*.
- *Controller Manager Server*: Controla as tarefas de replicação do *cluster*, salvando e lendo informações no *etcd*.
- *Scheduler Server*: Designa cargas de trabalho a específicos nós do *cluster*, considerando a utilização de recursos em cada nó, de forma a fazer uma distribuição justa.
- *Kubelet Service*: Utilizado pelos nós *workers* para se comunicar com os serviços do *cluster* (*maste*r e *etcd*). O serviço *kubelet* recebe comandos do *master* e a partir daí assume a responsabilidade de manter o estado desejado, até a próxima instrução.
- *Proxy Service*: Tem a função de encaminhar as requisições para os PODs, dentro de cada servidor, fazendo balanceamento de carga.
- *PODs*: É a unidade básica com a qual o *Kubernetes* trabalha. Os *containers* são agrupados em *PODs* e esses geralmente representam uma aplicação. Isso tem a vantagem de permitir que se possa garantir que *containers* executem juntos no mesmo nó quando desejado.
- *Services*: Permite que os serviços disponibilizados pelos *PODs* sejam acessíveis externamente ao *cluster*.
- *Replication Controllers*: É utilizado para definir PODs que podem ser escalados horizontalmente. É também responsável por manter o número desejado de PODs ativos no *cluster*.

- *Labels*: É utilizado para marcar as cargas de trabalho como um grupo. Isso ajuda em tarefas como a descoberta de instâncias e roteamento de requisições.

O *Kubernetes* possui uma ferramenta nativa de auto escalabilidade chamada de *Horizontal Pod Autoscaler* (HPA). Essa ferramenta trabalha escalando o ambiente a partir de limiares de consumo de CPU [19].

3.1.3 Flannel

Para facilitar a descoberta de serviços dentro de um *cluster*, o *Kubernetes* faz mapeamento de portas e aloca um endereço IP único para cada POD [49]. Para isso, cada *host* tem uma subrede alocada para seu uso exclusivo. Muitas vezes não é possível a alocação de uma subrede inteira para cada um dos *hosts* de um *cluster*. Para estes casos, o *flannel* pode ser usado, pois cria uma rede *overlay* que provisiona uma subrede completa para cada servidor.

Uma rede *overlay* é primeiramente configurada com uma faixa de IPs e tamanho da subrede para cada *host*. Por exemplo, uma rede *overlay* poderia ser configurada para usar 192.168.0.0/16, e cada *host* receberia uma rede /24. Dessa forma, o máquina 1 poderia receber a subrede 192.168.1.0/24 e a máquina 2 poderia receber a subrede 192.168.2.0/24. Para realizar o mapeamento entre as subredes alocadas e os IPs reais das máquinas, o *flannel* armazena as configurações em um banco *Etcd*. Para o encaminhamento do tráfego, o *flannel* usa UDP para encapsulamento dos datagramas IP para transmissão para os *hosts* remotos.

Portanto, o flannel facilita a configuração de um *cluster Kubernetes*, na media que abstrai as configurações necessárias para a comunicação entre os PODs em cada um dos nós do *Kubernetes*.

3.1.4 Apache Spark, Flume, HAproxy, Redis e Cairos

O *Apache Spark* é uma ferramenta de processamento distribuído, ideal para processamento de grandes bases de dados. Foi desenvolvido pela AMPLab (UC Berkeley) e realiza processamento de dados em memória. Sua estrutura básica de abstração são os *RDDs (Resilient Distributed DataSets)*, que são coleções de elementos que podem sofrer operações em paralelo, sendo possível a geração de novos RDDs a partir de transformações como *map, reduce, filter e join* em *RDDs* [42]. Essa ferramenta foi escolhida para integrar a solução, pois com ela é possível realizar processamento de grandes bases em formato de texto, de forma escalável.

O *Apache Spark* oferece uma API chamada *Spark Streaming*, que permite o processamento em tempo real de dados, por meio da criação de estruturas chamadas *DStream*

(*discretized stream*), que são representados como sequências de *RDDs*. A criação dos *DStreams* é feita por meio da classe *StreamingContext*, onde se configura a duração de cada janela de *DStreams* [43].

O *Flume* é um serviço que agrega, coleta e move grandes volumes de fluxo de dados. Para seu funcionamento é criada uma fonte que recebe os dados de interesse. Essa fonte (*Source*) é conectada a um canal (*Channel*), por onde os dados trafegarão em direção a um nó sorvedouro (*Sink*) [13].

O *HAproxy* é um balanceador de carga, que pode nas camada 4 ou 7 do modelo OSI, terminador SSL, *proxy* reverso entre outros [45]. Atualmente, esse é o balanceador utilizado por *sites* como *Reddit, Stack Overflow/Server Fault, Instagram*, entre outros, além de ter sido escolhido como balanceador da nuvem da Red Hat (*OpenShift*).

O Redis é uma ferramenta capaz de armazenar estrutura de dados em memória, podendo ser usado como banco de dados, *cache* ou gerenciador de filas de mensagens. Ele suporta estruturas de dados como *strings*, *hashes*, listas, listas ordenadas, conjuntos, índices geoespaciais entre outros [36].

O *Kairos* é uma biblioteca para *python* que permite o armazenamento de dados em forma de série temporal em bancos *Redis, Mongo, SQL e Cassandra* [23]. Essa biblioteca provê facilidades para o armazenamento das séries, como configuração do número de entradas a serem mantidas no banco e tamanho da unidade mínima de tempo de interesse da série. Além disso, essa biblioteca também permite o cálculo de parâmetros estatísticos da série, utilizando tamanho de janelas configuráveis de tempo.

Na Subseção 3.1.6 será descrito como essas ferramentas se integram para prover as funcionalidades desejadas da solução.

3.1.5 Arquitetura da Solução

A arquitetura proposta, chamada de PAS (PID *based Autoscaler*) é apresentada na Figura 3.1 e opera com base na seguinte sequência:

1. É estabelecido um limiar (*setpoint*) de tempo médio de resposta desejado para as requisições. O monitor de requisições recebe o tempo de resposta das requisições que chegam no balanceador;
2. O monitor de requisições envia o tempo médio de resposta (ex: Média dos últimos 200 segundos) para o dimensionador PID, que calcula o número de *containers* necessários para atingir ou permanecer no *setpoint*.
3. O Dimensionador PID executa o Algoritmo 2 e informa o número desejado de *containers* ao *Kubernetes*.

4. O *Kubernetes* cria ou remove novos *containers*, além de garantir que o ambiente permanecerá com o número desejado de *containers* até a próxima rodada do algoritmo (depois de 10 segundos). Esse valor de 10 segundos foi definido pois verificou-se que é suficiente para que o *Kubernetes* inicie novos *containers* e esses passem a responder as requisições no *cluster* de balanceamento.

Algoritmo 2: PAS

Entrada: Tempo médio de resposta do *cluster*, Número atual de *containers*

Saída: Número desejado de *Containers*

início

Leia o limiar de tempo médio de resposta desejado para as requisições: $t\_ms\_desejado$

Leia o número atual de *containers*: $n\_containers\_atual$

Leia tempo de resposta médio do cluster em ms: t_ms_atual

Calcule o erro: $e(t) = t\_ms\_desejado$ - t_ms_atual

Calcule a saída o controlador PID:

$$u(t) = K_p e(t) + K_i \int_0^t e(t)\delta t + K_d \frac{d}{dt} e(t) \tag{3.1}$$

$n\_desejado\_containers$=$n\_containers\_atual + u(t)$

fim

retorna $n\_desejado\_containers$

3.1.6 Descrição dos Componentes e Operação da solução

Para viabilizar a operação do algoritmo que trabalha com dados dinâmicos, recebidos em tempo real, utiliza-se o fluxo de processamento mostrado na Figura 3.2 que descreve o fluxo de trabalho entre todas as ferramentas integradas na arquitetura da Figura 3.1.

Nesta seção serão detalhados os componentes e como cada um foi integrado na solução dentro do fluxo apresentado.

Balanceador de carga

O *Haproxy* atua como balanceador de carga da solução. Seus *logs* são enviados ao *Flume*, que agrega os dados em um canal de memória e os envia ao *Spark Streaming*, disponibilizando, desta forma, as informações necessárias para operação do algoritmo de auto escalabilidade.

O *Haproxy* atua balanceando as requisições no modo *round robin* para os servidores *Docker/Kubernetes*, que hospedam os *containers*. Quando o nó *Docker/Kubernetes* recebe

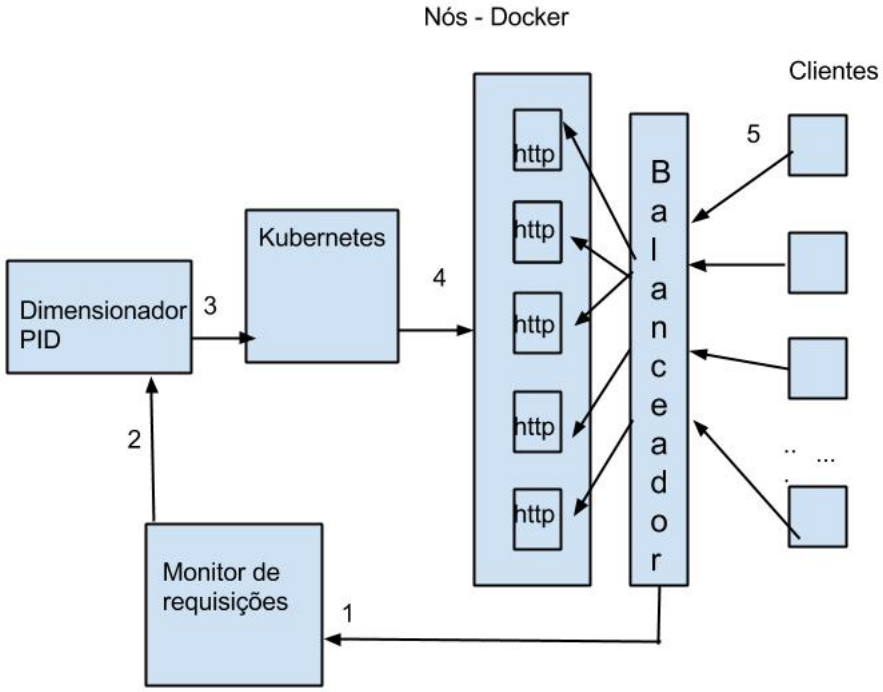

Figura 3.1: Arquitetura da Solução

a requisição, o serviço *Kubernetes proxy* é responsável por balancear a carga entre os *containers* de cada um dos nós *Docker/Kubernetes*.

Portanto, acontecem dois níveis de balanceamento, um entre os nós *Docker/Kubernetes*, em que o responsável pelo balanceamento é o *Haproxy* e outro internamente dentro do nó *Docker/Kubernetes*, onde o serviço *Kubernetes proxy* é o responsável pelo balanceamento.

Monitor de Requisições e Dimensionador PID

Para a disponibilização dos dados utilizados pelo dimensionador, o *Flume* realiza o envio, em tempo real, dos logs de acesso do balanceador carga, por meio da criação de uma fonte do tipo *Syslog*, que recebe os *logs* do balanceador de carga e um canal de memória que carrega os dados da fonte em memória. A partir daí, o *Spark* consome este fluxo de dados por meio de um *Sink* do tipo *Avro*, que trafega pela rede. O *Spark Streaming* é utilizado para processar os *logs* do balanceador de carga, coletando o tempo de resposta de cada

uma das requisições que o sistema atende, em tempo real. Essa informação do tempo de resposta é armazenada em formato de série temporal no servidor *Redis*, para uso do dimensionador que executa o algoritmo de auto escalabilidade proposto neste artigo.

O log do *HAproxy* apresenta o seguinte formato:

May 18 06:24:25 10.125.7.229 haproxy[1078]: 10.125.8.252:43839 [18/May/2016:06:24:24.988] cherrypy cherrypy/10.125.7.227 0/0/2/26/28 200 169 - - —- 1/1/1/0/0 0/0 "GET /generate HTTP/1.0"

Nesta linha, pode-se observar informações como o tempo de espera na fila do servidor de aplicação, método e código da resposta *http*, entre outras. A parte com *0/0/2/26/28* contém as informações : *Tq '/' Tw '/' Tc '/' Tr '/' Tt*, onde O *Tr* é o tempo em milissegundos que o balanceador demora para receber uma resposta completa de uma requisição *http* ao servidor [45]. Portanto, esse valor representa o tempo total de processamento da requisição pelo *container*.

No contexto deste trabalho, o *SparkStreaming* processa as entradas do *log*, separa o campo *Tr* e o armazena no banco *Redis*, no formato de uma série temporal. O *SparkStreaming* também é responsável por converter o formato da data de cada linha para o número de segundos desde a hora zero de cada dia, para suportar a criação da série temporal.

O *Redis* é utilizado para armazenar a série temporal, pois consegue prover baixa latência tanto para escrita como para leitura, por manter os dados como estruturas em memória [36].

A integração da solução com o *Redis* ocorre por meio da biblioteca *Kairos*, que cria a estrutura para armazenamento da série temporal com os tempos de resposta das requisições. No caso deste trabalho são mantidos armazenados na série a cada momento os dados dos últimos 600 segundos, o que é suficiente para operação do algoritmo. O *Kairos* também permite o cálculo de parâmetros estatísticos da série utilizando tamanho de janelas configuráveis de tempo. Isso permite que seja calculada a média de tempo de resposta de uma aplicação nos últimos 200 segundos. A média dos últimos 200 segundos foi definida, com base em testes experimentais, onde foi observado que este tempo é uma quantidade suficiente para que valores muito diferentes da média não exerçam influência negativa no dimensionamento, assim como permite que o sistema seja capaz de reagir rapidamente à mudanças no padrão da média do tempo de resposta. A unidade mínima de tempo configurada nesta solução é 1 segundo. Isso permite uma flexibilidade para configuração dos tamanhos das janelas de tempo para os cálculos estatísticos, além de permitir a geração de gráficos com boa resolução de tempo para monitoramento e avaliação da solução.

Com esses dados disponíveis no Redis, o PAS apresentado no algoritmo 2 é executado e determina-se o número desejado de *containers*.

O *Kubernetes* recebe a informação do número desejado de *containers* e utiliza a ferramenta *kubectl* para ajustar o número de *containers* dentro de um *Replication Controller*.

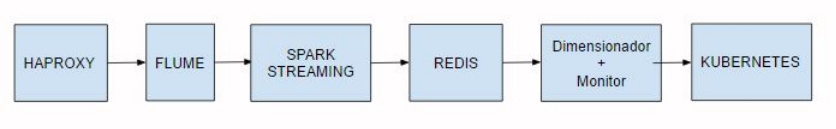

Figura 3.2: Fluxo de processamento entre o conjunto de ferramentas utilizadas

Ajuste dos parâmetros do PID

Para otimização dos parâmetros PID, o algoritmo *Coordinate Ascent* atua de modo a minimizar a soma do erro quadrático médio da série dos tempos médios de resposta em relação ao valor estabelecido no *setpoint*, durante um determinado período. Esse algoritmo está descrito em pseudo código no Algoritmo 1 e nesta subseção será descrito como o mesmo é utilizado para otimização dos parâmetros do PID (*Kp, Ki* e *Kd*).

Neste caso, são estabelecidos valores iniciais para os parâmetros *Kp, Ki* e *Kd* (ex: 0.1, 0.1 e 0.1) e um tamanho de passo inicial para cada um dos parâmetros (ex: 0.01, 0.01 e 0.01). Além disso, é estabelecido um valor inicial para o erro quadrático médio, que será chamado de erro. O erro quadrático médio é a função de pontuação do algoritmo neste caso. O erro quadrático médio utilizado nesse algoritmo é o quadrado da soma das diferenças dos tempos de resposta em relação ao *setpoint*, dentro de uma janela de tempo. Nessa implementação, a janela foi definida em 2000 segundos, pois verificou-se que este tempo era suficiente para verificação dos efeitos dos ajustes de cada um dos parâmetros do PID na função de pontuação.

A partir desse ponto, o algoritmo começa atuando em cima do *Kp*, aumentando o seu valor segundo o valor do passo inicial. Com este novo valor de *Kp*, é salva uma amostra da série e a partir dessa mudança é verificado o novo erro quadrático médio. Caso esse seja menor que o anterior, o valor de *Kp* é mantido, o tamanho do passo para o *Kp* é aumentado em 10 % e o erro é atualizado para o erro quadrático médio desta observação. Caso não seja observada a diminuição do erro, ao invés de se somar, se subtrai o passo inicial ao *Kp* e verifica-se o novo erro, conforme descrito anteriormente. Caso tenha havido a diminuição do erro, o valor de *Kp* é mantido, o tamanho do passo para o *Kp* é aumentado em 10 % e o erro é atualizado para o erro quadrático médio desta observação. Caso não haja melhora, o valor do *Kp* é retornado para o seu valor original e o tamanho do passo é diminuído em 10 %.

Esses ajustes se repetem sucessivamente em um laço, com modificações também no *Ki* e *Kd*, até que a soma dos passos de atualização de cada um dos parâmetros seja menor que um valor mínimo pré configurado (ex: 0.0000001).

3.2 Conclusão deste Capítulo

Este capítulo apresentou os componentes e como os mesmos serão integrados de modo a prover a solução proposta desejada deste trabalho. Foram detalhados os paradigmas de cada tecnologia a ser utilizada e como as mesmas foram integradas de forma inovadora em uma arquitetura de nuvem elástica baseada no tempo de resposta das aplicações. Foi descrito também a integração do controlador PID à arquitetura para prover valores ótimos de número de *containers* a serem utilizados para diferentes cargas de trabalho. O capítulo seguinte apresentará a avaliação experimental da solução proposta.

Capítulo 4

Resultados Experimentais

Para avaliar a proposta da arquitetura apresentada no capítulo anterior, neste capítulo são apresentados um conjunto de cenários experimentais e os seus resultados, os quais serão comparados com a proposta *HPA* [19].

4.1 Ambiente

Para avaliação da solução foi configurado um *cluster Kubernetes* v1.4.1 no sistema operacional *CoreOS* (899.6.0 (2016-02-02)), virtualizado em *VMWare ESXi* 5.5.0. Este ambiente foi configurado para validação da solução. Em um ambiente de produção o sistema *CoreOS*, poderia ser instalado diretamente em máquinas físicas, eliminando a camada de virtualização.

O *cluster* foi constituído com os seguintes componentes: 1 nó *master* (4 vCPUs, 6 GB de RAM), 1 nó *etcd* (4 vCPUs, 6 GB de RAM) e 3 nós *workers* (4 vCPUs, 6 GB de RAM). Máquinas virtuais (*VMWare ESXi* 5.5.0) foram instaladas com o sistema operacional Ubuntu 14.04.3 LTS, com as seguintes configurações e ferramentas: 1 nó *haproxy* 1.5.4 (4 vCPUs, 4G GB de RAM), 1 nó *Spark* 1.5.2 mais *Redis* 2.8.4 (2 vCPU, 10 GB de RAM) e 1 nó *Flume 1.7.0* mais o dimensionador (2 vCPU, 4 GB de RAM).

4.1.1 Carga de Trabalho para os Cenários 1 e 2

Para avaliação dos cenários 1 e 2 foi gerada uma imagem de *container* que executa o servidor web *cherrypy* 5.1.0.

A arquitetura elaborada para este serviço pode ser vista na Figura 4.1. O serviço provido é acessado por meio do balanceador de carga *HAproxy* que distribui as requisições para as instâncias do *cherrypy* que são providas por meio de *containers* em um *cluster Kubernetes*.

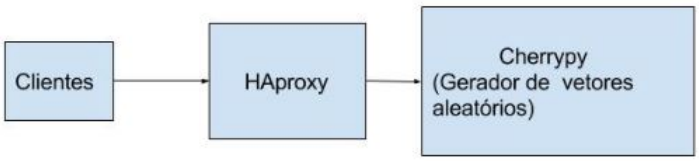

Figura 4.1: Arquitetura ambiente *Cherrypy*

A publicação do serviço no *Kubernetes* foi feita por meio da criação de um *Replication Controller*, que cria os *containers* com o servidor *web cherrypy* e a configuração de um serviço do tipo *NodePort*. O serviço *NodePort* expõe uma determinada porta em cada um dos nós *workers*, que ao receberem uma requisição, a mesma é direcionada para um dos *containers* que provêm o serviço em cada um dos nós. Este direcionamento da requisição da porta exposta no nó *worker* para um dos *containers* é realizado pelo serviço *kube-proxy* do *Kubernetes*. O serviço *kube-proxy*, escolhe um dos *containers* dentro de um *Replication Controller* por meio da política *Round-robin*.

O balanceador *Haproxy* foi configurado para balancear as requisições entre os nós *Docker/Kubernetes* usando os endereços IP dos nós e as portas publicadas pelo serviço do tipo *NodePort*. Cada *container* teve seus recursos de processamento e memória limitados a 18 MB de memória RAM e 24 milicores de CPU.

Este cenário foi implementado visando prover um serviço simples, mas que pudesse avaliar a viabilidade do algoritmo de auto elasticidade proposto.

4.1.2 Carga de Trabalho para os Cenários 3, 4 e 5

Os cenários 3, 4 e 5 utilizam a carga de trabalho Rubis [37], que é modelado para ser um clone do ebay (www.ebay.com). O RUBiS implementa as funcionalidades básicas do ebay: registro de produtos, venda, lances de leilão, navegação em produtos por região (Estados Unidos), e categorias. A versão instalada do RUBiS foi a 1.4.3 obtida em [44].

Nos testes foi utilizado a versão PHP do RUBiS, com banco de dados MySQL versão 5.5. O MySQL foi instalado em uma máquina virtual com Ubuntu 14.04.1 (16 vCPU e 4 GB de RAM). O MySQL foi configurado para permitir *cache* das tabelas do RUBiS. Essas configurações altas de recursos da máquina virtual do MySQL foram realizadas para garantir que não haveriam gargalos no acesso ao banco de dados pela aplicação, já que o ob-

jetivo dos testes é avaliar a auto escalabilidade do serviço *Web*. O banco de dados foi populado a partir do *dump* obtido em http://download.forge.ow2.org/rubis/rubis_dump.sql.gz.

A arquitetura deste ambiente está esquematizada na Figura 4.2. Os clientes acessam o serviço Rubis (provido por *containers* no *cluster Kubernetes*), passando pelo balanceador de carga, *HAproxy*. O Rubis faz acesso aos dados persistentes no banco MySQL, sendo que os dados recentemente acessados são providos pelo serviço de *cache* do MySQL.

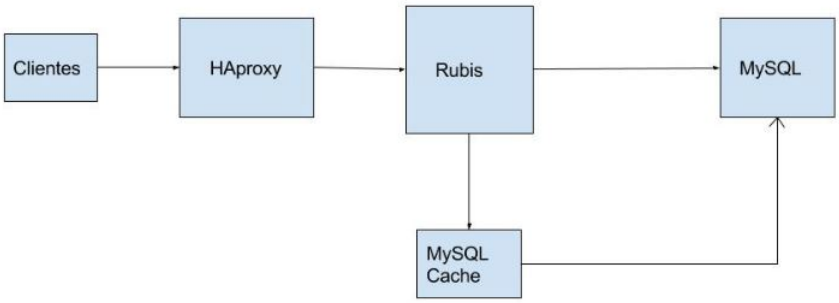

Figura 4.2: Arquitetura ambiente *Rubis*

Assim como na configuração descrita para os cenários 1 e 2, a publicação do serviço no *Kubernetes* foi feita por meio da criação de um *Replication Controller* e a configuração de um serviço do tipo *NodePort*. O balanceador *Haproxy* foi configurado para balancear as requisições entre os nós *Docker/Kubernetes* usando os endereços IP dos nós e as portas publicadas pelo serviço do tipo *NodePort*. Nos cenários 2, 3 e 4 cada *container* teve seus recursos de processamento e memória limitados a 500 MB de memória RAM e 160 milicores de CPU. Para o comparativo entre o HPA e o PAS (cenário 5), os recursos de processamento dos *containers* foram limitados a 300 milicores e 300 MB de memória RAM. Estes limites para o comparativo, foram estabelecidos verificando que estes recursos são suficientes para suportar a realização dos testes em diversas escalas de intensidade. Um ponto a ser observado é que todos os testes foram realizados, sem aquecimento dos algoritmos. Ou seja, as cargas não foram sendo aumentadas de forma forma gradual até se chegar a intensidade final.

4.1.3 Gerador de Carga de Trabalho

Para geração da carga com um perfil de requisições próximo do real, foi coletado um conjunto de acessos do portal da transparência (www.transparencia.gov.br), entre os dias 25/05/2015 a 25/06/2015. A série temporal, capturada na escala de 1 segundo representa o número de acessos naquele intervalo de tempo. A série foi caracterizada com o método

Kettani-Gubner [24]. A autossimilaridade e longa dependência da série foi confirmada com o parâmetro de Hurst H=0,87, nas escalas de 1 segundo, 100 segundos e 600 segundos.

Esta série é utilizada para gerar a cada segundo requisições simultâneas direcionadas ao endereço IP do balanceador de carga. O balanceador de carga distribui as requisições aos nós do *cluster Kubernetes*. A intensidade da carga foi ajustada em vários níveis, multiplicando a série temporal por 1, 1.5, 2, 3 e 4 e preservando o mesmo índice de autossimilaridade. Estas cargas são referidas respectivamente nos experimentos como carga_1, carga_1.5, carga_2 e carga_3 e carga_4.

A geração de carga de trabalho, descrita no parágrafo anterior, foi feita usando a ferramenta *ab (apache bench)* [2]. O *ab* é uma ferramenta que é projetada para realizar testes de carga em servidores *web*. Esta ferramenta permite que sejam configurados parâmetros para a geração da carga, como por exemplo, o número de total requisições desejadas e quantas destas requisições devem ser geradas em paralelo. No caso do gerador de tráfego criado neste trabalho, a série temporal com o número de requisições em cada segundo gerada a partir de *logs* de acessos ao Portal da Transparência [6] é utilizada para determinar quantas requisições simultâneas devem ser geradas a cada segundo do teste.

Uma amostra da série obtida utilizada para caracterizar a carga de trabalho, relativa a 2000 segundos pode ser vista na Figura 4.3 em que pode ser apreciada a alta explosividade da mesma.

Para os cenários que utilizam o *cherrypy*, foi configurada uma página *web* neste servidor, que ao ser acionada gera um vetor de tamanho aleatório entre 1.000 e 10.000 elementos. Neste caso a cada segundo, todas as requisições são direcionadas para esta página, que é atendida por um dos *containers* do *cluster*.

Para os cenários que utilizam o Rubis, os acessos simultâneos em cada segundo às páginas é dividido da seguinte forma: 10 porcento de acessos a página inicial, 10 porcento de consultas a lista de produtos com categoria aleatória e região aleatória, 40 porcento de consultas a produtos aleatórios e 40 porcento de consultas a perfis de usuários aleatórios. Essa distribuição percentual nos acessos foi definida considerando o comportamento de um usuário que acessa a página inicial do sistema, faz uma consulta por um produto, em uma determinada região geográfica, e a partir daí passa a maior parte do tempo navegando entre produtos, e verificando o perfil de outros usuários, que realizam as vendas.

4.1.4 Configuração dos Parâmetros PID de Forma Manual

Para os cenários 1, 2, 3, e parte do 5, o PID utilizou os seguintes parâmetros: Kp=0.016, Ki=0.000012 e Kd=0.096, que foram definidos após vários testes com a carga de trabalho e aplicação do método de ajuste manual (*guess and check*). Esses parâmetros foram

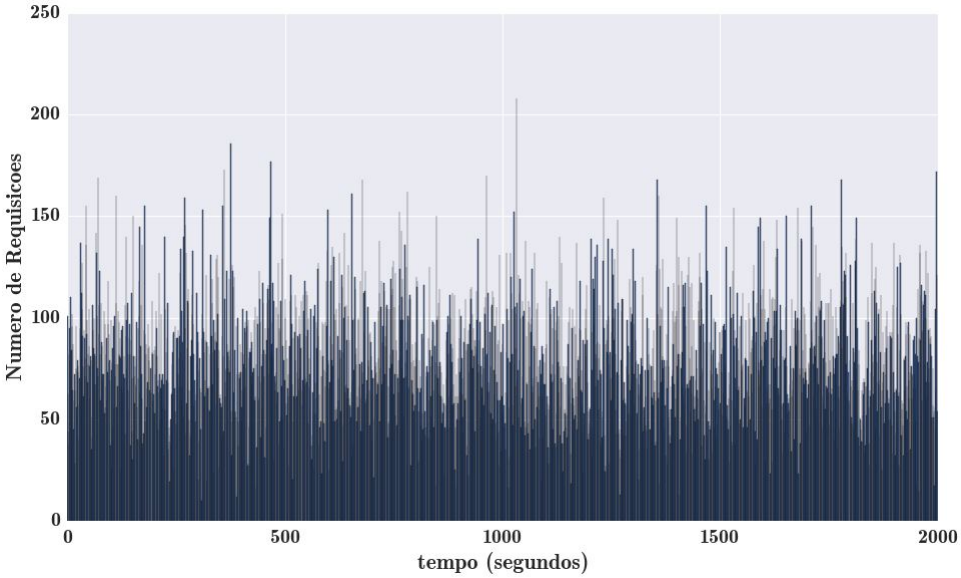

Figura 4.3: Amostra da carga utilizada nos testes, H=0,87

utilizados para todas as configurações de limiares que usaram configuração manual dos parâmetros do PID, ou seja 50 ms, 80 ms, 100 ms e 120 ms.

4.1.5 Configuração dos Parâmetros PID com *Coordinate Ascent*

Nos cenários 4 e parte do cenário 5, o algoritmo PID foi otimizado usando o algoritmo *Coordinate Ascent*. Os valores determinados por este algoritmo para cada um dos limiares pode ser visto na Tabela 4.1. Para a descoberta dos parâmetros para os limiares de 80, 100 e 120 ms do PAS, o algoritmo *Coordinate Ascent* foi executado da seguinte forma:

- Incialmente os parâmetros *Kp*, *Ki* e*Kd* foram configurados todos para 0.01;
- O passo de atualização foi configurado para 0.001;
- O limiar de convergência da soma dos passos, foi configurado para 0.000001;
- O sistema foi exposto a carga_4 e esperou-se a convergência do algoritmo para cada um dos casos.

A carga_4 foi escolhida para aplicação no sistema para a descoberta dos parâmetros, por ser uma carga de intensidade alta, o que permitiu que a subida dos tempos de resposta fossem suficientes para fazer com que os algoritmos tivessem que criar e destruir *containers* para manutenção dos limiares de tempo de resposta, permitindo assim a convergência dos parâmetros.

A ordem de grandeza dos parâmetros iniciais do *Kp*, *Ki* e*Kd*, assim como o valor do passo inicial de atualização foram determinadas a partir da constatação experimental que valores em ordens de grandeza maiores levavam o sistema a instabilidade, pois o *Coordinate Ascent* testava valores que deixavam o sistema muito reativo, o que levava a alocação de desalocação de *containers* de forma extrema, o que impedia a continuidade da busca por valores otimizados.

4.2 Cenários de Avaliação

Os cenários 1, 2, 3 e 4 foram elaborados no início dos experimentos como prova de conceito da proposta. Nestes cenários, é realizada apenas uma rodada de testes, portanto não foram gerados intervalos de confiança para os resultados obtidos. Já para o cenário 5, após a prova de conceito, foram realizados testes repetitivos com o cálculo de médias estatísticas e definição de intervalos de confiança.

4.2.1 Cenário 1

No primeiro cenário o limiar de tempo de resposta no balanceador (*setpoint*) foi estabelecido em 50 ms e foram aplicadas as cargas carga_1 e carga_1.5. A Figura 4.4 mostra o tempo de resposta do sistema ao ser submetido à carga_1. O gráfico mostra o ajuste provocado pelo PID e a estabilidade alcançada próximo do *setpoint* de 50 ms. Pode-se observar, que nos segundos iniciais do teste, o tempo de resposta sobe acima do limiar desejado, e a alocação de *containers* vai aumentando, chegando a um pico um pouco antes do segundo 200. Após este pico, o tempo de resposta começa a cair, assim como a alocação de *containers*, e o tempo de resposta se estabiliza no *setpoint*.

A Figura 4.5 mostra a alocação de *containers*. Observa-se que o número de *containers* no início do experimento é igual a 2. O sistema alocou o número necessário para que o tempo médio de resposta mostrado na Figura 4.4 atingisse o *setpoint*. No final da execução estavam alocados 26 *containers*.

A Figura 4.6 mostra o tempo médio de resposta do sistema quando submetido à carga_1.5 e o *setpoint* mantido em 50 ms. A Figura 4.7 mostra a alocação de *containers* durante este teste. Observa-se que a alocação de *containers* foi ajustada de modo a manter o tempo de resposta médio do sistema próximo ao *setpoint*. No final da execução estavam alocados 52 *containers*. Neste caso, o sistema submetido a uma carga maior, alocou mais *containers* para manter o tempo de resposta dentro do limiar definido.

Os resultados do cenário 1, indicam que o PAS pode ser adequado para promover a auto elasticidade, dentro da janela de tempo observada, permitindo a manutenção de

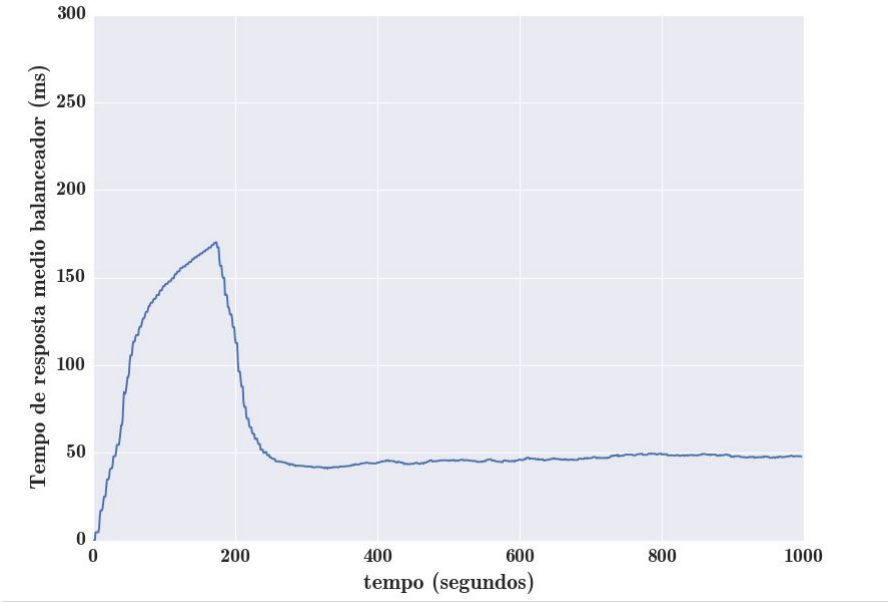

Figura 4.4: Tempo de Resposta (ms) x Tempo (s), carga carga_1

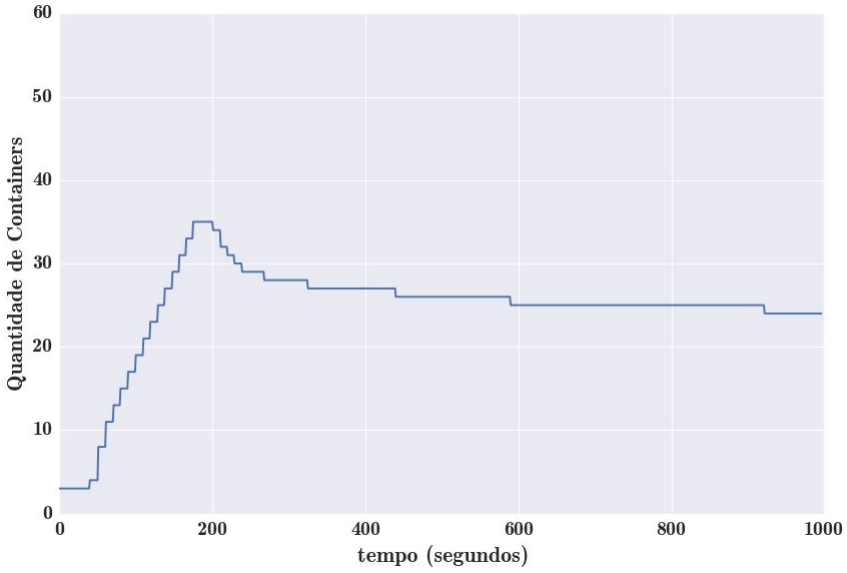

Figura 4.5: Número de *containers* x Tempo (s), carga_1, *setpoint* 50 ms

tempos de resposta da aplicação dentro de limiares pré definidos, mesmo sendo submetido a testes com intensidades de carga diferentes.

4.2.2 Cenário 2

Neste cenário foi avaliado o comportamento da proposta quando o sistema é submetido a uma carga que varia de intensidade durante o teste, diferentemente do cenário 1, onde a intensidade da carga permanece constante.

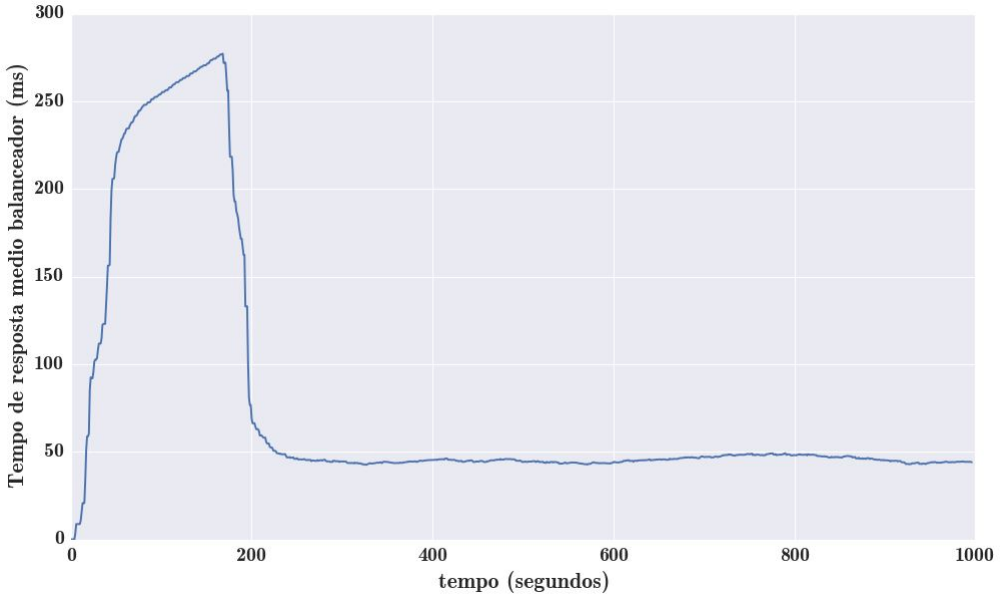

Figura 4.6: Tempo de Resposta (ms) x Tempo (s), carga carga_1.5, *setpoint* 50 ms

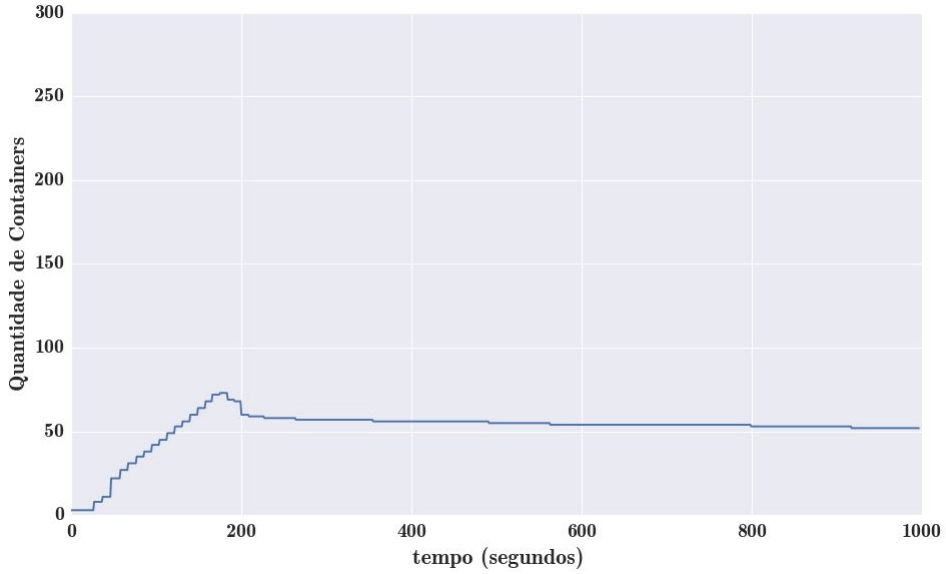

Figura 4.7: Número de *Containers* x Tempo (ms), carga carga_1.5, *setpoint* 50 ms

O *setpoint* foi definido em 50 ms e foi aplicada a carga_1 por 1000 segundos. Posteriormente foi aplicada a carga_1.5 por mais 1000 segundos (começando no segundo 1001), e por fim foi aplicada a carga_1 por mais 1000 segundos (começando no segundo 2001). O gráfico da Figura 4.9 mostra que no início do teste, haviam dois *containers* alocados. Ao se aplicar a carga_1, o tempo de resposta da aplicação foi subindo (Figura 4.8), assim como a alocação de *containers*. O sistema PAS conseguiu controlar o tempo de resposta para a carga_1, em 50 ms, alocando pouco mais de 20 *containers*.

Ainda observando-se as figuras 4.8 e 4.9, verifica-se que quando iniciou-se a aplicação da carga_1.5, o tempo de resposta começou a subir novamente, e o PAS realizou a alocação de mais *containers* de modo a manter o tempo de resposta dentro do limiar de 50 ms. No momento que a carga_1 voltou a ser aplicada, a alocação de *containers* voltou a cair.

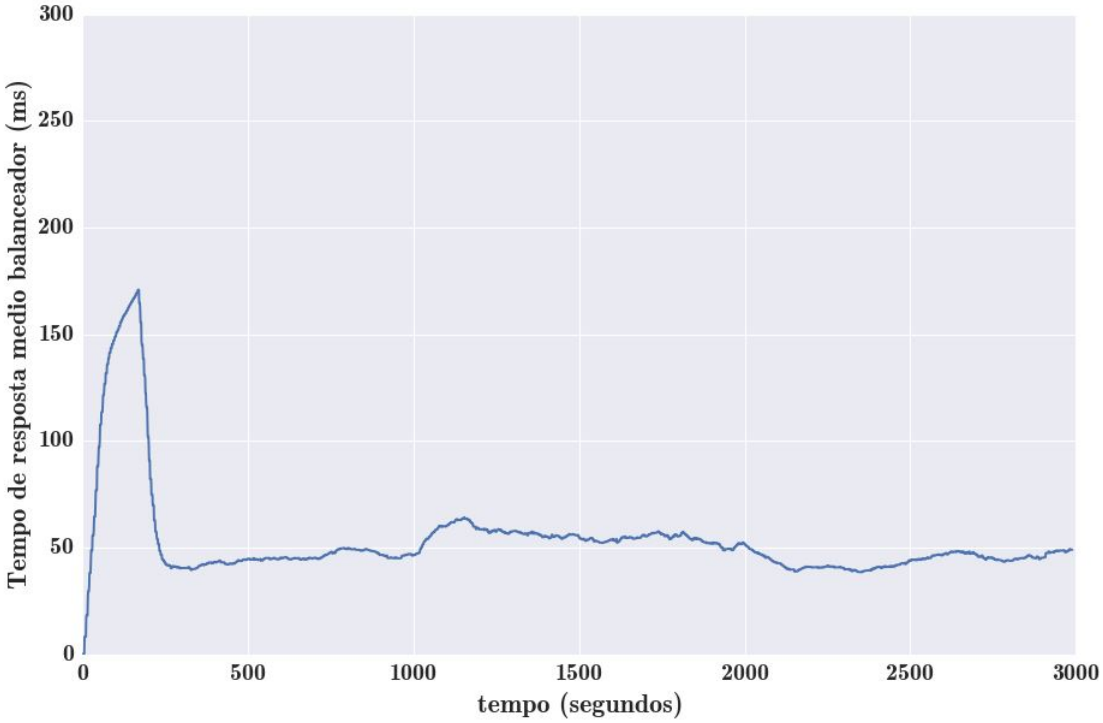

Figura 4.8: Tempo de Resposta (ms) x Tempo (s), carga carga variável, *setpoint* 50 ms

Os resultados do cenário 2, indicam que o PAS é capaz de promover ajustes na alocação de *containers*, em cenários onde ocorrem mudanças bruscas na intensidade da carga aplicada ao ambiente. Quando o sistema foi submetido a um aumento de 50 % na intensidade da carga, mais recursos foram alocados, de modo a manter o tempo de resposta da aplicação dentro do limiar definido. Quando a carga voltou a intensidade inicial, recursos foram liberados, indicando o correto funcionamento da solução na provisão de auto elasticidade.

4.2.3 Cenário 3

No cenário 3 é avaliado o comportamento do algoritmo PAS no ambiente do RUBiS. São utilizadas as cargas carga_1 e carga_1.5.

No primeiro teste o *setpoint* é ajustado para 50 ms e é aplicada a carga_1 e depois a carga_1.5. Cada carga é aplicada durante 3000 segundos. As figuras 4.10 e 4.11 demonstram a evolução do tempo de resposta e a alocação de *containers* durante o teste utilizada

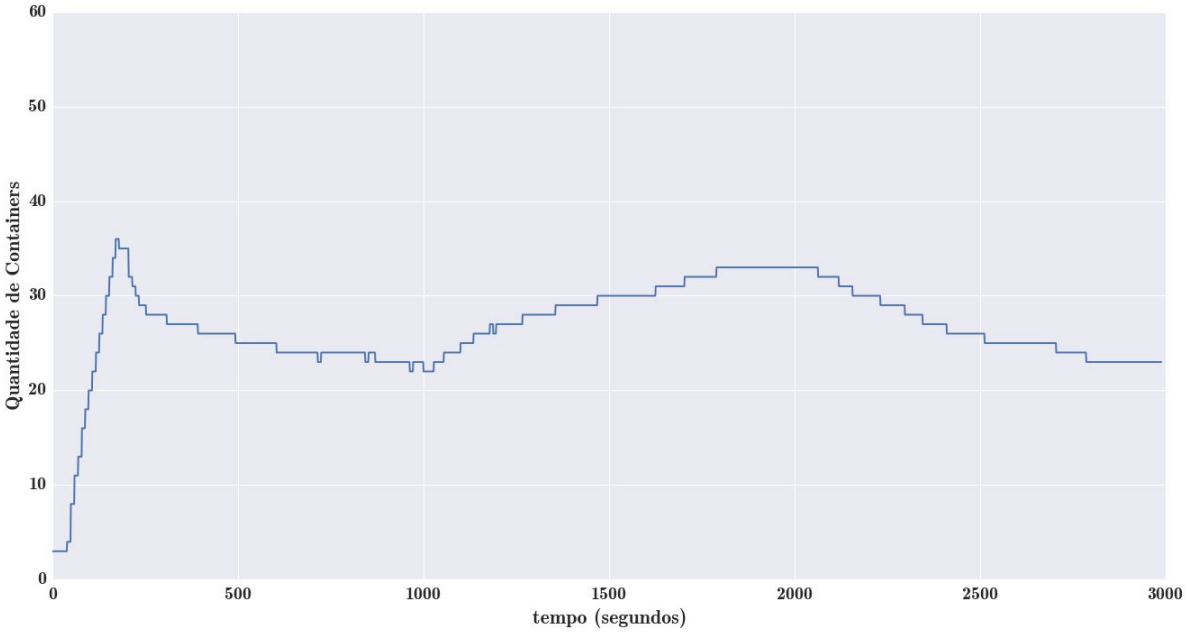

Figura 4.9: Número de *Containers* x Tempo (ms), carga carga variável, *setpoint* 50 ms

para controlar o tempo de resposta (*setpoint*=50 ms) da aplicação para a carga_1, e as figuras 4.12 e 4.13 demonstram a evolução do tempo de resposta e a alocação de *containers* durante o teste utilizada para controlar o tempo de resposta (*setpoint*=50 ms) da aplicação para a carga_1.5.

No segundo teste o *setpoint* é ajustado para 80 ms e são aplicadas as cargas carga_1 e carga_1.5. As figuras 4.14 e 4.15 demonstram a evolução do tempo de resposta e a alocação de *containers* durante o teste utilizada para controlar o tempo de resposta (*setpoint*=80 ms) da aplicação para a carga_1, e as figuras 4.16 e 4.17 demonstram a evolução do tempo de resposta e a alocação de *containers* durante o teste utilizada para controlar o tempo de resposta (*setpoint*=80 ms) da aplicação para a carga_1.5. Observa-se que neste cenário ocorre uma maior oscilação do tempo de resposta médio em torno do *setpoint*, e também oscilação da alocação do número de containers. Isto indica que os parâmetros do PID (*Kp, Ki, e Kd*) deveriam ser revistos para este novo *setpoint* de modo a otimizar ainda mais a alocação de *containers* para este caso. Entretanto, esta oscilação não prejudicaria a percepção do usuário em relação ao tempo de resposta do sistema, já que a oscilação máxima ocorrida neste teste ficou próxima a 20 ms. Apesar disto, no próximo cenário, os parâmetros do PID foram otimizados usando o algoritmo *Coodinate Ascent*, de modo a tentar minimizar este fenômeno.

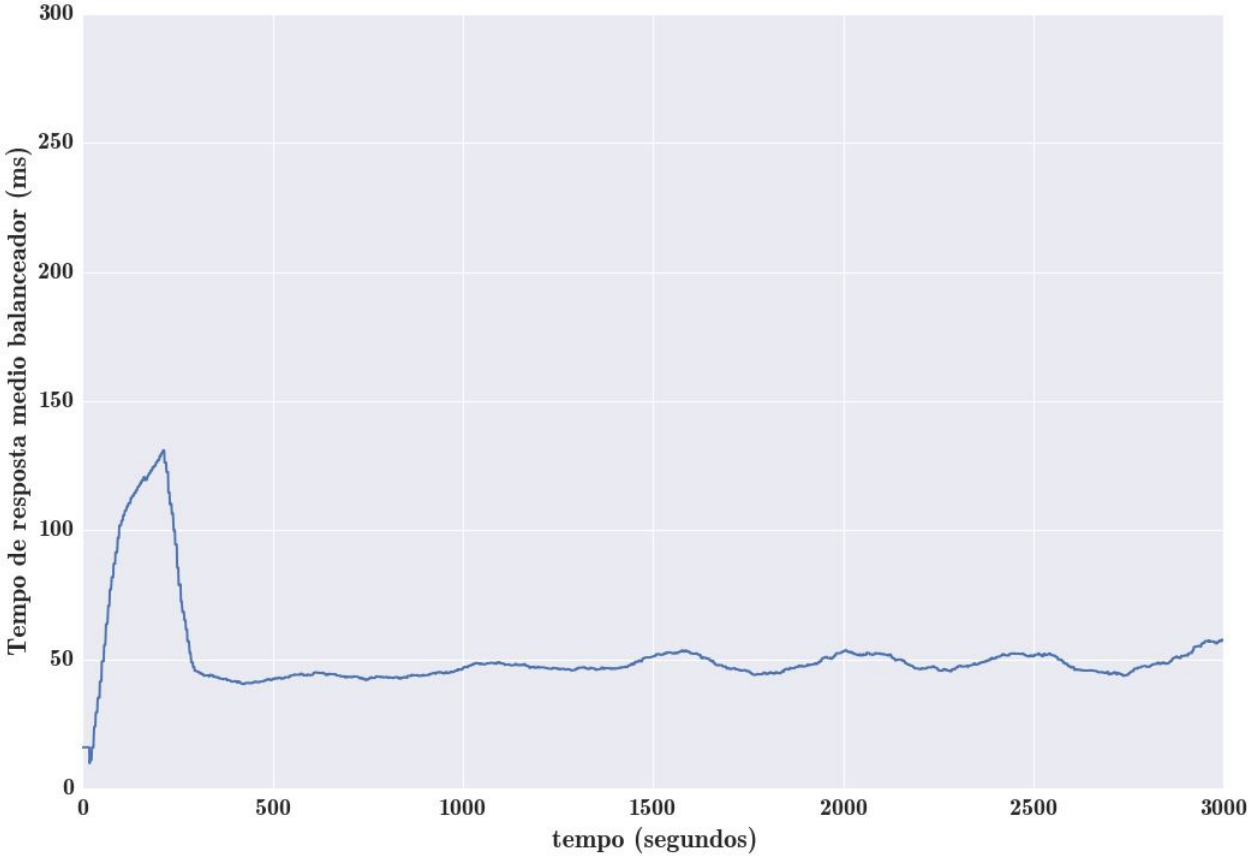

Figura 4.10: Tempo de Resposta Rubis (ms) x Tempo (s), carga 1, *setpoint* 50 ms

4.2.4 Cenário 4

Neste cenário realizou-se testes modificando o *setpoint* para 80 ms e realizando testes com a carga carga_4. Para esses testes, cada *container* foi limitado a 300 MB de memória RAM e 300 millicores de processamento. Os parâmetros otimizaram pelo *Coordinate Ascent*, foram Kp=0.00121, Ki=0.0000505 e Kd=0.00112. A metodologia utilizada para execução desta otimização é a que foi detalhada na Sessão 4.1.5. Os parâmetros estabelecidos pelo método de ajuste manual foram: Kp=0.001, Ki=0.000012 e Kd=0.096.

Os resultados para estes testes podem ser vistos nas figuras 4.18, 4.19, 4.20 e 4.21. Na Figura 4.18 é possível observar que o controle do tempo de resposta utilizando os parâmetros otimizados pelo algoritmo *Coordinate Ascent* promoveram uma curva de tempo de resposta com menos oscilações do que a curva de tempo de resposta obtida com os parâmetros configurados de forma manual (Figura 4.20). Na Figura 4.19 observa-se também maior estabilidade na alocação de *containers* do que em 4.21.

Os resultados deste cenário mostram um controle do tempo de resposta otimizado

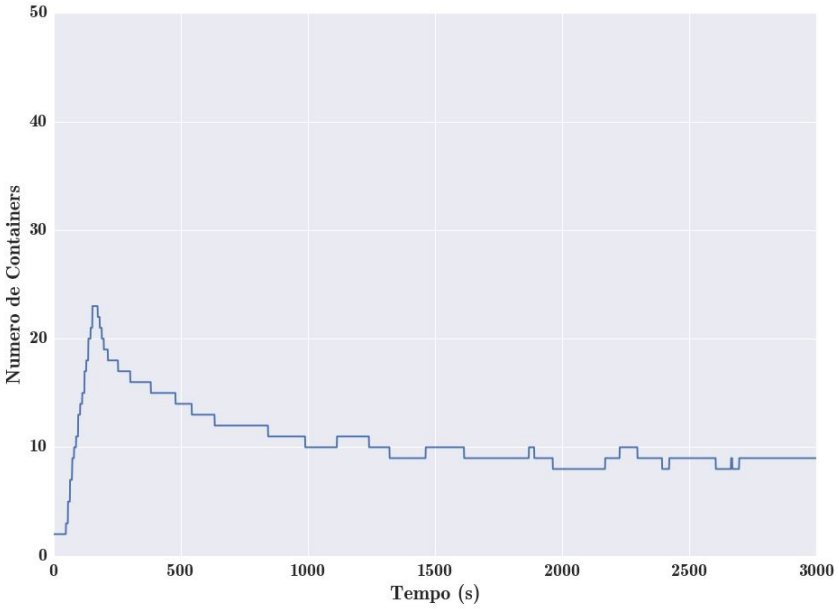

Figura 4.11: Número de *Containers* Rubis x Tempo (ms), carga 1, *setpoint* 50 ms

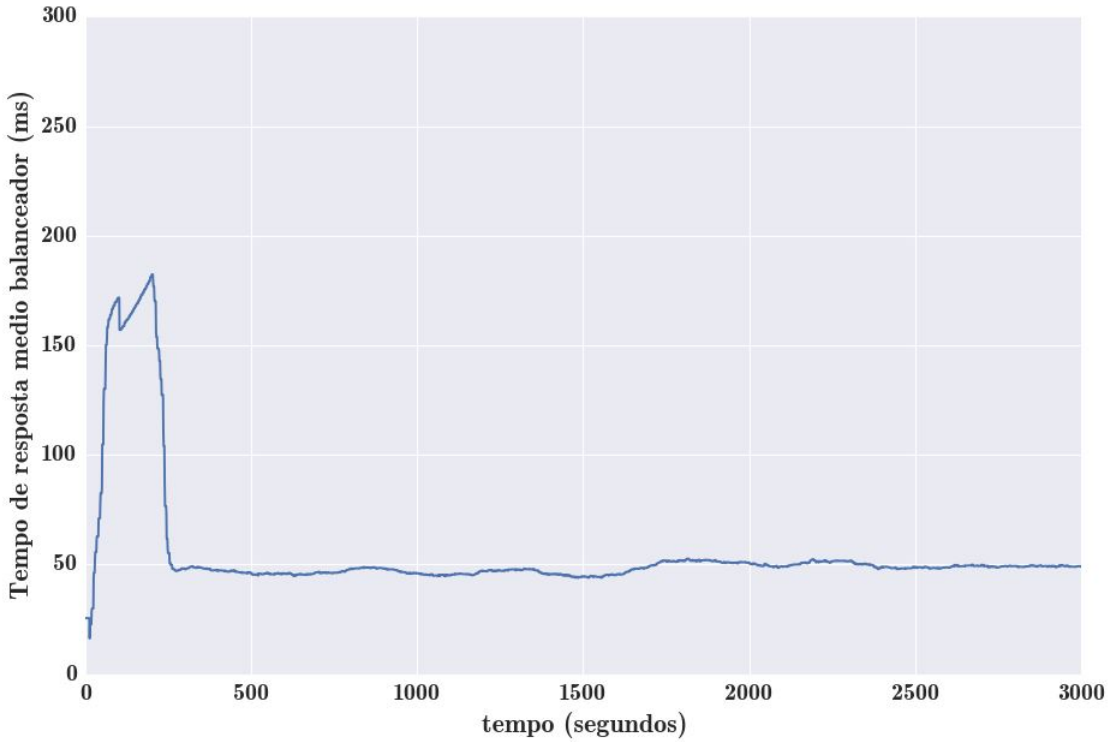

Figura 4.12: Tempo de Resposta Rubis (ms) x Tempo (s), carga 1.5, *setpoint* 50 ms

mediante o *Coordinate Ascent*. Esse melhor controle permite que a atividade de criação e destruição de *containers* permaneça mais estável do que o caso em que os ajustes de

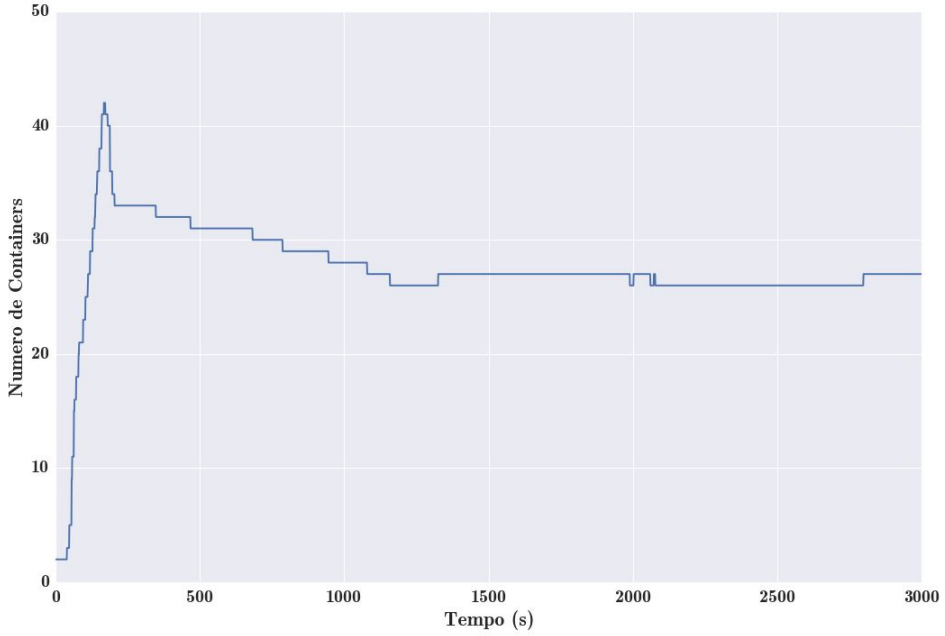

Figura 4.13: Número de *Containers* Rubis x Tempo (ms), carga 1.5, *setpoint* 50 ms

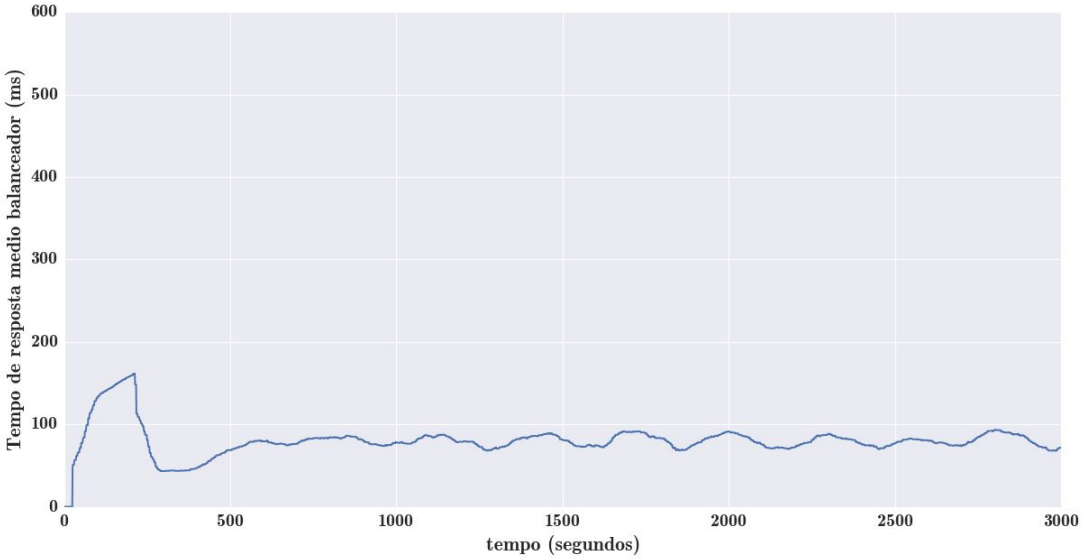

Figura 4.14: Tempo de Resposta Rubis (ms) x Tempo (s), carga 1, *setpoint* 80 ms

parâmetros foram feitos de forma manual, para este limiar. Isso demonstra que a otimização de parâmetros utilizando o *Coordinate Ascent* apresentou potencial para deixar o sistema com uma maior robustez.

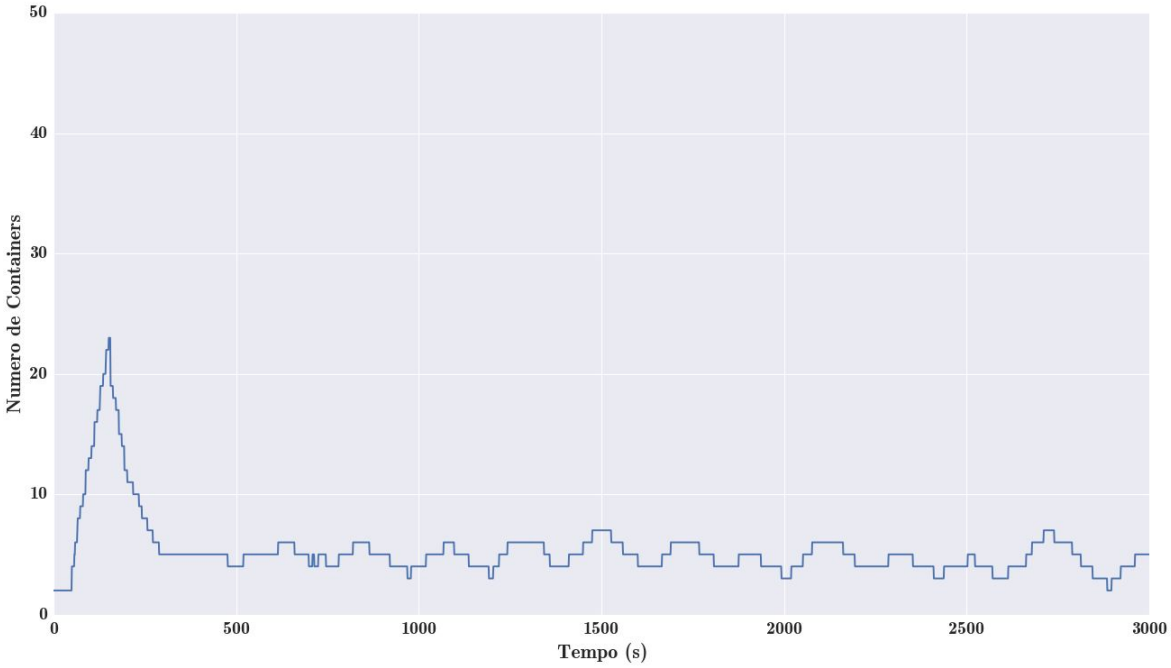

Figura 4.15: Número de *Containers* Rubis x Tempo (ms), carga 1, *setpoint* 80 ms

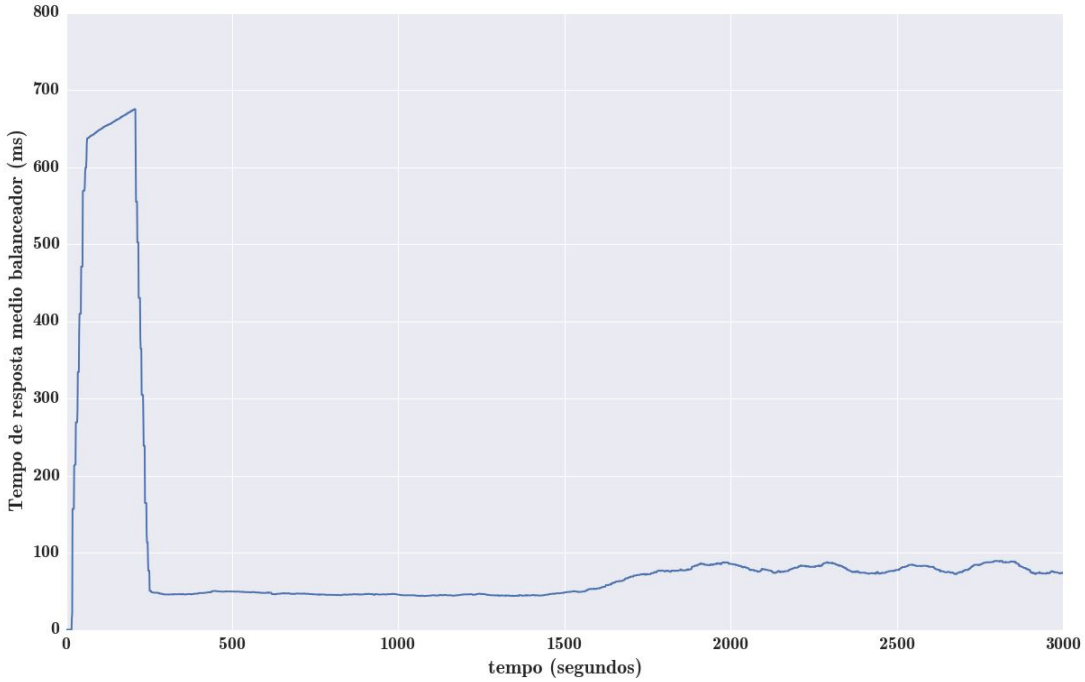

Figura 4.16: Tempo de Resposta Rubis (ms) x Tempo (s), carga 1.5, *setpoint* 80 ms

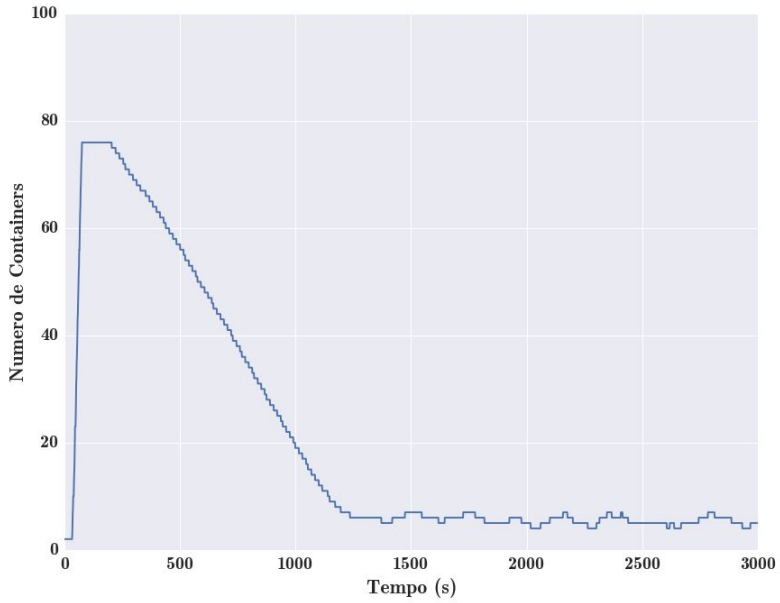

Figura 4.17: Número de *Containers* Rubis x Tempo (ms), carga 1.5, *setpoint* 80 ms

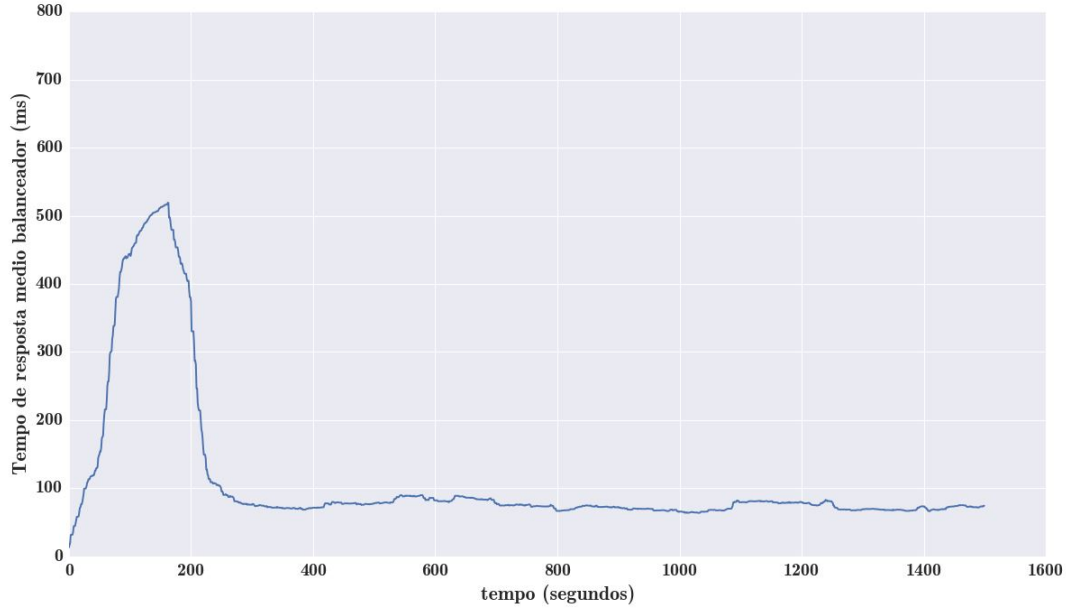

Figura 4.18: Tempo de Resposta Rubis (ms) x Tempo (s), carga 4, *setpoint* 80 ms, *Coordinate Ascent*

4.2.5 Cenário 5

O objetivo deste cenário é generalizar estatisticamente as provas de conceito obtidas nos cenários anteriores, além de comparar diversas configurações do algoritmo PAS e do HPA.

Para tal, foram realizadas várias rodadas de testes com cargas em diversos níveis de intensidade e os algoritmos HPA e PAS receberam diversas configurações. O objetivo

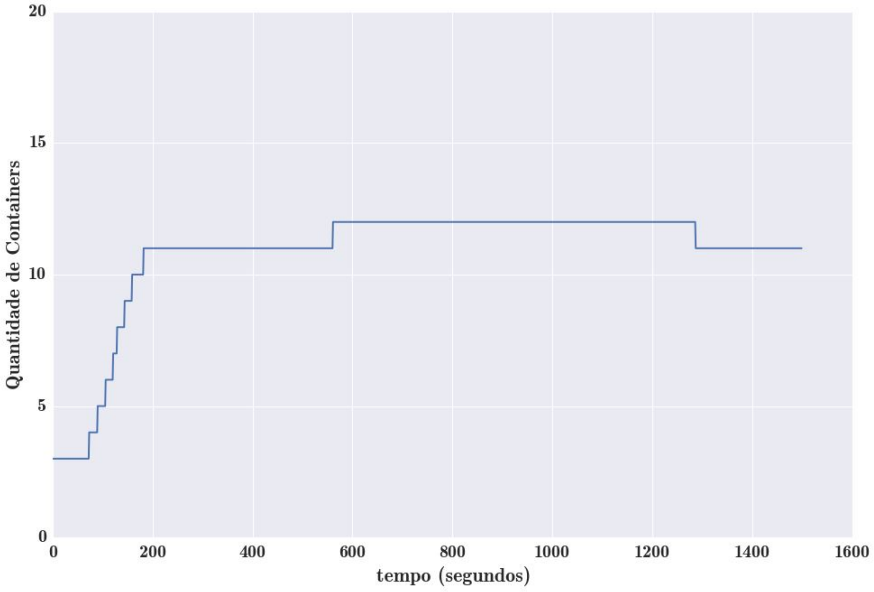

Figura 4.19: Número de *Containers* Rubis x Tempo (ms), carga 4, *setpoint* 80 ms, *Coordinate Ascent*

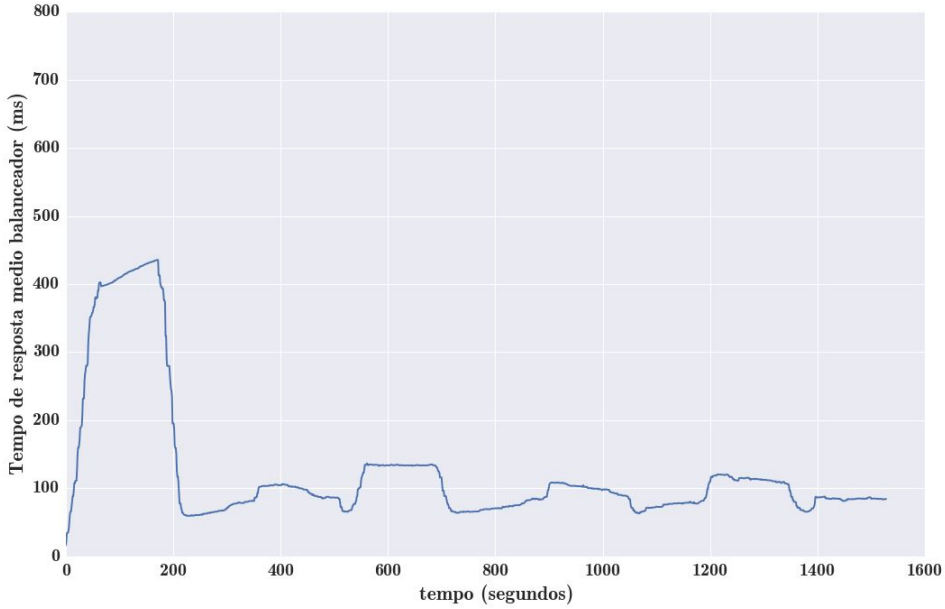

Figura 4.20: Tempo de Resposta Rubis (ms) x Tempo (s), carga 4, *setpoint* 80 ms, sem *Coordinate Ascent*

destes testes é comparar o comportamento de cada um dos algoritmos em cada uma das configurações e para cada um dos níveis de intensidade de carga.

Em todos os testes foram utilizados o ambiente de testes com o Rubis, descrito na Seção 4.1.2. Os limites de alocação de recursos para esses testes foram configurados como 300 milicores de processamento e 300 MB de memória RAM. Essas configurações foram

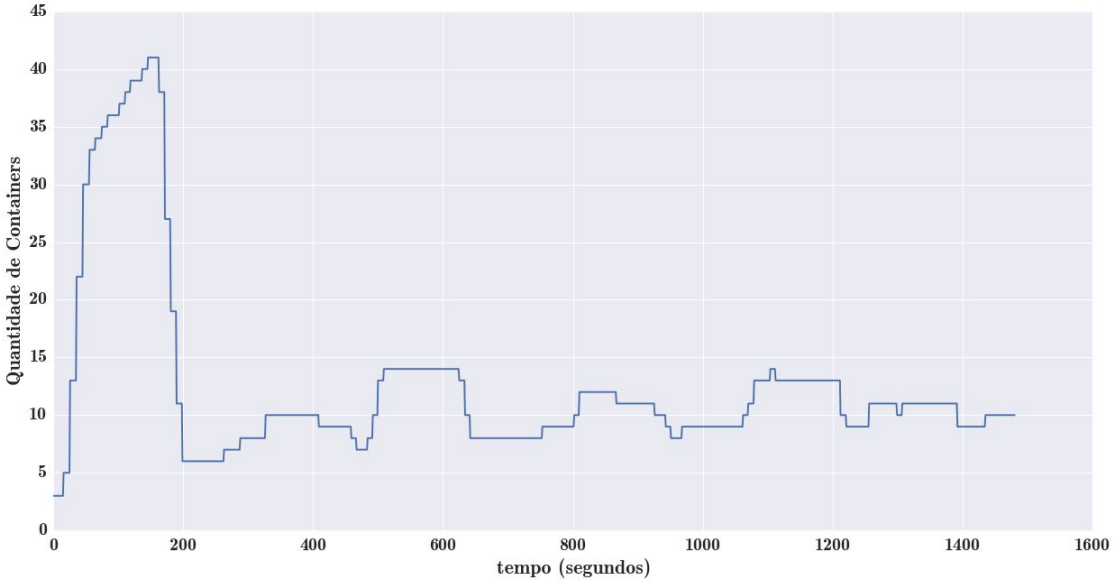

Figura 4.21: Número de *Containers* Rubis x Tempo (ms), carga 4, *setpoint* 80 ms, sem *Coordinate Ascent*

estabelecidas para permitirem que o sistema Rubis fosse iniciado em todos os testes em boas condições de consumo de CPU e processamento. O número mínimo de *containers* em todos os testes foi configurado para três. Isso para permitir que sempre houvesse pelo menos um *container* rodando em cada nó.

As configurações do HPA e do PAS, assim como as nomenclaturas utilizadas, podem ser vistas na Tabela 4.1. No caso do HPA são configurados limites de processamento médio máximo desejado, e no caso do PAS é configurado o limite de tempo de resposta e os parâmetros do PID (*Kp*, *Ki* e*Kd*). Para as versões do algoritmo PAS, PAS 80, PAS 100 e PAS 120 são usados os parâmetros encontrados de forma manual. Para as versões PAS 80 CA, PAS 100 CA e PAS 120 CA são usadas os parâmetros encontrados pelo algoritmo *Coordinate Ascent*, descrito na Seção 3.1.6.

Tabela 4.1: Configuração dos Algoritmos

Algoritmo	**Limite CPU**	**Limite tempo de Resposta**	**Kp**	**Ki**	**Kd**
HPA 20	20%	-	-	-	-
HPA 50	50%	-	-	-	-
PAS 80	-	80 ms	0.01	0.000012	0.096
PAS 80 CA	-	80 ms	0.00121	0.0000505	0.00112
PAS 100	-	100 ms	0.01	0.000012	0.096
PAS 100 CA	-	100 ms	0.00131	0.0000515	0.00101
PAS 120	-	120 ms	0.01	0.000012	0.096
PAS 120 CA	-	120 ms	0.00135	0.000041	0.00115

4.2.6 Resultados Individuais dos Testes

Nesta seção serão apresentados os resultados dos testes para cada um dos algoritmos nas diferentes escalas de intensidade de carga aplicadas. As diversas configurações de cada um dos algoritmos foram testadas com diversas escalas de intensidade de carga (carga_1, carga_2 e carga_3).

As métricas analisadas para os testes foram:

- Tempo médio de espera na camada de aplicação do cliente;
- Tempo médio de espera da requisição no balanceador de carga;
- Porcentagem de requisições não atendidas;
- Número médio de *containers* alocados durante os testes;
- Eficiência na alocação dos *containers*.

Para todos os testes foram configurados intervalos de confiança usando distribuição t de *Student*, com nove graus de liberdade. Para cada teste foram geradas 10 amostras [22], ou seja, cada teste representa a repetição do experimento 10 vezes. Os intervalos de confiança foram definidos em 95%.

O número médio de *containers* foi calculado registrando o número de *containers* alocados a cada segundo, e no final dividindo a soma do número total, pelo tempo em segundos.

A eficiência média na alocação de *containers* foi definida conforme a Equação 4.1, onde Nc é o número médio de *containers* durante os experimentos e T é o tempo médio de resposta em milissegundos das requisições na camada de aplicação. O cálculo da eficiência utilizando esta equação foi idealizado, pois quanto maior o tempo de resposta médio, ou quanto maior a alocação média de *containers*, menor será a eficiência obtida.

$$E = 1/(Nc * T) \tag{4.1}$$

Os resultados para cada uma das métricas estão registrados como o intervalo entre os valores mínimos e máximos, para um intervalo de confiança de 95 %.

As colunas das tabelas 4.2, 4.3 e 4.4, apresentam o seguinte formato:

- (1) Representa o intervalo de confiança para o tempo médio de resposta em milissegundos na camada de aplicação.
- (2) Representa o intervalo de confiança para a alocação média de *containers*
- (3) Representa o intervalo de confiança para a porcentagem de requisições não atendidas;

- (4) Representa o intervalo de confiança para a eficiência do algoritmo;
- (5) Representa o intervalo de confiança para o tempo médio de resposta em milissegundos medido no balanceador;

Resultados para carga_1

Os testes para a carga 1 não serão apresentados, pois verificou-se que com a alocação inicial de recursos realizada para estes testes, os algoritmos de auto elasticidade nas configurações apresentadas na tabela 4.1 não eram perturbados a ponto de realizarem a alocação de novos recursos, portanto, os resultados não foram considerados relevantes para a comparação dos algoritmos.

Desse modo, a apresentação dos resultados, assim como a análise, será realizada a partir da carga_2.

Resultados para carga_2

Neste experimento, o sistema foi submetido a carga_2 durante 1200 segundos. Os resultados obtidos estão sumarizados na tabela 4.2 e nos gráficos das figuras 4.22, 4.23, 4.24, 4.25 e 4.26.

Tabela 4.2: Resultados dos testes para carga_2 com 95% de confiança

	1	2	3	4	5
HPA 20	128.474, 147.355	3.678, 3.779	0.0, 0.0	0.001, 0.001	73.016, 89.781
HPA 50	140.829, 175.214	3.0, 3.0	0.0, 0.0	0.002, 0.002	90.348, 103.327
PAS 80	139.632, 148.248	3.633, 3.644	0.000003, 0.000003	0.001, 0.001	81.110, 81.979
PAS 80 CA	135.526, 139.204	4.036, 4.048	0.0, 0.0	0.001, 0.001	75.965, 76.782
PAS 100	140.908, 152.892	3.015, 3.015	0.0, 0.0	0.002, 0.002	91.022, 93.987
PAS 100 CA	139.546, 154.080	3.020, 3.023	0.0, 0.0	0.002, 0.002	91.131, 94.915
PAS 120	145.905, 164.929	3.0, 3.0	0.0, 0.0	0.002, 0.002	90.601, 99.916
PAS 120 CA	143.285, 165.867	3.0, 3.0	0.0, 0.0	0.002, 0.002	91.797, 98.768

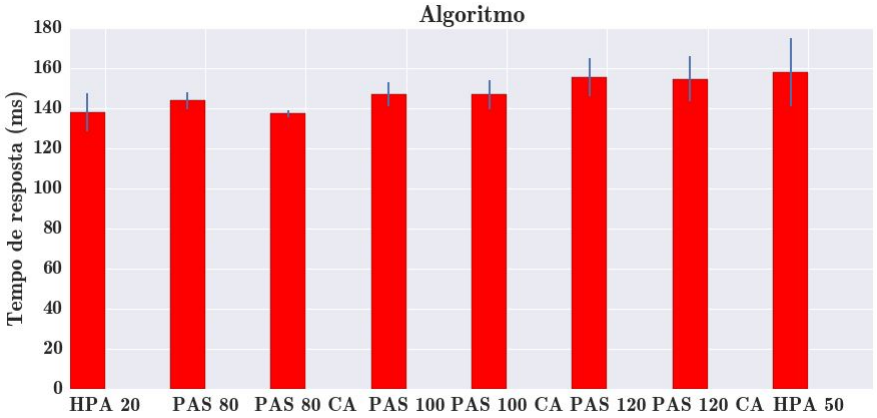

Figura 4.22: Tempo de espera clientes (ms) para carga 2

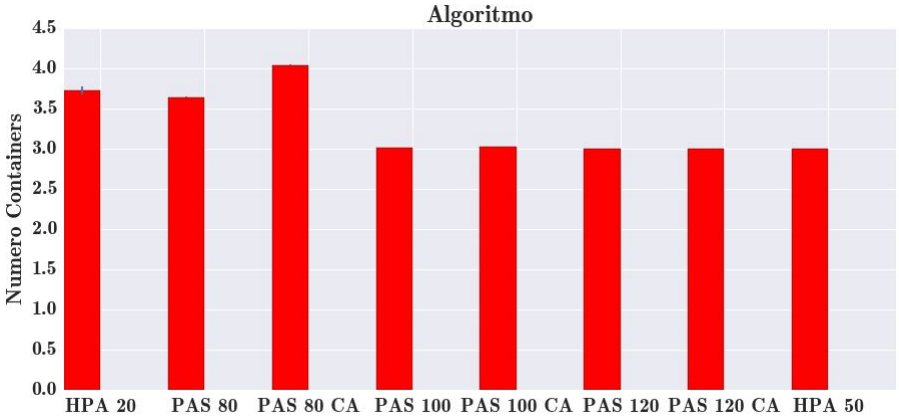

Figura 4.23: Número médio de *containers* para carga 2

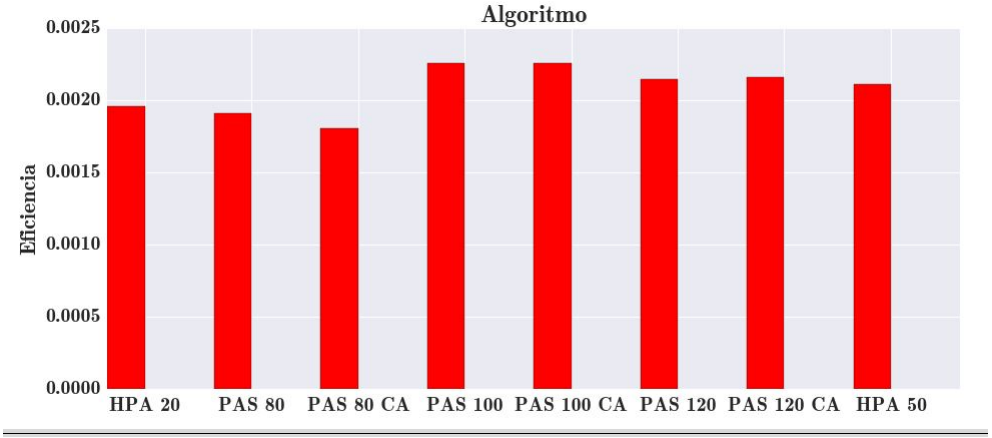

Figura 4.24: Eficiência para carga 2

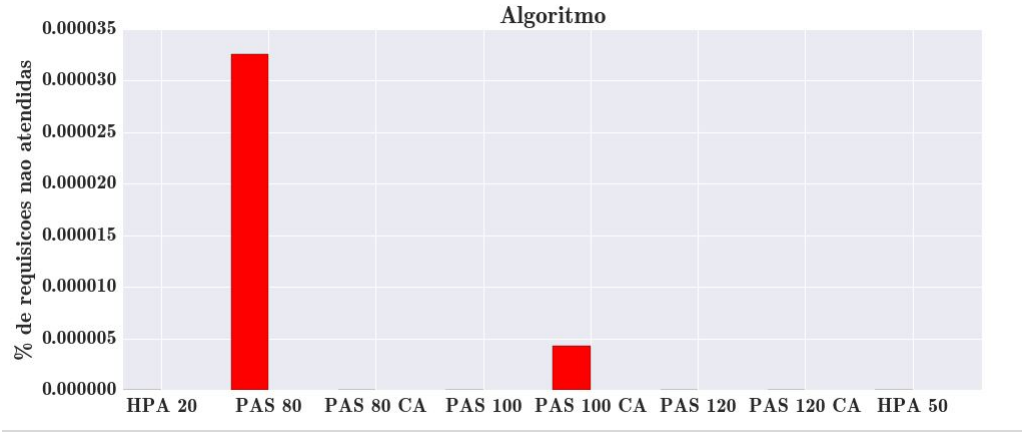

Figura 4.25: Porcentagem de requisições não atendidas para carga 2

O gráfico da Figura 4.22 mostra o tempo de resposta médio obtido na camada de aplicação para a carga_2 em cada uma das configurações. É possível verificar que o HPA 20, PAS 80 e PAS 80 CA obtiveram os melhores tempos de resposta dentro do intervalo

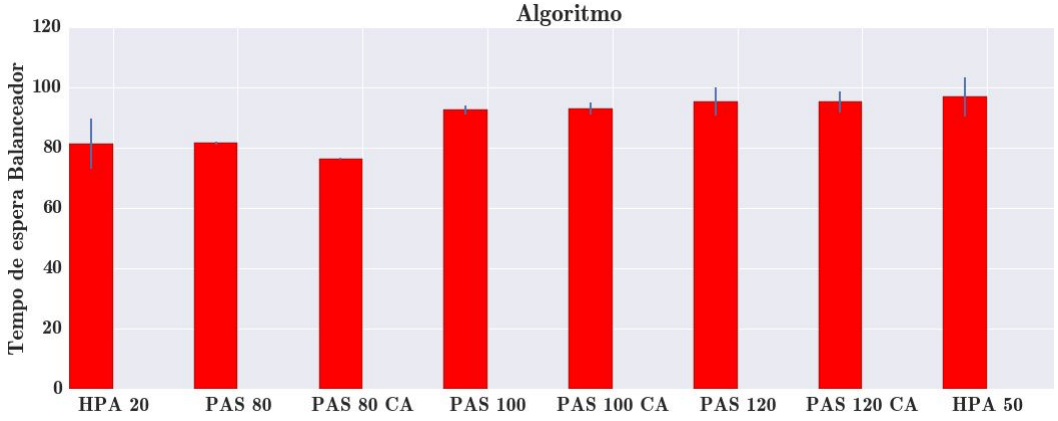

Figura 4.26: Tempo de resposta no balanceador para carga 2

de confiança. A Figura 4.23 mostra que, para isso, os algoritmos tiveram que alocar mais *containers* em média do que as versões com métricas de desempenho mais relaxadas (PAS 100, PAS 100 CA, PAS 120, PAS 120 CA e HPA 50), o que demonstra a coerência dos resultados.

É interessante observar que a carga_2 não foi suficiente para que o processamento médio dos *containers* atingisse 50 %, portanto o HPA 50 não realizou alocação adicional de *containers*. Essa carga também não foi suficiente para que o PAS 120 e o PAS 120 CA realizassem alocação adicional de *containers*, como pode ser observado na Figura 4.23.

O gráfico da Figura 4.24 mostra a eficiência de cada uma das configurações para a carga_2. É possível verificar que, para tempos de resposta equivalentes dentro do intervalo de confiança, o algoritmo HPA 20 obtém melhor eficiência que o PAS 80 e o PAS 80 CA. Porém, a eficiência desses algoritmos para a carga é bem próxima. Para a carga_2, os algoritmos que se sobressaem na métrica de eficiência são os algoritmos PAS 100 e PAS 100 CA. Isso ocorre pois, para a carga_2, esses algoritmos realizaram uma alocação menor de *containers* adicionais e conseguiram abaixar o tempo de resposta médio para os clientes.

A Figura 4.25 mostra a porcentagem de requisições não atendidas para cada uma das configurações. É possível observar que esses valores permanecem em zero na maioria dos testes e os valores acima de zero são baixos, o que demonstra robustez dos algoritmos para criar e destruir *containers* sem prejuízos para o atendimento das requisições.

A Figura 4.26 mostra o tempo médio de espera das requisições medidos no balanceador. Aqui é interessante observar que o PAS 80 e o PAS 80 CA conseguiram médias próximas ao limiar desejado (80 ms). Para as configurações PAS 100, PAS 100 CA, PAS 120 e PAS 120 CA, a intensidade da carga não foi suficiente para que fosse necessária a atuação mais intensa do algoritmo PAS, fazendo com que as médias permanecessem abaixo de 100 ms.

Resultados para carga_3

Neste experimento, o sistema foi submetido a carga_3 durante 1200 segundos. Os resultados obtidos estão sumarizados na tabela 4.3 e nos gráficos das figuras 4.27, 4.28, 4.29, 4.30 e 4.31.

Tabela 4.3: Resultados dos testes para carga_3 com 95% de confiança

	1	2	3	4	5
HPA 20	164.797, 204.326	5.560, 5.684	0.00001, 0.000001	0.0009, 0.0009	61.696, 87.191
HPA 50	199.988, 271.039	3.0, 3.0	0.0, 0.0	0.001, 0.001	134.060, 166.074
PAS 80	168.619, 225.594	5.009, 5.111	0.00007, 0.00007	0.00100, 0.00100	84.202, 95.339
PAS 80 CA	181.819, 185.631	5.270, 5.283	0.00004, 0.00004	0.00103, 0.00103	84.544, 85.648
PAS 100	193.845, 221.755	4.254, 4.277	0.00009, 0.00009	0.0011, 0.0011	103.742, 110.421
PAS 100 CA	180.234, 207.569	4.370, 4.388	0.00001, 0.00001	0.0011, 0.0011	98.089, 99.315
PAS 120	207.609, 215.419	3.492, 3.502	0.00009, 0.00009	0.0013, 0.0013	118.899, 120.642
PAS 120 CA	203.998, 208.970	3.556, 3.561	0.000009, 0.000009	0.0013, 0.0013	113.701, 117.323

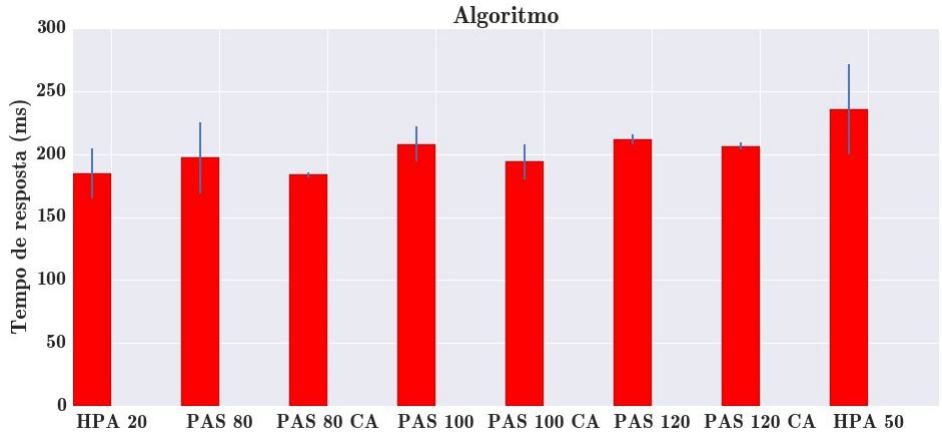

Figura 4.27: Tempo de espera clientes (ms) para carga 3

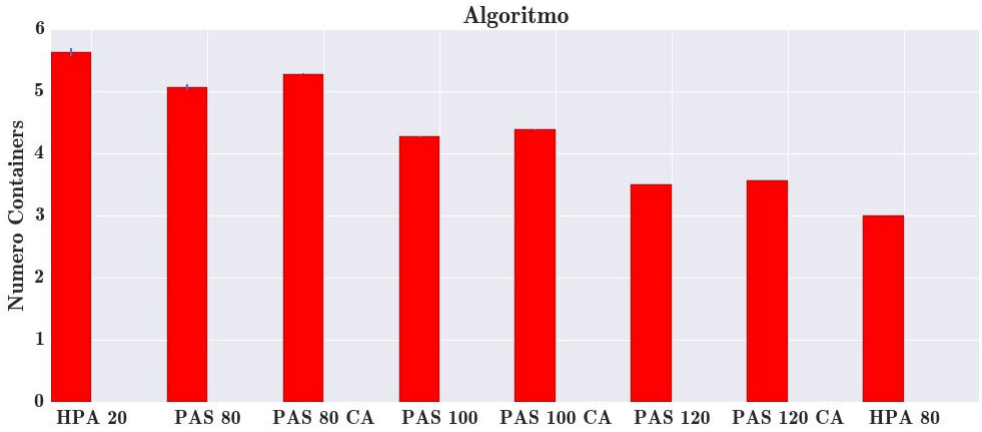

Figura 4.28: Número médio de *containers* para a carga 3

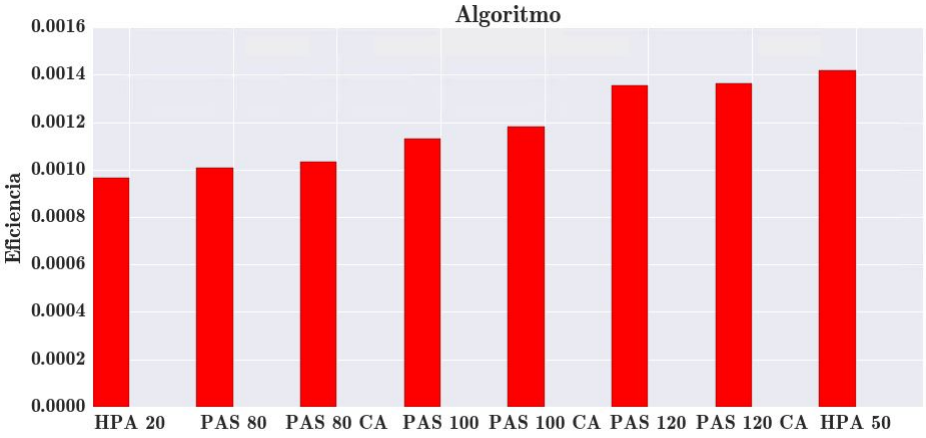

Figura 4.29: Eficiência para carga 3

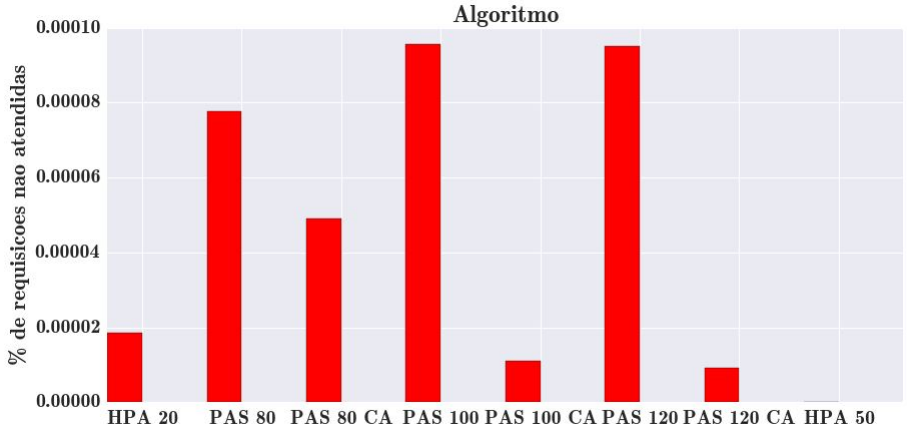

Figura 4.30: Porcentagem de requisições não atendidas para carga 3

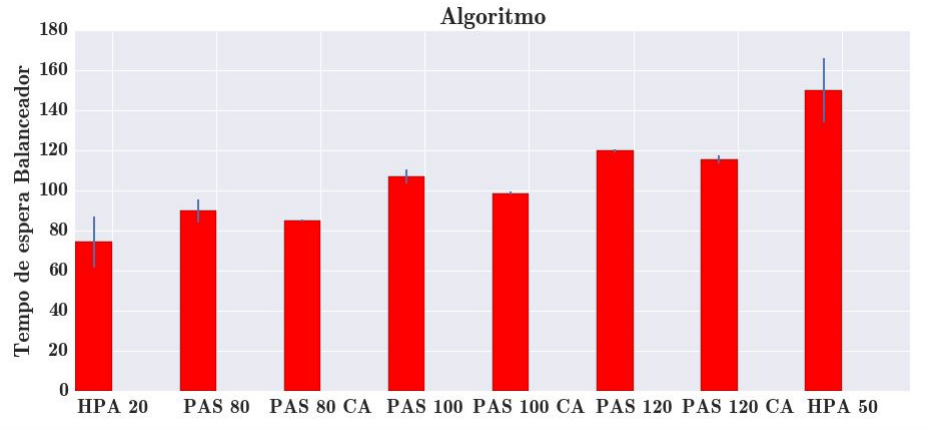

Figura 4.31: Tempo de resposta no balanceador para carga 3

A figura 4.27 mostra o tempo de espera dos clientes para a carga_3. Nesse teste, o PAS 80 CA obteve um tempo de resposta médio bem próximo ao HPA 20 e apresentou um intervalo de confiança menor, o que demonstra a previsibilidade de comportamento em

relação ao tempo de resposta para a carga_3. Assim como nos testes com a carga_2, os algoritmos PAS 100, PAS 100 CA, PAS 120, PAS 120 CA e HPA 50 obtiveram tempos de resposta médios mais altos, realizando uma alocação menor de *containers*. Isso mostra que o objetivo desses algoritmos de auto escalabilidade é atingido, já que é possível escolher um nível de serviço desejado e os algoritmos garantem que o mesmo seja atingido. Caso sejam necessários melhores tempos de resposta, mais *containers* deverão ser alocados. Esse comportamento de melhores tempos de resposta, dependentes da maior alocação de *containers*, será evidenciado na seção seguinte.

A figura 4.30 evidencia que a criação e destruição de *containers* feita pelos algoritmos de auto elasticidade não representaram problemas críticos em relação ao não atendimento de requisições. Um fato interessante é que as versões do algoritmo PAS CA tenderam a perder menos requisições do que as versões otimizadas manualmente. Isso pode ser atribuído ao fato de que os parâmetros otimizados pelo CA fizeram com que o comportamento do sistema apresentasse menores oscilações no tempo de resposta, o que se traduziu em menor atividade de criação e destruição de *containers*, levando a uma menor perda de requisições.

A figura 4.31 mostra que as configurações do PAS conseguiram manter o tempo médio de resposta no balanceador, próximo aos limiares desejados (80, 100 e 120), respectivamente. É possível observar também que para a carga_3 os algoritmos HPA tenderam a ter intervalos de confiança maiores do que os do PAS. Isso demonstra uma maior previsibilidade no atendimento de requisitos de tempo de resposta do PAS. Além disso, as versões do algoritmo PAS 80 CA e PAS 100 CA tenderam a ter intervalos de confiança mais estreitos do que os otimizados manualmente.

As figuras 4.27 e 4.29 mostram que as versões do algoritmo PAS CA obtiveram melhores tempos de resposta que as versões otimizadas com parâmetros manuais, além de obterem melhor eficiência para a carga_3. Além disso, para um tempo de resposta equivalente ao obtido pelo HPA 20, o PAS 80 CA obteve melhor eficiência. Isso demostra que as versões do PAS CA obtiveram excelente desempenho, sendo uma alternativa melhor às versões do PAS otimizadas manualmente, podendo ser também uma alternativa ao HPA.

Resultados para carga variável

Para verificação da diferença de comportamento entre o PAS e o HPA, foram realizados testes com cargas variáveis para as configurações PAS 80 CA e HPA 20. Nesse teste, o sistema foi submetido a carga_1 por 1000 segundos, em seguida a carga_2 por 1000 segundos, depois a carga_3 por mais 1000 segundos e, por fim, novamente a carga_1 por mais 1000 segundos. Essa carga foi chamada de carga_1_2_3_2_1. Os resultados

obtidos estão sumarizados na tabela 4.4 e e nos gráficos das figuras 4.32, 4.33, 4.34, 4.35 e 4.36.

Para efeitos de comparação, os testes com a carga variável foram realizados apenas para os algoritmos HPA 20 e PAS 80 CA.

Tabela 4.4: Resultados dos testes para carga_1_2_3_2_1 com 95% de confiança

	1	2	3	4	5
HPA 20	108.786, 115.244	4.137, 4.143	0.000003, 0.000003	0.0021, 0.0021	73.477, 75.618
PAS 80 CA	108.523, 110.753	4.052, 4.078	0.000003, 0.000003	0.0022, 0.0022	70.861, 71.536

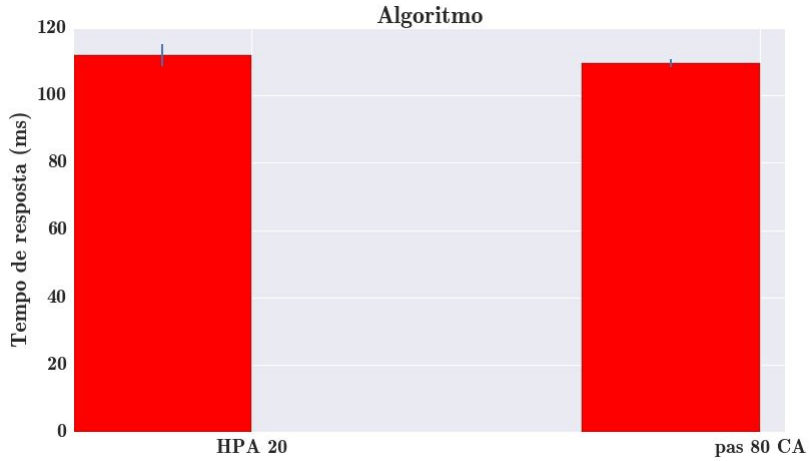

Figura 4.32: Tempo de espera clientes (ms) para carga variável

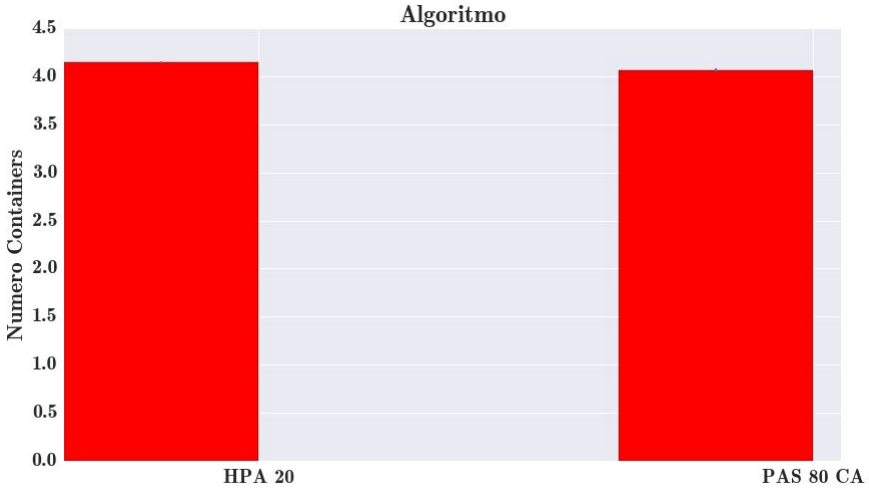

Figura 4.33: Número médio de *containers* para a carga variável

A figura 4.33 mostra o número médio de *containers* alocados durante os testes com a carga variável e a figura 4.32 mostra o tempo de resposta médio na camada de aplicação

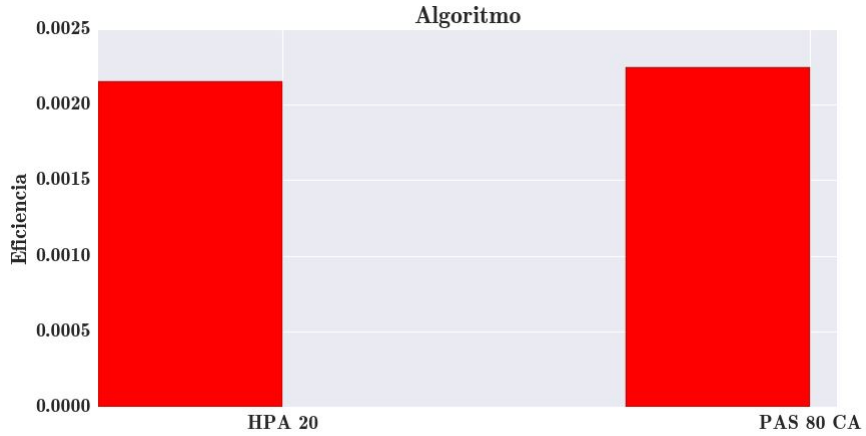

Figura 4.34: Eficiência para carga variável

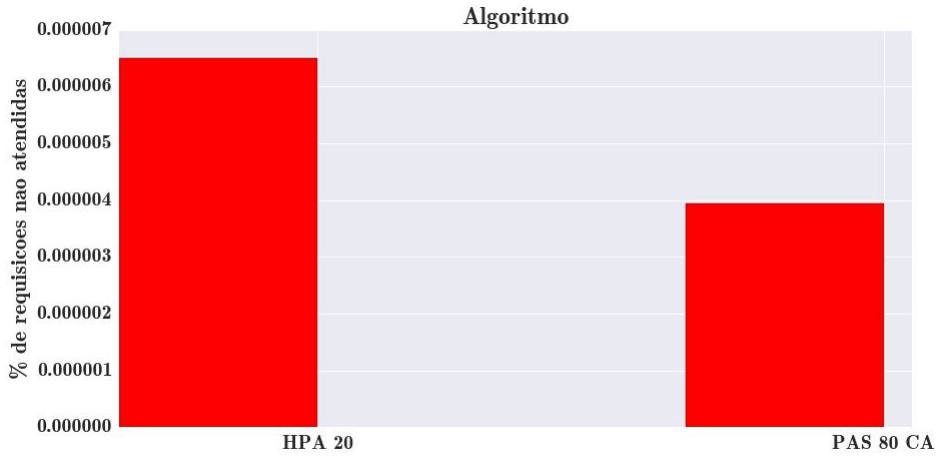

Figura 4.35: Porcentagem de requisições não atendidas para carga variável

obtido para o HPA 20 e o PAS 80 CA. A figura 4.32 mostra que os dois algoritmos tiveram tempos médios de resposta próximos, dentro do intervalo de confiança. Na tabela 4.4 pode ser verificado que o intervalo de confiança obtido para os tempos de resposta se apresenta mais estreito para o PAS 80 CA, o que indica uma maior previsibilidade de desempenho em relação ao HPA 20. Em relação ao número médio de *containers* alocados, o PAS 80 CA alocou menos.

A figura 4.35 mostra a percentagem de requisições não atendidas para o HPA 20 e o PAS 80 CA. Para os dois casos, essa porcentagem foi baixa, o que demonstra a robustez dos dois algoritmos para cargas com mudanças abruptas de intensidade.

A figura 4.34 mostra a eficiência obtida para os algoritmos nesse conjunto de testes. É possível observar que o PAS 80 CA apresentou melhor eficiência que o HPA 20. Isto está de acordo com os outros dados obtidos, já que o PAS 80 CA teve tempos de resposta próximos aos do HPA 20, porém alocou menos *containers* de maneira geral.

A figura 4.36 mostra os tempos médios de resposta, medidos no balanceador de carga.

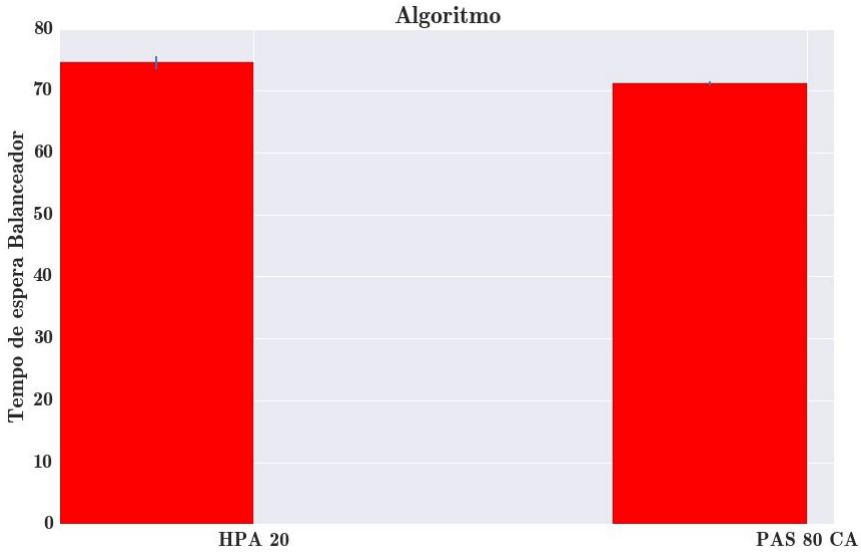

Figura 4.36: Tempo de resposta no balanceador para carga variável

É possível observar que o PAS 80 CA obteve tempos de resposta médio menores que o HPA 20. O PAS 80 CA estava configurado para manter este valor próximo a 80 ms, porém, verifica-se que este valor ficou próximo de 70 ms. Isso pode ser explicado pelo fato de que nesses testes com carga variável, o sistema passa 2000 segundos sendo submetido a carga_1, e esta carga deixa o tempo de resposta abaixo de 80 ms, mesmo sem que o sistema seja escalado.

4.2.7 Testes de Média Zero

Para comparação entre os resultados dos diversos algoritmos e suas configurações foram realizados testes de média zero para auxiliar na análise das diferenças de desempenho entre os algoritmos, para cada um dos cenários.

Os testes de média zero servem para evidenciar estatisticamente se os dados obtidos em uma bateria de testes em um sistema podem ser considerados diferentes dos dados obtidos em baterias de testes realizados em outro sistema, dentro de um intervalo de confiança [22].

Esses testes são realizados da seguinte forma: Para cada parâmetro que se deseja comparar entre dois sistemas, pega-se amostra por amostra obtida em cada teste e subtrai-se os valores obtidos no sistema 1 pelos valores obtidos pelo sistema 2, para as n amostras. Após isso é calculado o intervalo de confiança usando a distribuição t de *Student*, para n - 1 graus de liberdade [22].

Desta forma, intervalos totalmente positivos indicam que os parâmetros obtidos pelo sistema 1 podem ser considerados maiores que os valores obtidos pelo sistema 2. Inter-

valos totalmente negativos indicam que os parâmetros obtidos pelo sistema 2 podem ser considerados maiores que os valores obtidos pelo sistema 1. Intervalos que apresentam valores negativos e positivos indicam que os parâmetros obtidos pelos dois sistemas não podem ser considerados diferentes, dentro do intervalo de confiança estabelecido.

Os testes realizados nesta seção foram configurados para apresentarem intervalo de confiança de 95 %, usando-se a distribuição t de *Student*, com 9 graus de liberdade.

No contexto deste trabalho, cada sistema é representado por um dos algoritmos avaliados. As tabelas que apresentam os resultados dos testes de média zero (tabelas 4.5 a 4.17) o fazem mediante uma comparação cruzada para cada um dos algoritmos, em que cada tupla (x,y) representa no caso de x o algoritmo denominado na linha, e no caso de y o algoritmo denominado na coluna. Por exemplo, na tabela 4.5, a segunda coluna na primeira linha contém o valor (-19.4075207, 0.511573), que representa o teste de média zero do PAS CA 80 em relação ao PAS CA 100.

A primeira comparação realizada visa demonstrar as diferenças entre os tempos de resposta médio na camada de aplicação obtidos para as diversas configurações do PAS CA e a diferença na alocação de *containers* para a carga_2 e a carga_3.

O segundo teste de média zero realizado visa comparar o desempenho do HPA 20 com o PAS 80 CA. Esses dois sistemas foram escolhidos para comparação pois são as configurações com maiores requisitos de desempenho dentre as avaliadas e, portanto, são as aquelas em que os algoritmos tem maior atividade. Serão comparados os tempos de resposta, alocação de *containers* e eficiência. Para essa comparação, além das cargas_2 e carga_3, também será analisada a diferença dos algoritmos para a carga variável (carga_1_2_3_2_1).

A tabela 4.5 mostra a comparação dos tempos de resposta do PAS CA para a carga_2. Na linha do PAS CA 80, a comparação com o PAS CA 100 e o PAS CA 120 demonstra que apesar de nos intervalos aparecerem valores negativos e positivos, a porção negativa dos intervalos é bem maior que a porção positiva, o que indica que os tempos de resposta do PAS CA 100 e PAS CA 120 foram maiores que os do PAS CA 80. O mesmo ocorre verificando na linha do PAS CA 100 a comparação com o PAS CA 120. Esse é o comportamento esperado, indicando que configurações do PAS com limiares inferiores de tempo de resposta conseguem atingir esta meta na camada de aplicação dos clientes.

A tabela 4.6 mostra a comparação dos tempos de resposta para o PAS CA para a carga_3. Assim como na carga_2, a comparação do PAS 80 CA com o PAS 100 CA demonstra uma maior porção negativa dentro do intervalo de confiança para o teste, indicando maiores tempos de resposta para o PAS 100 CA. Já para a comparação do PAS 80 CA com o PAS 120 CA, o intervalo do teste é totalmente negativo, evidenciando melhores tempos de resposta para o PAS 80 CA.

As tabelas 4.7 e 4.8 mostram os testes de média zero para comparação entre os PAS CA, para a carga_2 e carga_3. As linhas de comparação do PAS CA 80 com os outros dois algoritmos possuem intervalos totalmente positivos nas duas tabelas. Isso indica que o PAS 80 CA alocou mais *containers* que o PAS 100 CA e o PAS 120 CA. É possível observar também nas duas tabelas que o PAS 100 CA alocou mais *containers* que o PAS 120 CA.

Os resultados anteriores indicam a coerência no funcionamento do algoritmo para as diversas configurações. Na continuação, será comparado o PAS 80 CA com o HPA 20. As tabelas 4.9 e 4.10 mostram os testes de média zero para o tempo de resposta, para as cargas 2 e 3. O intervalo apresentado nas duas tabelas indica intervalos de valores negativos e positivos quase simétricos. Isso indica que os tempos de resposta obtidos para cada um dos algoritmos para a carga_2 e carga_3 foram equivalentes. A tabela 4.11 mostra os testes de média zero para os dois algoritmos nos testes com a carga variável (carga_1_2_3_2_1). O intervalo apresentado indica tempos de resposta equivalentes, porém com o HPA 20 tendendo a ter tempos de resposta um pouco maiores.

As tabelas 4.12, 4.13 e 4.14 apresentam a alocação de *containers* nos testes. Para a carga_2, o PAS 80 CA fez uma alocação maior de *containers*. Já para a carga_3 e para a carga variável, o HPA 20 fez uma alocação maior de *containers*.

As tabelas 4.15, 4.16 e 4.17 demonstram a diferença de eficiência entre os algoritmos para os testes com as três cargas. Para a carga_2, o HPA 20 teve uma eficiência maior. Porém, para a carga_3 e para a carga variável, o PAS 80 CA foi mais eficiente.

Neste trabalho, a comparação entre o PAS 80 CA e o HPA 20 demonstrou que, apesar da dificuldade de se comparar algoritmos que trabalham com métricas diferentes para auto escalabilidade, utilizando-se a observação da proximidade dos resultados entre o HPA 20 e o PAS 80 CA, foi possível apreciar que os dois algoritmos obtiveram tempos de resposta próximos na camada de aplicação, assim como na alocação de *containers*. O PAS 80 CA obteve uma melhor eficiência que o HPA 20 para duas das três cargas avaliadas.

Tabela 4.5: Testes média zero para o tempo de resposta, carga_2, 95% de confiança, PAS 80 CA, PAS 100 CA e PAS 120 CA

Algoritmo	**PAS CA 80**	**PAS CA 100**	**PAS CA 120**
PAS CA 80	(0.0, 0.0)	(-19.4075207, 0.511573)	(-34.4990513, 0.07708)
PAS CA 100	(-0.5115725, 19.407521)	(0.0, 0.0)	(-25.2108117, 9.684788)
PAS CA 120	(-0.0770798, 34.499051)	(-9.6847883, 25.210812)	(0.0, 0.0)

Tabela 4.6: Testes média zero para o tempo de resposta, carga_3, 95% de confiança, PAS 80 CA, PAS 100 CA e PAS 120 CA

Algoritmo	**PAS CA 80**	**PAS CA 100**	**PAS CA 120**
PAS CA 80	(0.0, 0.0)	(-26.575802, 6.223212)	(-29.9324576, -15.584492)
PAS CA 100	(-6.223212, 26.575802)	(0.0, 0.0)	(-28.0133645, 2.849005)
PAS CA 120	(15.5844917, 29.932458)	(-2.8490051, 28.013364)	(0.0, 0.0)

Tabela 4.7: Testes média zero para a alocação média de *containers*, carga_2, 95% de confiança, PAS 80 CA, PAS 100 CA e PAS 120 CA

Algoritmo	**PAS CA 80**	**PAS CA 100**	**PAS CA 120**
PAS CA 80	(0.0, 0.0)	(1.0110342, 1.030403)	(1.0364059, 1.048306)
PAS CA 100	(-1.0304027, -1.011034)	(0.0, 0.0)	(0.0202153, 0.02306)
PAS CA 120	(-1.0483059, -1.036406)	(-0.0230595, -0.020215)	(0.0, 0.0)

Tabela 4.8: Testes média zero para a alocação média de *containers*, carga_3, 95% de confiança, PAS 80 CA, PAS 100 CA e PAS 120 CA

Algoritmo	**PAS CA 80**	**PAS CA 100**	**PAS CA 120**
PAS CA 80	(0.0, 0.0)	(0.8882183, 0.907256)	(1.7109587, 1.725472)
PAS CA 100	(-0.907256, -0.888218)	(0.0, 0.0)	(0.8104369, 0.83052)
PAS CA 120	(-1.7254721, -1.710959)	(-0.8305197, -0.810437)	(0.0, 0.0)

Tabela 4.9: Testes média zero para tempo de resposta, carga_2, 95% de confiança, HPA 20, PAS 80 CA

Algoritmo	**HPA 20**	**PAS CA 80**
HPA 20	(0.0, 0.0)	(-12.0449785, 13.144644)
PAS 80 CA	(-13.1446443, 12.044978)	(0.0, 0.0)

Tabela 4.10: Testes média zero para tempo de resposta, carga_3, 95% de confiança, HPA 20, PAS 80 CA

Algoritmo	**HPA 20**	**PAS CA 80**
HPA 20	(0.0, 0.0)	(-20.7722752, 22.444744)
PAS 80 CA	(-22.4447442, 20.772275)	(0.0, 0.0)

Tabela 4.11: Testes média zero para tempo de resposta, carga variável, 95% de confiança, HPA 20, PAS 80 CA

Algoritmo	**HPA 20**	**PAS CA 80**
HPA 20	(0.0, 0.0)	(-0.5108585, 5.26436)
PAS 80 CA	(-5.26436, 0.510859)	(0.0, 0.0)

Tabela 4.12: Testes média zero para número médio de *containers*, carga_2, 95% de confiança, HPA 20, PAS 80 CA

Algoritmo	**HPA 20**	**PAS CA 80**
HPA 20	(0.0, 0.0)	(-0.3848726, -0.241527)
PAS 80 CA	(0.2415267, 0.384873)	(0.0, 0.0)

Tabela 4.13: Testes média zero para número médio de *containers*, carga_3, 95% de confiança, HPA 20, PAS 80 CA

Algoritmo	**HPA 20**	**PAS CA 80**
HPA 20	(0.0, 0.0)	(0.2767679, 0.414394)
PAS 80 CA	(-0.4143941, -0.276768)	(0.0, 0.0)

Tabela 4.14: Testes média zero para número médio de *containers*, carga variável, 95% de confiança, HPA 20, PAS 80 CA

Algoritmo	**HPA 20**	**PAS CA 80**
HPA 20	(0.0, 0.0)	(0.0589784, 0.097679)
PAS 80 CA	(-0.0976786, -0.058978)	(0.0, 0.0)

Tabela 4.15: Testes média zero para eficiência, 95% de confiança, carga_2, HPA 20, PAS 80 CA

Algoritmo	**HPA 20**	**PAS CA 80**
HPA 20	(0.0, 0.0)	(0.0001546, 0.000155)
PAS 80 CA	(-0.0001546, -0.000155)	(0.0, 0.0)

Tabela 4.16: Testes média zero para eficiência, 95% de confiança, carga_3, HPA 20, PAS 80 CA

Algoritmo	**HPA 20**	**PAS CA 80**
HPA 20	(0.0, 0.0)	(-6.53e-05, -6.5e-05)
PAS 80 CA	(6.53e-05, 6.5e-05)	(0.0, 0.0)

Tabela 4.17: Testes média zero para eficiência, carga variável, 95% de confiança, HPA 20, PAS 80 CA

Algoritmo	**HPA 20**	**PAS CA 80**
HPA 20	(0.0, 0.0)	(-8.95e-05, -8.9e-05)
PAS 80 CA	(8.95e-05, 8.9e-05)	(0.0, 0.0)

4.2.8 Análise dos Resultados

Os resultados experimentais mostram que nos cenários avaliados o algoritmo PAS otimiza a alocação do número de *containers* para manter os valores de tempos de resposta dentro de um limiar estabelecido.

Foram realizados alguns testes simples, que serviram como prova de conceito para avaliar a viabilidade da proposta. Posteriormente, foi definida uma sistemática de avaliação de desempenho para comparar a solução proposta com uma das soluções mais conhecidas do mercado.

Duas versões do algoritmo PAS foram avaliadas. A primeira versão teve os parâmetros do controlador PID configurados de forma manual (PAS). A segunda versão teve os parâmetros do PID configurados de forma automática usando o algoritmo *coordinate ascent* (PAS CA). Os resultados demonstraram que o PAS CA obteve bom funcionamento durante os testes, sendo eficiente na alocação de *containers* para a manutenção dos limiares de tempo de resposta. A configuração automática dos parâmetros do PID, sem a necessidade de intervenção humana, facilita a configuração do algoritmo, tornando-o mais prático.

A comparação do PAS com o HPA demonstrou que o PAS tem potencial para ser uma alternativa para prover auto escalabilidade em sistemas de nuvem. O uso da métrica de tempo de resposta, ao invés de consumo de CPU, é interessante por se tratar de métrica diretamente relacionada com a percepção do usuário sobre o desempenho do sistema. Além disso, aplicações com requisitos de baixa latência podem ser se beneficiar da auto escalabilidade, com configurações que respeitem os tempos de resposta necessários.

Os resultados apresentados nos experimentos mostram que a arquitetura proposta com o uso do algoritmo PAS é promissora para promover uma maior eficiência na auto escalabilidade de sistemas em nuvem com requisitos específicos de *QoS*.

4.3 Conclusão deste Capítulo

Este capítulo descreveu a validação experimental da arquitetura proposta com diferentes intensidades de carga de trabalho. Foram analisadas as métricas de tempos de resposta, alocação de *containers*, porcentagem de requisições não atendias e eficiência, que demonstraram a qualidade da proposta apresentada.

A arquitetura proposta neste trabalho foi comparada com o algoritmo de auto escalabilidade nativo do *Kubernetes*, o *HPA*. Trata-se de um algoritmo desenvolvido pelo *Google*, utilizado em produção por empresas de diversos portes. O PAS CA conseguiu obter funcionalidades de auto escalabilidade similares ao *HPA*, porém trabalhando com métricas diferentes para engatilhar a escalabilidade automática do ambiente.

Vale destacar que, apesar da proposta deste trabalho utilizar métricas de tempo de resposta médio, a arquitetura proposta poderia ser adaptada para trabalhar com qualquer métrica desejada, como consumo médio de CPU, memória, número de requisições ou mesmo com combinações de métricas. No caso de métricas de consumo de CPU e memória, uma ferramenta de monitoramento de recursos de *containers* como o *cadvisor* [18] ou o *heapster* [20] poderiam ser integrados ao sistema. No caso de número de requisições, o PAS poderia continuar recebendo dados do balanceador de carga. Essas métricas adicionais tornariam o PAS uma alternativa mais flexível.

Outro ponto a ser observado é que o algoritmo PAS é independente do *Kubernetes* e poderia ser adaptado para ser utilizado com outros gerenciadores de *containers*, como o *Docker Swarm* [10] ou o *Marathon* [30]. Já o HPA é implementado para uso com o *Kubernetes*, não podendo ser aproveitado em outros ambientes. Esse ponto é importante, pois o PAS tem potencial para ser uma ferramenta de auto escalabilidade genérica, facilitando sua adoção e evitando a amarração em tecnologias (*vendor lock in*).

Capítulo 5

Conclusões e Trabalhos Futuros

Este trabalho apresentou uma proposta de uma arquitetura auto escalável para computação em nuvem. O interesse da pesquisa neste tema surgiu da possibilidade de migração de serviços da Controladoria-Geral da União para ambientes de nuvem.

A computação em nuvem traz diversas características que a tornam interessantes para uso de entidades do governo federal, assim como entidades privadas. A possibilidade do uso de infraestrutura de terceiros, para provisão de serviços, como é feito por exemplo em nuvens públicas, podem levar a barateamento de custos com operação, compra de equipamentos e licenças de *software*. Além disso, funcionalidades como auto elasticidade, podem levar a economia ainda maior, pois os recursos são escalados de acordo com a demanda. Isso diminui os custos com recursos ociosos.

Os paradigmas de nuvem privada também trazem ganhos para as corporações na medida em que são capazes de trazer maior agilidade para provisionamento de recursos, dimensionamento adequado de recursos por meio de técnicas de auto elasticidade e melhor visibilidade da infraestrutura. Isso pode se traduzir em ganhos operacionais e financeiros.

Neste sentido, foi apresentada neste trabalho uma arquitetura inovadora para provisão de auto elasticidade para ambientes de nuvem. Durante a pesquisa do estado da arte de ambientes de computação auto elásticos, verificou-se que maioria dos trabalhos propunha o uso de máquinas virtuais e métodos de elasticidade baseados em recursos como uso de CPU e memória RAM. A proposta deste trabalho é o uso de *containers*, e a provisão da auto elasticidade é feita de acordo com a análise de uma métrica que afeta diretamente a percepção de um usuário de um sistema, no caso, o tempo de resposta da aplicação.

Para atingir esse objetivo, foram integradas ferramentas de *Big Data*, orquestrador de *containers*, banco *NoSQL*, além de técnicas de controle e de otimização de parâmetros. Isso trouxe um caráter inovador para a proposta, pois foram integradas ferramentas e técnicas de diversos domínios, que atuaram de forma a trazer as funcionalidades idealizadas.

A proposta foi testada, usando cargas de trabalho simples, com as quais a viabilidade da proposta foi confirmada, e também com cargas mais complexas, com as quais também se obteve sucesso na provisão de auto elasticidade. Os testes com a ferramenta foram extensos, sendo que, em um dos cenários, a arquitetura foi submetida a cargas durante mais de uma semana, vinte e quatro horas por dia. Isso demonstra a robustez da proposta, mesmo essa ainda estando na fase de protótipo.

A arquitetura foi comparada com uma das ferramentas dominantes de mercado para provisão de auto elasticidade e os resultados dos testes comparativos demonstraram o potencial da proposta, com a provisão de bom desempenho e alocação de recursos de forma eficiente.

5.1 Trabalhos Futuros

A sequência deste trabalho pode ir no caminho da integração de novas métricas para provisão de auto elasticidade, como CPU, memória e requisições por segundo. Outra evolução interessante seria a integração desta proposta com outras ferramentas de orquestração de *containers*, o que tornaria esta proposta um mecanismo genérico para provisão de auto elasticidade.

Outros algoritmos poderiam ser avaliados no lugar do algoritmo de controle utilizado neste trabalho. Poderiam, por exemplo, ser explorados algoritmos de inteligência artificial. A implantação desta proposta em um ambiente de produção será importante para a verificação de possíveis questões que não ficaram evidentes durante os testes realizados com cargas sintéticas, ainda que estas fossem criadas a partir de cargas reais.

A avaliação deste trabalho foi feita em um ambiente privado, portanto, seria interessante a realização de testes com esta proposta em um ambiente de nuvem pública. Neste caso, poderia ser feito um estudo que verificasse a economia financeira em termos quantitativos, que pode-se ter utilizando esta proposta.

Seria interessante a verificação do comportamento da solução no seguinte cenário: Toda a arquitetura proposta neste trabalho, atuando como serviço em um ambiente remoto de nuvem pública, provendo auto elasticidade para sistemas que poderiam estar em outros provedores de nuvem ou mesmo em nuvens privadas. Isso poderia se apresentar na forma de "Auto Elasticidade Como Serviço".

Referências

[1] Khalid Alhamazani, Rajiv Ranjan, Karan Mitra, Fethi Rabhi, Prem Prakash Jayaraman, Samee Ullah Khan, Adnene Guabtni, e Vasudha Bhatnagar. An overview of the commercial cloud monitoring tools: research dimensions, design issues, and state-of-the-art. *Computing*, 97(4):357–377, 2015. 8

[2] Apache. ab apache http server benchmarking tool. `http://httpd.apache.org/docs/2.4/programs/ab.html`, 2016. [Acessado em 03/04/2016]. 38

[3] Priscila América Solis Mendez Barreto. *Uma Metodologia de Engenharia de Tráfego Baseada na Abordagem Auto-Similar para a caracterização de parâmetros e a otimização de redes multimídia.* Tese (Doutorado), Universidade de Brasília, 2010. 16, 17

[4] Alessio Botta, Walter de Donato, Valerio Persico, e Antonio Pescapé. Integration of cloud computing and internet of things: a survey. *Future Generation Computer Systems*, 56:684–700, 2016. 10

[5] Richard B Bunt, Derek L Eager, Gregory M Oster, e Carey L Williamson. Achieving load balance and e ective caching in clustered web servers. 1999. 22

[6] CGU. Portal da transparência. `http://transparencia.gov.br/`, 2016. [Acessado em 23/11/2016]. 1, 38

[7] Mark E Crovella e Azer Bestavros. Self-similarity in world wide web traffic: evidence and possible causes. *Networking, IEEE/ACM Transactions on*, 5(6):835–846, 1997. 16, 17

[8] Ministério do Planejamento. Boas práticas, orientações e vedações para contratação de serviços de computação em nuvem. `https://www.governoeletronico.gov.br/documentos-e-arquivos/orientacaoservicosemnuvem.pdf`, 2016. [Acessado em 16/11/2016]. 1

[9] Docker. The definitive guide to docker containers. `https://www.Docker.com/sites/default/files/WP-%20Definitive%20Guide%20To%20Containers.pdf`, 2016. [Acessado em 03/04/2016]. 14, 15, 24

[10] Docker. Docker swarm overview. `https://docs.docker.com/swarm/overview/`, 2016. [Acessado em 10/11/2016]. 67

[11] Governo Federal. Lei nº 12.527, de 18 de novembro de 2011. http://www.planalto.gov.br/ccivil_03/_ato2011-2014/2011/lei/l12527.htm, 2011. [Acessado em 16/11/2016]. 1

[12] Wes Felter, Alexandre Ferreira, Ram Rajamony, e Juan Rubio. An updated performance comparison of virtual machines and linux containers. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*, pages 171–172. IEEE, 2015. 14

[13] Flume. Flume user guide. https://flume.apache.org/FlumeUserGuide.html, 2016. [Acessado em 03/04/2016]. 29

[14] Katja Gilly, Carlos Juiz, e Ramon Puigjaner. An up-to-date survey in web load balancing. *World Wide Web*, 14(2):105–131, 2011. 22

[15] Zhenhuan Gong, Xiaohui Gu, e John Wilkes. Press: Predictive elastic resource scaling for cloud systems. In *Network and Service Management (CNSM), 2010 International Conference on*, pages 9–16. IEEE, 2010. 2, 23

[16] Google. Architecture: Web application on google app engine. https://mesosphere.github.io/marathon/, 2016. [Acessado em 10/11/2016]. 10

[17] Google. Container cluster manager from google. https://github.com/kubernetes/kubernetes, 2016. [Acessado em 03/04/2016]. 26

[18] Google. Google cadvisor (container advisor). https://github.com/google/cadvisor, 2016. [Acessado em 03/04/2016]. 67

[19] Google. Horizontal pod autoscaler. https://github.com/kubernetes/kubernetes/blob/release-1.2/docs/design/horizontal-pod-autoscaler.md, 2016. [Acessado em 03/04/2016]. 23, 28, 35

[20] heapster. Heapster compute resource usage analysis and monitoring of container clusters. https://github.com/kubernetes/heapster, 2016. [Acessado em 03/11/2016]. 67

[21] National Instruments. Explicando a teoria pid. http://www.ni.com/white-paper/3782/pt/, 2015. [Acessado em 20/08/2015]. 18, 20

[22] Raj Jain. *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling*. John Wiley & Sons, 1990. 52, 61

[23] Kairos. Time series data storage in redis, mongo, sql and cassandra. https://pypi.python.org/pypi/kairos, 2015. [Acessado em 03/04/2016]. 29

[24] Houssain Kettani, John Gubner, et al. A novel approach to the estimation of the hurst parameter in self-similar traffic. In *Local Computer Networks, 2002. Proceedings. LCN 2002. 27th Annual IEEE Conference on*, pages 160–165. IEEE, 2002. 38

[25] Linux. Man-page namespaces(7). http://man7.org/linux/man-pages/man7/namespaces.7.html, 2016. [Acessado em 16/11/2016]. 14

[26] Tania Lorido-Botrán, José Miguel-Alonso, e Jose Antonio Lozano. Auto-scaling techniques for elastic applications in cloud environments. *Department of Computer Architecture and Technology, University of Basque Country, Tech. Rep. EHU-KAT-IK-09*, 12:2012, 2012. 2, 23

[27] Nick Martin. A brief history of docker containers' overnight success. http://searchservervirtualization.techtarget.com/feature/A-brief-history-of-Docker-Containers-overnight-success, 2015. [Acessado em 03/04/2016]. 25

[28] Toni Mastelic, Ariel Oleksiak, Holger Claussen, Ivona Brandic, Jean-Marc Pierson, e Athanasios V Vasilakos. Cloud computing: Survey on energy efficiency. *ACM Computing Surveys (CSUR)*, 47(2):33, 2015. 2, 3, 9

[29] Peter Mell e Tim Grance. The nist definition of cloud computing. *National Institute of Standards and Technology*, 53(6):50, 2009. 4

[30] Mesosphere. A container orchestration platform for mesos and dc/os. https://mesosphere.github.io/marathon/, 2016. [Acessado em 03/04/2016]. 67

[31] Microsoft. Visão geral de eliminação de duplicação de dados. https://technet.microsoft.com/pt-br/library/hh831602(v=ws.11).aspx, 2012. [Acessado em 10/11/2016]. 9

[32] Microsoft. Cache redis do azure. https://azure.microsoft.com/pt-br/services/cache/, 2016. [Acessado em 10/11/2016]. 11

[33] Rishabh Poddar, Anilkumar Vishnoi, e Vijay Mann. Haven: Holistic load balancing and auto scaling in the cloud. In *2015 7th International Conference on Communication Systems and Networks (COMSNETS)*, pages 1–8. IEEE, 2015. 2, 23

[34] Lócio Fernando Postai. Caracterização de tráfego em uma rede san-ip utilizando protocolo iscsi. Dissertação (Mestrado), Universidade de Brasília, 2011. 16, 17

[35] RedHat. Resource management guide. https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Resource_Management_Guide/ch01.html, 2016. [Acessado em 16/11/2016]. 15

[36] Redis. Redis documentation. http://redis.io/documentation, 2015. [Acessado em 03/04/2016]. 29, 32

[37] Rubis. Rubis rice university bidding system. http://rubis.ow2.org/, 2009. [Acessado em 03/04/2016]. 36

[38] Omar Sefraoui, Mohammed Aissaoui, e Mohsine Eleuldj. Openstack: toward an open-source solution for cloud computing. *International Journal of Computer Applications*, 55(3), 2012. 13

[39] Marcelo Abranches; Priscila Solis. An algorithm based on response time and traffic demands to scale containers on a cloud computing system. *he 15th IEEE International Symposium on Network Computing and Applications (NCA 2016)*, 1:343–350, 2016. x, 99

[40] Marcelo Abranches; Priscila Solis. Um algoritmo de auto elasticidade para ambientes de computação em nuvem e tráfego autossimilar. *2016 XLII Latin American Computing Conference CLEI*, 1:244–252, 2016. x, 89

[41] Marcelo Abranches; Priscila Solis. Um mecanismo de auto elasticidade com base no tempo de resposta para ambientes de computação em nuvem baseados em containers. *Workshop on Cloud Networks*, 36:1959–1972, 2016. x, 74

[42] Spark. Spark programming guide. `https://spark.apache.org/docs/1.5.2/programming-guide.html`, 2015. [Acessado em 03/04/2016]. 28

[43] Spark. Spark streaming programming guide. `https://spark.apache.org/docs/1.5.2/streaming-programming-guide.html`, 2015. [Acessado em 03/04/2016]. 29

[44] Squazt. Implementation of the rice university bidding system (rubis). `https://github.com/sguazt/RUBiS`, 2012. [Acessado em 10/11/2015]. 36

[45] Willy Tarreau. Haproxy configuration manual. `http://www.haproxy.org/download/1.5/doc/configuration.txt`, 2015. [Acessado em 03/04/2016]. 29, 32

[46] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, e John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, page 18. ACM, 2015. 26

[47] VMware. What is virtualization. `http://www.vmware.com/solutions/virtualization.html`, 2016. [Acessado em 16/11/2016]. 13

[48] Sean Walberg. Automate system administration tasks with puppet. *Linux Journal*, 2008(176):5, 2008. 14

[49] E. Yakubovich. Introducing flannel: An etcd backed overlay network for containers. `https://coreos.com/blog/introducing-rudder/`, 2014. [Acessado em 16/11/2016]. 28

[50] Larry Zhang. Understand storage space tiering in windows server 2012 r2. `UnderstandStorageSpaceTieringinWindowsServer2012R2`, 2015. [Acessado em 10/11/2016]. 9

[51] Qi Zhang, Lu Cheng, e Raouf Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of internet services and applications*, 1(1):7–18, 2010. 7, 8, 9

Anexo I

Artigo publicado no Congresso WCN 2016 - Porto Alegre/Brasil [41]

Um Mecanismo de Auto Elasticidade com base no Tempo de Resposta para Ambientes de Computação em Nuvem baseados em Containers

Marcelo Cerqueira de Abranches[1,2], Priscila Solis[1]

[1]Departamento de Ciência da Computação – Universidade de Brasília (UnB)
Prédio CIC/EST, Caixa Postal 4466 – 70.910-900 – Brasília – DF – Brazil

[2]Diretoria de Sistemas e Informação – Controladoria-Geral da União (CGU)
SAUS, Quadra 1, Bloco A – 70.050-904 – Brasília – DF – Brazil

marcelo.abranches@cgu.gov.br, pris@cic.unb.br

Abstract. *This paper proposes a cloud computing architecture based on containers and an algorithm for auto elasticity based on the response time requirements in a Web system. The proposed algorithm promotes an efficient allocation of containers to achieve expected response time in processing requests. This proposal was evaluated with a real time series obtained from a web system hosted by Controladoria-Geral da União (CGU). The results show that the proposed algorithm achieves the expected response times allocating a lower number of containers than other related proposals.*

Resumo. *Este artigo propõe uma arquitetura de computação em nuvem baseada em containers e um algoritmo de auto elasticidade com base no tempo de resposta de um sistema Web. O algoritmo proposto promove a otimização do processamento das requisições por meio da alocação eficiente do número de containers para alcançar tempos esperados de resposta no processamento de requisições. A proposta foi avaliada com uma carga de trabalho caracterizada com base em uma série temporal real resultante das requisições de um sistema Web da Controladoria Geral da União (CGU). Os resultados mostram que o algoritmo proposto consegue cumprir os requerimentos de desempenho alocando um número menor de containers que outras propostas relacionadas.*

1. Introdução

A pesquisa nos ambientes de computação em nuvem apresenta como um dos maiores pontos de interesse algoritmos de auto escalabilidade [Poddar et al. 2015] e balanceamento de carga. Como em qualquer processo de dimensionamento e planejamento, um dos pontos cruciais para esta tarefa é a correta caracterização da carga de trabalho e a otimização no uso de recursos, os quais devem atender um conjunto de requerimentos de desempenho das aplicações.

Diversos trabalhos recentes abordam o dimensionamento elástico e a auto escalabilidade em ambientes de computação em nuvem [Poddar et al. 2015], [Lorido-Botrán et al. 2012], [Gong et al. 2010] e [Google 2016b]. As soluções propostas nestes trabalhos são na sua maioria baseadas em heurísticas, teoria de filas, análises de séries temporais, reação a aumento de consumo de recursos entre outros.

Este trabalho propõe um algoritmo para promover a auto elasticidade em um ambiente de nuvem baseado na alocação eficiente do número de *containers* para atender requisições de um ambiente *web*. A proposta foi avaliada em um ambiente simulado e os resultados mostram potencial para otimizar o processo de alocação de recursos de processamento em um ambiente de computação em nuvem.

O trabalho é estruturado da seguinte forma: a seção 2 apresenta os trabalhos relacionados. A seção 3 contém a revisão bibliográfica. A seção 4 descreve as principais características das ferramentas utilizadas, a arquitetura da solução e detalha o algoritmo de auto elasticidade desenvolvido. A seção 5 apresenta os resultados experimentais. Finalmente a seção 6 apresenta as conclusões e trabalhos futuros desta pesquisa.

2. Trabalhos Relacionados

O trabalho de [Lorido-Botrán et al. 2012], realiza uma comparação entre os diversos métodos para obter escalabilidade e auto elasticidade em um ambiente de nuvem. Estes métodos são separados em duas categorias: reativos e preditivos. As técnicas utilizadas são baseadas em aprendizado de máquina, teoria de filas, teoria de controle, análises de séries temporais, entre outros. A proposta deste trabalho pode ser descrita como reativa, pois reage a variações no tempo médio de resposta de uma aplicação, alocando ou desalocando recursos, utilizando teoria de controle para isto.

Outros trabalhos também abordam o dimensionamento elástico (auto-escalável) para ambientes de nuvem. No trabalho de [Gong et al. 2010], os autores propõem um algoritmo chamado de PRESS para predição de carga de CPU, que extrai padrões de consumo e ajusta a alocação de recursos. A abordagem dos autores utiliza dois métodos para realização de previsões em linha. O primeiro se baseia no uso de processamento de sinais e transformadas rápidas de Fourier para extração de frequências dominantes. Com essas frequências são geradas séries temporais e diversas janelas de tempo são comparadas. É gerado um índice de correlação de Pearson para as várias janelas comparadas. Caso se obtenha um índice de correlação maior que 0,85, o valor médio do uso de recursos dentro de cada posição da série temporal é utilizado para gerar uma previsão para a próxima janela e os recursos das máquinas virtuais são ajustados. Caso um padrão não seja identificado, os autores propõem uma abordagem que utiliza uma cadeia de Markov com número finito de estados para a realização das previsões.

Outro trabalho que propõe dimensionamento elástico (auto-escalável) para ambientes de nuvem é o Haven [Poddar et al. 2015]. Esta proposta utiliza ferramentas de um sistema operacional de nuvem para monitoramento de cargas de CPU e memória a que cada máquina virtual de um *pool* de balanceamento está submetido. A partir de limiares previamente estabelecidos para consumo de CPU e memória, o Haven pode instanciar novas máquinas virtuais e realizar a sua inserção em um *pool* de balanceamento de carga.

O [Google 2016b] propõe e disponibiliza uma ferramenta nativa de auto escalabilidade chamada de *Horizontal Pod Autoscaler* (HPA). Esta ferramenta trabalha escalando o ambiente a partir de limiares médios de consumo de CPU dos *containers* que atendem a determinada aplicação.

A proposta deste artigo se diferencia da abordagem PRESS na medida que não realiza predição de carga, mas sim um dimensionamento de recursos, baseado na observação do tempo de resposta de uma aplicação atrás de um balanceador de carga. Outra

diferença é que o PRESS realiza escalabilidade vertical, ou seja, realiza aumento de recursos de memória e cpu, para se ajustar a carga de trabalho. Este trabalho propõe a escalabilidade horizontal, ou seja, novas instâncias capazes de atender a carga de trabalho, são alocadas atrás de um balanceador de carga, de modo a se ajustar a variações na demanda. A escalabilidade horizontal tem a vantagem de que os recursos alocados para atender a carga de trabalho, não são limitados aos recursos físicos de uma máquina. Além disso, esta abordagem facilita a alta disponibilidade, uma vez que as demandas são atendidas por um conjunto de instâncias em paralelo.

A diferença em relação à abordagem Haven é propor um método de dimensionamento do sistema a partir da observação do tempo de resposta das requisições. Isso permite uma visibilidade global do comportamento do sistema, pois nos casos em que não haja excessos de consumo de processamento e memória, o ambiente pode se beneficiar do aumento do paralelismo no atendimento das requisições. Assim como a proposta deste artigo, o Haven também realiza escalabilidade horizontal. Outra diferença na proposta deste artigo é a infraestrutura utilizada. Os trabalhos PRESS e HAVEN trabalham com a tecnologia de máquinas virtuais e a proposta deste artigo utiliza *containers* que são considerados nas publicações recentes [Docker 2016] como mais leves por não fazerem virtualização de hardware.

Este trabalho usa o *Kubernetes* como orquestrador de *containers*, porém propõe outro algoritmo de auto escalabilidade, baseado no tempo médio de resposta das aplicações. Outra diferença deste trabalho, é onde são feitas as medidas que decidem se o ambiente deve ser escalado. Enquanto o HPA realiza medidas de consumo de CPU nos próprios *containers*, a proposta deste trabalho realiza medidas externas aos *containers*, no caso são realizadas medidas do tempo médio de resposta das aplicações do ponto de vista do balanceador de carga. Esta abordagem traz a vantagem de poder observar a performance do ambiente, independente do nível de utilização dos recursos dos containers. Na Seção 5 é feita uma comparação entre o HPA e a solução proposta neste trabalho.

3. Revisão Bibliográfica

3.1. Containers

Container é uma tecnologia para a criação de instâncias de processamento separadas ou isoladas que permitem a virtualização no nível do sistema operacional ao disponibilizar porções protegidas de processamento. Dois *containers* rodando no mesmo sistema, não sabem que estão compartilhando recursos, pois tem sua própria abstração de camada de rede, memória e processos [Docker 2016].

Os *containers* apresentam uma maior portabilidade que as máquinas virtuais, ao serem configurados de forma genérica para qualquer sistema operacional baseado em Linux. A virtualização via *hypervisors* consome mais recursos do que a virtualização por *containers* dado que os últimos são executados em sistemas operacionais que rodam em espaços isolados entre si. Se um *container* não está executando nenhuma tarefa, ele não está consumindo recursos no servidor [Docker 2016]. Além disto os *containers* apresentam um grande dinamismo para serem criados e destruídos, dado que apenas tem que iniciar ou destruir processos em seu espaço isolado. Portanto verifica-se que existem vantagens em se utilizar a tecnologia de *containers*. Esta foi a tecnologia escolhida para atender o processamento das cargas de trabalho *http* nesta solução.

3.2. Serviços Web e Balanceadores de Carga

Os sistemas web podem sofrer com a alta variabilidade da carga demandada, muitas vezes de forma inesperada. Neste caso, a infraestrutura que hospeda este tipo de serviço, deve estar preparada para o atendimento da demanda com alto desempenho, escalabilidade e com alta disponibilidade. Uma abordagem comum para disponibilizar estas características é o uso de arquiteturas distribuídas que podem rotear as requisições para diversos servidores, de forma transparente ao usuário. Esta abordagem além de melhorar a performance e a disponibilidade dos serviços, pode aumentar sua escalabilidade ao se adicionar e remover membros no *cluster*. [F5 2012]

Porém vários desafios se impõem para que um *cluster* distribuído de servidores possam funcionar de forma eficiente como se fosse um único servidor. Esses desafios vão desde o roteamento das requisições para os membros do *cluster*, métodos para escolha do membro que receberá a demanda e manutenção do estado da conexão. [Gilly et al. 2011]

No balanceamento na camada de transporte, as requisições são distribuídas entre os membros do *cluster*. O balanceador distribui as conexões de clientes que conhecem o endereço IP do *cluster*, entre os diversos servidores que efetivamente respondem as requisições. Neste caso, como a distribuição é feita com base na camada 4, o balanceador escolhe um servidor, sem considerar o conteúdo ou o tipo da requisição. [Anicas 2014]

3.3. Controladores PID

Os controladores PID (Proporcional-Integral-Derivativo) são algoritmos de controle muito utilizados na industria. Os controladores PID possuem três coeficientes: proporcional, integral e derivativo. Esses coeficientes são variados de forma a se obter a resposta de controle ideal desejada para um processo.

Um controlador PID trabalha dentro de um sistema em malha fechada, onde é possível a leitura do estado atual de determinada variável que se deseja controlar, e de acordo com seu valor, uma ação é executada de modo que a variável chegue, e tente permanecer no nível desejado (ainda considerando distúrbios externos), nas próximas iterações de tempo [Instruments 2015].

Sendo assim, o PID deve ler o estado atual da variável e calcular a resposta da saída do atuador, por meio do cálculo dos componentes proporcional, integral e derivativo e então somar os três para calcular a saída. O componente proporcional depende da diferença entre o valor desejado (*setpoint*) e o valor atual da variável. Esta diferença é referida como erro. O componente integral soma o termo de erro ao longo do tempo. A resposta derivativa é proporcional a taxa de variação da variável do processo.

Para que o controlador PID produza os ajustes necessários ao sistema, os parâmetros de ganho K_p, K_i e K_d devem ser ajustados. Existem diversos métodos de ajuste destes parâmetros, como por exemplo O método manual (*guess and check*) e o método Ziegler-Nichols [Instruments 2015].

No método manual, os ganhos de cada um dos componentes é ajustado usando tentativa e erro. Para isso, ao ajustar os parâmetros, devem ser conhecidos os efeitos que cada parâmetro provoca na saída do controlador. Neste método, os termos K_i e K_d são ajustados para zero, e o termo K_p é aumentado até que a saída do ciclo comece a oscilar. A partir daí aumenta-se lentamente o termo K_i para reduzir o erro estacionário. Neste

ponto inicia-se o incremento do termo K_d, de modo a diminuir as oscilações na saída do ciclo [Instruments 2015]. A discussão sobre outros métodos de ajuste dos parâmetros foge ao escopo deste trabalho, já que foi utilizado o método de ajuste manual.

4. Solução Proposta

O objetivo deste trabalho, é a provisão de um ambiente elástico de nuvem privada, auto escalável e com balanceamento de carga, para hospedagem de sistemas Web.

A solução propõe um ambiente de *cluster* baseado em *containers*, e um método de auto elasticidade que reage a aumentos no tempo de resposta do sistema, de modo a mantê-lo dentro de um limite. Para isso é utilizado um sistema de malha fechada com um controlador PID, que reage a variações no tempo de resposta do sistema aumentando ou diminuindo o número de *containers* no *cluster* de balanceamento de carga capaz de responder as requisições.

4.1. Ferramentas Utilizadas para Implementação

4.2. Docker

O *Docker* começou como projeto da empresa de PaaS (*Platform as a Service*) dotCloud em 2013 [Martin 2015], propondo ser um integrador e facilitador para adoção da tecnologia de *containers* em produção e em larga escala. O Docker utiliza o LXC (linux *containers*) para isolar os containers do servidor, criando isolamento de processos, rede e privilégios. A limitação e contabilização de recursos (CPU, memória, espaço em disco e E/S) é feita por meio dos *cgroups*. Além disso a utilização do sistema de arquivos é feita de forma eficiente pois se baseia em *copy-on-write*, que permite que as alterações em um *container* sejam simplesmente uma atualização diferencial da imagem anterior.

Uma das grandes vantagens do *Docker* é a habilidade de encontrar, baixar e iniciar imagens de *containers* que foram criados por outros programadores de forma muito rápida e prática. O *Docker* viabiliza o uso da tecnologia de *containers* de forma prática, razão pela qual é utilizado nesta solução.

4.3. Kubernetes

Kubernetes é um sistema desenvolvido pelo Google [Google 2016a], e disponibilizado para a comunidade, que visa gerenciar o ciclo de vida de *containers* nos nós de um *cluster*.

Desta forma, o *Kubernetes* é um orquestrador de *containers*, sendo capaz de agendar o lançamento de *containers* entre os nós do *cluster*, assim como fazer o controle de admissão dos *containers*, balanceamento de recursos e prover escalabilidade ao ambiente. O *Kubernetes* também provê funcionalidades como descoberta de serviços entre os *containers*, publicação do serviço para acessos a partir de entidades fora do *cluster* e balanceamento de carga entre os *containers* [Google 2016a]. Estas funcionalidades foram determinantes para a decisão de usar o Kubernetes na solução.

A infraestrutura de um *Kubernetes* é composta por nós do tipo *Master* que controlam os nós do tipo *Workers*, e estes rodam os *containers* . Todas as configurações do *cluster* ficam armazenadas de em um repositório de configurações distribuído, o *Etcd*. Os *PODs* são a unidade básica com a qual o *Kubernetes* trabalha. Os *containers* são agrupados em *PODs* e estes geralmente representam uma aplicação. Estes são criados por meio

dos Replication Controllers que são utilizados para definir PODs que podem ser escalados horizontalmente. Os Replication Controllers também são responsáveis por manter o número desejado de PODs ativos no *cluster*.

4.4. Apache Spark, Flume, HAproxy e Redis

O *Apache Spark* é uma ferramenta de processamento distribuído, ideal para processamento de grandes bases de dados. Foi desenvolvido pela AMPLab (UC Berkeley), e realiza processamento de dados em memória. Sua estrutura básica de abstração são os *RDDs (Resilient Distributed DataSets)*, que são coleções de elementos que podem sofrer operações em paralelo, sendo possível a geração de novos RDDs a partir de transformações como *map, reduce, filter e join* em *RDDs*. [Spark 2015a] Esta ferramenta foi escolhida para integrar a solução pois com ela é possível realizar processamento de grandes bases em formato de texto, de forma escalável.

O *Apache Spark* oferece uma API chamada *Spark Streaming*, que permite o processamento em tempo real de dados, por meio da criação de estruturas chamadas *DStream* (*discretized stream*), que são representados como sequências de *RDDs*. A criação dos *DStreams* é feita por meio da classe *StreamingContext*, onde se configura a duração de cada janela de *DStreams* [Spark 2015b]. Neste trabalho, a duração de cada janela foi configurada como 5 segundos, o que é suficiente para a geração da série temporal em tempo real utilizada pela solução.

Neste contexto o *Spark Streaming* é utilizado para processar os logs do balanceador de carga, coletando o tempo de resposta de cada uma das requisições que o sistema atende, em tempo real. Esta informação do tempo de resposta é armazenada em formato de série temporal no servidor *Redis*, para uso no algoritmo de auto escalabilidade proposto neste artigo.

O recebimento das entradas de log em texto do balanceador de carga pelo *Spark*, ocorre por meio da ferramenta *Flume*. O *Flume* é um serviço que agrega, coleta e move grandes volumes de fluxo de dados. Para seu funcionamento é criada uma fonte que recebe os dados de interesse. Essa fonte (*Source*) é conectada a um canal (*Channel*), por onde os dados trafegarão em direção a um *Sink* [Flume 2016].

Portanto, a função desta ferramenta na solução, é realizar o envio, em tempo real, dos logs de acesso do balanceador carga, por meio da criação de uma fonte do tipo *Syslog*, que recebe os logs do balanceador de carga, e um canal de memória que carrega os dados da fonte em memória. A partir daí, o *Spark* consome este fluxo de dados por meio de um *Sink* do tipo *Avro*, que trafega pela rede.

O balanceador de carga utilizado na solução, é o *HAproxy*, que pode atuar como balanceador de camada 4 ou 7, terminador SSL, proxy reverso entre outros [Tarreau 2015]. Atualmente, este é o balanceador utilizado por *sites* como *Reddit, Stack Overflow/Server Fault, Instagram* entre outros, além de ter sido escolhido como balanceador da nuvem da Red Hat (*OpenShift*).

Esta adoção pela comunidade foi uma das razões de se ter escolhido esta ferramenta para compor a solução. Além disso, o *HAproxy* disponibiliza um *log* com informações viabilizam a implementação da proposta deste trabalho.

O log do *HAproxy* apresenta o seguinte formato:

May 18 06:24:25 10.125.7.229 haproxy[1078]: 10.125.8.252:43839 [18/May/2016:06:24:24.988] cherrypy cherrypy/10.125.7.227 0/0/2/26/28 200 169 - - —- 1/1/1/0/0 0/0 "GET /generate HTTP/1.0"

Nesta linha pode-se observar informações como o tempo de espera na fila do servidor de aplicação, método e código da resposta *http* entre outras. A parte com *0/0/2/26/28* contém as informações : *Tq '/' Tw '/' Tc '/' Tr '/' Tt*, onde O *Tr* é o tempo em milisegundos que o balanceador demora para receber uma resposta completa de uma requisição *http* ao servidor [Tarreau 2015]. Portanto este valor representa o tempo total de processamento da requisição pelo *container*.

No contexto deste trabalho o *SparkStreaming*, processa as entradas do *log*, separa o campo *Tr* e o armazena no banco *Redis*, no formato de uma série temporal. O *SparkStreaming* também é responsável por converter o formato da data de cada linha, para o número de segundos desde 0 hora de cada dia, para suportar a criação da série temporal.

O *Redis* é utilizado para armazenar a série temporal, pois consegue prover baixa latência tanto para escrita como para leitura, por manter os dados como estruturas em memória [Redis 2015].

A integração da solução com o *Redis*, ocorre por meio da biblioteca *Kairos*, que cria uma estrutura para armazenamento de séries temporais em bancos como *Redis*, *Mongo*, *SQL* ou *Cassandra* [Kairos 2015].

Esta biblioteca provê facilidades, como configuração do número de entradas a serem mantidas no banco e tamanho da unidade mínima de tempo de interesse da série. No caso deste trabalho são mantidos armazenados na série a cada momento os dados dos últimos 600 segundos, o que é suficiente para operação do algoritmo.

O *Kairos* também permite o cálculo de parâmetros estatísticos da série, utilizando tamanho de janelas configuráveis de tempo. Isso permite que seja calculada a média de tempo de resposta de uma aplicação nos últimos dois minutos por exemplo. A unidade mínima de tempo configurada nesta solução é 1 segundo. Isso permite boa flexibilidade para configuração dos tamanhos das janelas de tempo para os cálculos estatísticos, além de permitir a geração de gráficos com boa resolução de tempo para monitoramento e avaliação da solução.

4.5. Arquitetura da Solução

A arquitetura proposta é apresentada na Figura 1, em que é utilizado um controlador PID para manter o tempo médio de resposta de determinada aplicação próxima de um determinado limiar. A arquitetura proposta, chamada de PAS (PID *based Autoscaler*) opera com base na seguinte sequência:

1. É estabelecido um limiar (*setpoint*) de tempo médio de resposta desejado para as requisições. O monitor recebe o tempo de resposta das requisições que chegam no balanceador;
2. O monitor envia o tempo médio de resposta (ex: média dos últimos 200 segundos) para que o dimensionador PID calcule o número de *containers* necessários para atingir ou permanecer no *setpoint*. A média dos últimos 200 segundos foi definida, pois verificou-se experimentalmente que é uma quantidade suficiente para

que valores muito diferentes da média não exerçam influência negativa no dimensionamento, assim como permite que o sistema seja capaz de reagir rapidamente a mudanças no padrão da média do tempo de resposta.

3. O Dimensionador PID executa o algoritmo 1 e informa o número desejado de *containers* ao *Kubernetes*. O número atual de *containers* é pesquisado usando a ferramenta do kubernetes *kubectl*, e o tempo médio de resposta do *cluster* é determinado usando a série temporal presente no *Redis*.
4. O *Kubernetes* cria ou remove novos *containers*, além de garantir que o ambiente permanecerá com o número desejado de *containers* até a próxima rodada do algoritmo (ex: depois de 10 segundos). Este valor de 10 segundos foi definido pois verificou-se que é suficiente para que o *Kubernetes* inicie novos *containers* e estes passem a responder as requisições no *cluster* de balanceamento.

Algoritmo 1: PAS

Entrada: Tempo médio de resposta do cluster, Número atual de *containers*
Saída: Número desejado de Containers

início
  Leia o limiar de tempo médio de resposta desejado para as requisições: $t\_ms\_desejado$
  Leia o número atual de *containers*: $n\_containers\_atual$
  Leia tempo de resposta médio do cluster em ms: t_ms_atual
  Calcule o erro: $e(t)$ = $t\_ms\_desejado$ - t_ms_atual
  Calcule a saída o controlador pid:

$$u(t) = K_p e(t) + K_i \int_0^t e(t)\delta t + K_d \frac{d}{dt} e(t) \quad (1)$$

  $n\_desejado\_containers$=$n\_containers\_atual$ + $u(t)$
fim
retorna *n_desejado_containers*

4.6. Operação da solução

Para viabilizar a operação do algoritmo que trabalha com dados dinâmicos, recebidos em tempo real, utiliza-se o fluxo de processamento mostrado na Figura 2. O *Haproxy* atua como balanceador de carga da solução. Seus *logs* são enviados ao *Flume* que envia os dados em um canal de memória e os envia ao *Spark Streaming*, desta forma disponibilizando as informações necessárias para operação do algoritmo de auto escalabilidade.

O *Haproxy* atua balanceando as requisições no modo *round robin* para os servidores *Docker/Kubernetes* que hospedam os *containers*. Quando o nó *Docker/Kubernetes* recebe a requisição, o serviço *Kubernetes proxy* é responsável por balancear a carga entre os *containers* de cada um dos servidores.

Desta forma, acontecem dois níveis de balanceamento, um entre os nós *Docker/Kubernetes* em que o responsável pelo balanceamento é o *Haproxy*, e outro internamente dentro do nó *Docker/Kubernetes*, onde o serviço *Kubernetes proxy* é o responsável pelo balanceamento.

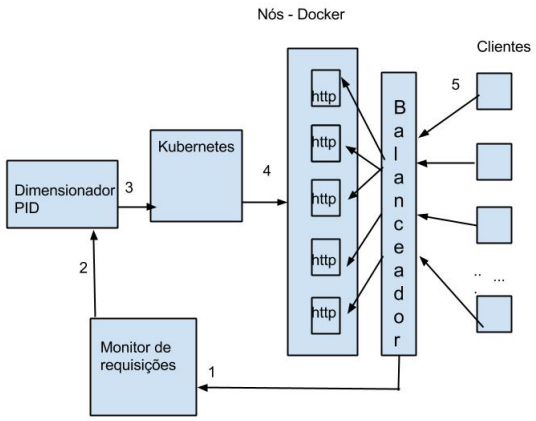

Figura 1. Arquitetura da Solução

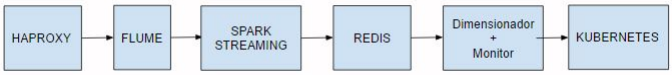

Figura 2. Fluxo de processamento entre o conjunto de ferramentas utilizadas

5. Resultados Experimentais

5.1. Ambiente

Para avaliação da solução foi configurado um *cluster Kubernetes* v1.1.2 no sistema operacional *CoreOS* (899.6.0 (2016-02-02)), virtualizado em *VMWare ESXi* 5.5.0. Este ambiente foi configurado para validação da solução. Um ambiente de produção poderia se beneficiar mais se o sistema *CoreOS* fosse instalado diretamente em máquinas físicas, pois seria eliminada a camada de virtualização.

O *cluster* foi constituído com os seguintes componentes: 1 nó *master* (4 vCPUs, 6 GB de RAM), 1 nó *etcd* (4 vCPUs, 6 GB de RAM) e 4 nós *workers* (4 vCPUs, 6 GB de RAM). As máquinas virtuais (*VMWare ESXi* 5.5.0) foram instaladas no sistema operacional Ubuntu 14.04.3 LTS, com as seguintes configurações e ferramentas: 1 nó *haproxy* 1.5.4 (4 vCPUs, 4G GB de RAM), 1 nó *Spark* 1.5.2 + *Redis* 2.8.4 (2 vCPU, 10 GB de RAM) e 1 nó *Flume 1.7.0* + dimensionador (2 vCPU, 4 GB de RAM).

Para validação do ambiente foi gerada uma imagem de *container* que roda o servidor web *cherrypy* 5.1.0. Foi configurado um *link* neste servidor. O *link* gera um *array* de tamanho aleatório entre 1.000 e 10.000 elementos a cada requisição.

A publicação do serviço no *Kubernetes* foi feita por meio da criação de um *Replicaction Controller* e a configuração de um serviço do tipo *NodePort*. O balanceador *Haproxy* foi configurado para balancear as requisições entre os nós *Docker/Kubernetes* usando os endereços IP dos nós e as portas publicadas pelo serviço do tipo *NodePort*. Cada *container* teve seus recursos de processamento e memória limitados a 18 MB de

memória RAM e 24 ms de CPU.

O algoritmo PID utilizou os seguintes parâmetros: *Kp*=0.016, *Ki*=0.000012 e *Kd*=0.096, que foram definidos após vários testes com a carga de trabalho, e aplicação do método de ajuste manual, ou *guess and check*, conforme descrito na seção 3.3

5.2. Carga de Trabalho

A geração de carga de trabalho foi feita usando a ferramenta *ab (apache bench)*. Para geração de uma carga com um perfil de requisições por segundo realista, foi coletado um conjunto de acessos do portal da transparência (www.transparencia.gov.br), entre os dias 25/05/2015 a 25/06/2015. A série temporal capturada na escala de 1 segundo, representa o número de acessos naquele intervalo de tempo. A série foi caracterizada com o método Kettani-Gubner [Kettani et al. 2002]. A autossimilaridade e longa dependência da série foi confirmada com o parâmetro de Hurst *H*=0,87. nas escalas de 1 segundo, 100 segundos e 600 segundos.

Esta série é utilizada para gerar a cada segundo requisições simultâneas direcionadas ao endereço IP do balanceador de carga que distribui as requisições nos nós do *cluster Kubernetes*. A intensidade da carga foi ajustada em 3 níveis, multiplicando a série temporal por 1, 1.5 e 2 e preservando o mesmo índice de autossimilaridade. Estas cargas são referidas respectivamente nos experimentos como carga_1, carga_1.5 e carga_2.

Uma amostra da série obtida utilizada para caracterizar a carga de trabalho, relativa a 2000 segundos pode ser vista na Figura 3.

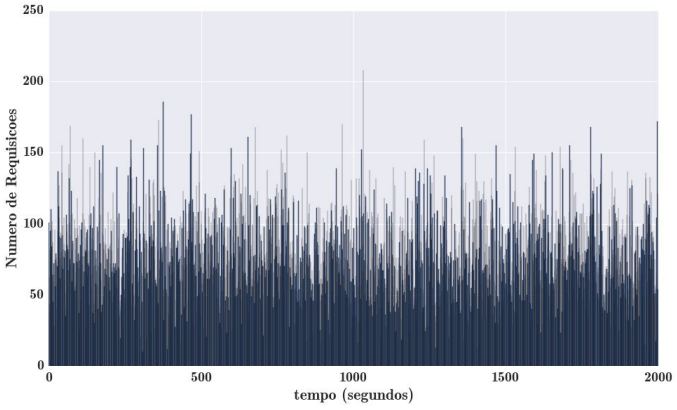

Figura 3. Amostra da carga utilizada nos testes, *H*=0,87

5.3. Cenários de avaliação

5.3.1. Cenário 1

No primeiro cenário o limiar de tempo de resposta no balanceador (*setpoint*) foi estabelecido em 50 ms e foram aplicadas as cargas carga_1 e carga_1.5. A Figura 4 mostra o tempo de resposta do sistema ao ser submetido à carga_1. O gráfico mostra o ajuste provocado pelo PID e a estabilidade alcançada próximo do *setpoint* de 50 ms.

A Figura 4 mostra também a alocação de *containers*. Observa-se que o número de *containers* no início do experimento é igual a 2. O sistema alocou o número necessário

para que o tempo médio de resposta mostrado na Figura 4 atingisse o *setpoint*. No final da execução estavam alocados 26 *containers*.

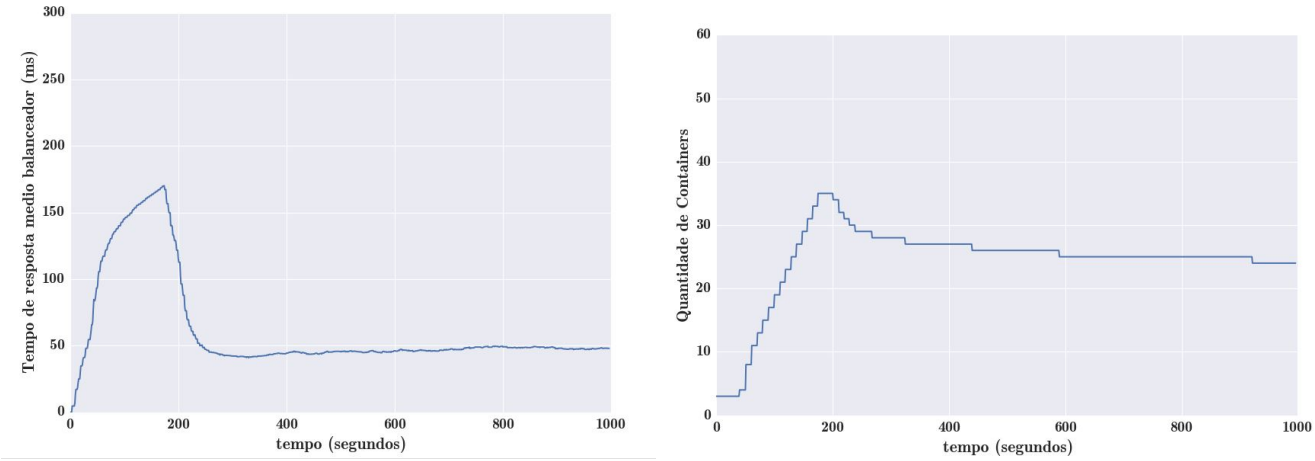

Figura 4. Carga_1, *setpoint* 50 ms

A Figura 5 mostra o tempo médio de resposta com o sistema sendo submetido a carga_1.5 e o *setpoint* mantido em 50 ms. A Figura 5 também mostra a alocação de *containers* durante este teste. Observa-se que a alocação de *containers* foi sendo ajustada de modo a manter o tempo de resposta médio do sistema próximo ao *setpoint*. No final da execução estavam alocados 52 *containers*. Neste caso, o sistema submetido a uma carga maior, alocou mais containers para manter o tempo de resposta dentro do limiar definido.

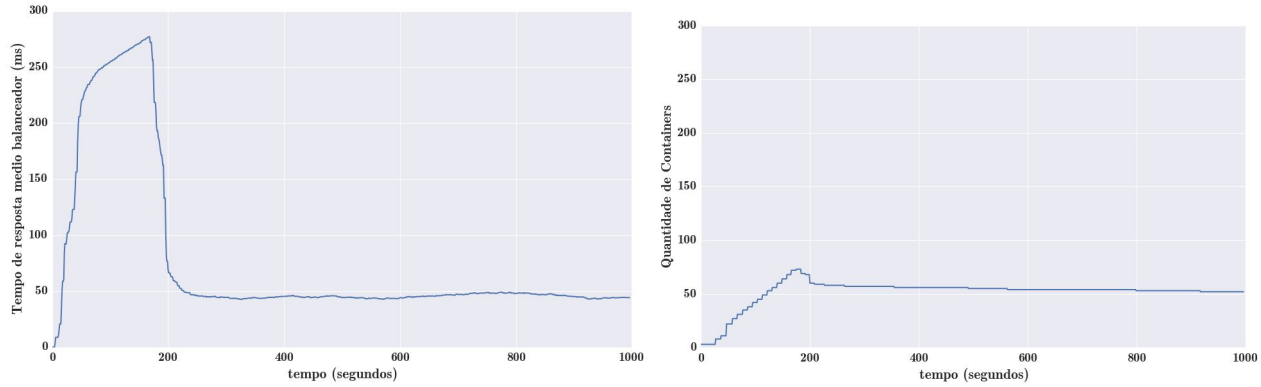

Figura 5. Carga_1.5, *setpoint* 50 ms

5.3.2. Cenário 2

O Cenário 2, compara o comportamento do algoritmo de escalabilidade proposto neste trabalho com o utilizado nativamente pelo *Kubernetes*, denominado de HPA [Google 2016b].

A comparação é feita da seguinte forma: o sistema configurado com o HPA_80 (configurado para escalar quando o consumo médio dos *containers* estiver acima de 80 %) será exposto às cargas carga_1, carga_1.5 e carga_2 durante 1000 segundos. No fim

da avaliação o tempo médio de espera das requisições na camada de aplicação (ponto de vista dos clientes) é observado.

A partir destes dados, foi definido um *setpoint* para ser utilizado com o algoritmo proposto (PAS) que permita comparar com o tempo de resposta próximo ao de referência do HPA. Os resultados serão comparados, verificando a quantidade média de *containers* utilizados durante os experimentos e os tempos de resposta obtidos na camada de aplicação dos clientes.

Como pode ser observado na Figura 6 os tempos médios de resposta na camada de aplicação com o algoritmo HPA para cada carga, foram: 66.18 ms para a carga_1, 110.12 ms para a carga_1.5 e 144.01 ms para a carga_2.

Para efeitos comparativos, os *setpoints* do PAS, foram configurados para entregar um tempo de resposta próximo aos obtidos com o HPA. Para isto foram configurados *setpoints* um pouco abaixo dos observados no HPA, dado que o *setpoint* é controlado no balanceador, portanto o tempo observado na camada de aplicação dos clientes deve ser maior. Os valores definidos para os *setpoints* são os seguintes: 50 ms (PAS_50) para a carga_1, 80 ms (PAS_80) para a carga_1.5 e 100 ms (PAS_100) para a carga_2.

As Figuras 6 e 7 mostram que para um tempo de resposta aproximado ao do HPA, o sistema PAS, alocou menos *containers* do que o HPA. O comparativo de alocação de *containers* mostra que: PAS_50 alocou 44.02 % do que foi alocado pelo HPA_80, PAS_80 alocou 36.07 % do que foi alocado pelo HPA_80, para a carga_1.5 e PAS_100 alocou 12.72 % do que foi alocado pelo HPA_80, para a carga_2.

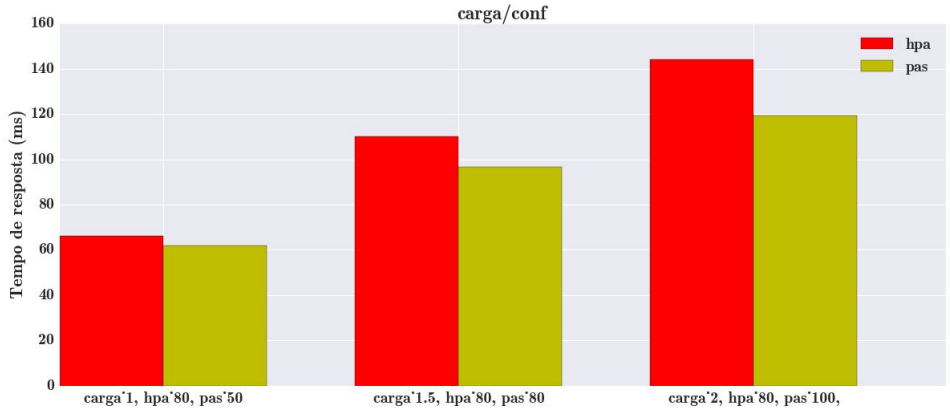

Figura 6. Comparativo entre HPA e PAS (tempo médio de resposta do sistema)

5.4. Análise dos Resultados

Os resultados experimentais mostram que nos cenários avaliados, o algoritmo PAS otimiza a alocação do número de *containers* para manter os valores de *setpoints* dentro de um limiar estabelecido. O HPA aloca um número maior de *containers* para alcançar limiares equivalentes de tempo de resposta para as requisições na camada de aplicação. Este resultado evidencia que a alocação de um número maior de *containers* pode aumentar a complexidade do tempo de balanceamento de carga e não necessariamente produzir melhores tempos de resposta.

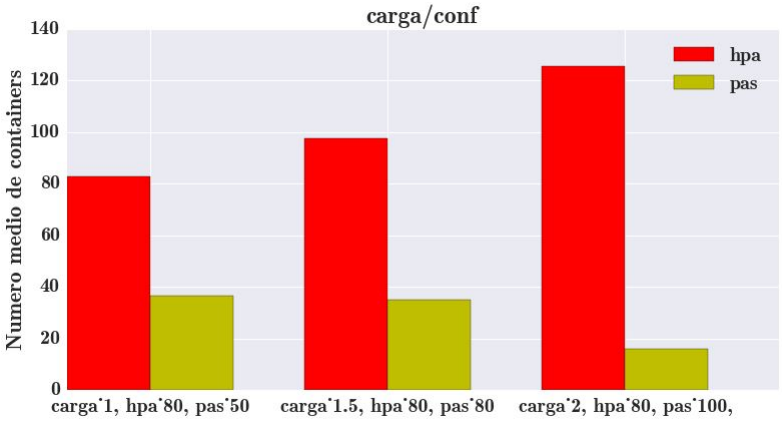

Figura 7. Comparativo entre HPA e PAS (número médio de containers)

Os resultados obtidos mostram que o algoritmo PAS, proposto neste trabalho apresenta potencial para promover a otimização de alocação do número de *containers* em um ambiente de computação em nuvem.

6. Conclusão e Trabalhos Futuros

Este trabalho apresentou o algoritmo PAS (PID *based Autoscaler*) e os resultados mostram um potencial para prover auto elasticidade a um ambiente de computação em nuvem baseado em *containers*. A comparação com a ferramenta de auto escalabilidade nativa do *Kubernetes*, mostra uma maior eficiência da solução proposta para alocação da quantidade de *containers* com o objetivo de cumprir limiares pré estabelecidos de tempo de resposta na execução de requisições para aplicações Web.

O uso de ferramentas como *spark* e *flume* possibilita que a arquitetura seja escalável, podendo vir a ser utilizada por *sites* com servidores com grande número de acessos, em que a série temporal resultante do número de requisições em diversas escalas de tempo tende a ter um alto grau de autossimilaridade.

A proposta possibilita um uso eficiente de recursos, o que pode permitir um menor custo de operação da infraestrutura instalada e dos serviços oferecidos em nuvem.

Os trabalhos futuros pretendem validar os resultados experimentais em um maior número de cenários e confronta-los com medições em ambientes de produção que permitam expandir a aplicação desta pesquisa para ambientes reais de computação em nuvem.

Referências

Anicas, M. (2014). Mitchel Anicas an introduction to haproxy and load balancing concepts. `https://www.digitalocean.com/community/tutorials/an-introduction-to-haproxy-and-load-balancing-concepts`. [Accessado em 03/04/2016].

Docker (2016). Docker the definitive guide to docker containers. `https://www.Docker.com/sites/default/files/WP-%20Definitive%20Guide%20To%20Containers.pdf`. [Accessado em 03/04/2016].

F5 (2012). F5 load balancing 101 - nuts and bolts. https://f5.com/resources/white-papers/load-balancing-101-nuts-and-bolts. [Accessado em 03/04/2016].

Flume (2016). Flume flume user guide. https://flume.apache.org/FlumeUserGuide.html. [Accessado em 03/04/2016].

Gilly, K., Juiz, C., and Puigjaner, R. (2011). An up-to-date survey in web load balancing. *World Wide Web*, 14(2):105–131.

Gong, Z., Gu, X., and Wilkes, J. (2010). Press: Predictive elastic resource scaling for cloud systems. In *Network and Service Management (CNSM), 2010 International Conference on*, pages 9–16. IEEE.

Google (2016a). Google container cluster manager from google. https://github.com/kubernetes/kubernetes. [Accessado em 03/04/2016].

Google (2016b). Google horizontal pod autoscaler. https://github.com/kubernetes/kubernetes/blob/release-1.2/docs/design/horizontal-pod-autoscaler.md. [Accessado em 03/04/2016].

Instruments, N. (2015). National Instruments explicando a teoria pid. http://www.ni.com/white-paper/3782/pt/. [Accessado em 20/08/2015].

Kairos (2015). Kairos time series data storage in redis, mongo, sql and cassandra. https://pypi.python.org/pypi/kairos. [Accessado em 03/04/2016].

Kettani, H., Gubner, J., et al. (2002). A novel approach to the estimation of the hurst parameter in self-similar traffic. In *Local Computer Networks, 2002. Proceedings. LCN 2002. 27th Annual IEEE Conference on*, pages 160–165. IEEE.

Lorido-Botrán, T., Miguel-Alonso, J., and Lozano, J. A. (2012). Auto-scaling techniques for elastic applications in cloud environments. *Department of Computer Architecture and Technology, University of Basque Country, Tech. Rep. EHU-KAT-IK-09*, 12:2012.

Martin, N. (2015). Nick Martin a brief history of docker containers' overnight success. http://searchservervirtualization.techtarget.com/feature/A-brief-history-of-Docker-Containers-overnight-success. [Accessado em 03/04/2016].

Poddar, R., Vishnoi, A., and Mann, V. (2015). Haven: Holistic load balancing and auto scaling in the cloud.

Redis (2015). Redis redis documentation. http://redis.io/documentation. [Accessado em 03/04/2016].

Spark (2015a). Spark spark programming guide. https://spark.apache.org/docs/1.5.2/programming-guide.html. [Accessado em 03/04/2016].

Spark (2015b). Spark spark streaming programming guide. https://spark.apache.org/docs/1.5.2/streaming-programming-guide.html. [Accessado em 03/04/2016].

Tarreau, W. (2015). HAProxy haproxy configuration manual. http://www.haproxy.org/download/1.5/doc/configuration.txt. [Accessado em 03/04/2016].

Anexo II

Artigo publicado no Congresso CLEI 2016 - Vinas del Mar/Chile [40]

A Mechanism of Auto Elasticity Based on Response Times for Cloud Computer Enviroments and Autossimilar Workload

Marcelo Cerqueira de Abranches
Departamento de Ciência da Computação
Universidade de Brasília
Brasília, Distrito Federal (61) 3107-6737/3658
Email: marcelo.abranches@cgu.gov.br

Priscila Solis
Departamento de Ciência da Computação
Universidade de Brasília
Brasília, Distrito Federal (61) 3107-6737/3658
Email: pris@cic.unb.br

***Abstract*—This paper proposes a cloud computing architecture based on containers and an algorithm for auto elasticity based on the response time requirements in a Web system. The proposed algorithm promotes an efficient allocation of containers to achieve expected response time in processing requests. This proposal was evaluated with a real time series obtained from a web system hosted by Controladoria-Geral da União (CGU). The results show that the proposed algorithm achieves the expected response times allocating a lower number of containers than other related proposals.**

***Keywords*—cloud computing, containers, cluster, performance, auto scaling, elasticity, autossimilarity.**

I. Introdução

A pesquisa nos ambientes de computação em nuvem apresenta como um dos maiores pontos de interesse algoritmos de auto escalabilidade [16] e balanceamento de carga. Como em qualquer processo de dimensionamento e planejamento, um dos pontos cruciais para esta tarefa é a correta caracterização da carga de trabalho e a otimização no uso de recursos, os quais devem atender um conjunto de requerimentos de desempenho das aplicações.

Diversos trabalhos recentes abordam o dimensionamento elástico e a auto escalabilidade em ambientes de computação em nuvem [16], [14], [8] e [10]. As soluções propostas nestes trabalhos são na sua maioria baseadas em heurísticas, teoria de filas, análises de séries temporais, reação a aumento de consumo de recursos entre outros.

Este trabalho propõe um algoritmo para promover a auto elasticidade em um ambiente de nuvem baseado na alocação eficiente do número de *containers* para atender requisições de um ambiente *web*. A proposta foi avaliada em um ambiente simulado e os resultados mostram potencial para otimizar o processo de alocação de recursos de processamento em um ambiente de computação em nuvem.

O trabalho é estruturado da seguinte forma: a seção 2 apresenta os trabalhos relacionados. A seção 3 contém a revisão bibliográfica. A seção 4 descreve as principais características das ferramentas utilizadas, a arquitetura da solução e detalha o algoritmo de auto elasticidade desenvolvido. A seção 5 apresenta os resultados experimentais. Finalmente a seção 6 apresenta as conclusões e trabalhos futuros desta pesquisa.

978-1-5090-1633-4/16/$31.00

II. Trabalhos Relacionados

O trabalho de [14], realiza uma comparação entre os diversos métodos para obter escalabilidade e auto elasticidade em um ambiente de nuvem. Estes métodos são separados em duas categorias: reativos e preditivos. As técnicas utilizadas são baseadas em aprendizado de máquina, teoria de filas, teoria de controle, análises de séries temporais, entre outros. A proposta deste trabalho pode ser descrita como reativa, pois reage a variações no tempo médio de resposta de uma aplicação, alocando ou desalocando recursos, utilizando teoria de controle para isto.

Outros trabalhos também abordam o dimensionamento elástico (auto-escalável) para ambientes de nuvem. No trabalho de [8], os autores propõem um algoritmo chamado de PRESS para predição de carga de CPU, que extrai padrões de consumo e ajusta a alocação de recursos. A abordagem dos autores utiliza dois métodos para realização de previsões em linha. O primeiro se baseia no uso de processamento de sinais e transformadas rápidas de Fourier para extração de frequências dominantes. Com essas frequências são geradas séries temporais e diversas janelas de tempo são comparadas. É gerado um índice de correlação de Pearson para as várias janelas comparadas. Caso se obtenha um índice de correlação maior que 0,85, o valor médio do uso de recursos dentro de cada posição da série temporal é utilizado para gerar uma previsão para a próxima janela e os recursos das máquinas virtuais são ajustados. Caso um padrão não seja identificado, os autores propõem uma abordagem que utiliza uma cadeia de Markov com número finito de estados para a realização das previsões.

Outro trabalho que propõe dimensionamento elástico (auto-escalável) para ambientes de nuvem é o Haven [16]. Esta proposta utiliza ferramentas de um sistema operacional de nuvem para monitoramento de cargas de CPU e memória a que cada máquina virtual de um *pool* de balanceamento está submetido. A partir de limiares previamente estabelecidos para consumo de CPU e memória, o Haven pode instanciar novas máquinas virtuais e realizar a sua inserção em um *pool* de balanceamento de carga.

O [10] propõe e disponibiliza uma ferramenta nativa de auto escalabilidade chamada de *Horizontal Pod Autoscaler* (HPA). Esta ferramenta trabalha escalando o ambiente a partir de limiares médios de consumo de CPU dos *containers* que atendem a determinada aplicação.

A proposta deste artigo se diferencia da abordagem PRESS na medida que não realiza predição de carga, mas sim um dimensionamento de recursos, baseado na observação do tempo de resposta de uma aplicação atrás de um balanceador de carga. Outra diferença é que o PRESS realiza escalabilidade vertical, ou seja, realiza aumento de recursos de memória e cpu, para se ajustar a carga de trabalho. Este trabalho propõe a escalabilidade horizontal, ou seja, novas instâncias capazes de atender a carga de trabalho, são alocadas atrás de um balanceador de carga, de modo a se ajustar a variações na demanda. A escalabilidade horizontal tem a vantagem de que os recursos alocados para atender a carga de trabalho, não são limitados aos recursos físicos de uma máquina. Além disso, esta abordagem facilita a alta disponibilidade, uma vez que as demandas são atendidas por um conjunto de instâncias em paralelo.

A diferença em relação à abordagem Haven é propor um método de dimensionamento do sistema a partir da observação do tempo de resposta das requisições. Isso permite uma visibilidade global do comportamento do sistema, pois nos casos em que não haja excessos de consumo de processamento e memória, o ambiente pode se beneficiar do aumento do paralelismo no atendimento das requisições. Assim como a proposta deste artigo, o Haven também realiza escalabilidade horizontal. Outra diferença na proposta deste artigo é a infraestrutura utilizada. Os trabalhos PRESS e HAVEN trabalham com a tecnologia de máquinas virtuais e a proposta deste artigo utiliza *containers* que são considerados nas publicações recentes [4] como mais leves por não fazerem virtualização de hardware.

Este trabalho usa o *Kubernetes* como orquestrador de *containers*, porém propõe outro algoritmo de auto escalabilidade, baseado no tempo médio de resposta das aplicações. Outra diferença deste trabalho, é onde são feitas as medidas que decidem se o ambiente deve ser escalado. Enquanto o HPA realiza medidas de consumo de CPU nos próprios *containers*, a proposta deste trabalho realiza medidas externas aos *containers*, no caso são realizadas medidas do tempo médio de resposta das aplicações do ponto de vista do balanceador de carga. Esta abordagem traz a vantagem de poder observar a performance do ambiente, independente do nível de utilização dos recursos dos containers. Na Seção 5 é feita uma comparação entre o HPA e a solução proposta neste trabalho.

III. Revisão Bibliográfica

A. Containers

Container é uma tecnologia para a criação de instâncias de processamento separadas ou isoladas que permitem a virtualização no nível do sistema operacional ao disponibilizar porções protegidas de processamento. Dois *containers* rodando no mesmo sistema, não sabem que estão compartilhando recursos, pois tem sua própria abstração de camada de rede, memória e processos [4].

Os *containers* apresentam uma maior portabilidade que as máquinas virtuais, ao serem configurados de forma genérica para qualquer sistema operacional baseado em Linux. A virtualização via *hypervisors* consome mais recursos do que a virtualização por *containers* dado que os últimos são executados em sistemas operacionais que rodam em espaços isolados entre si. Se um *container* não está executando nenhuma tarefa, ele não está consumindo recursos no servidor [4]. Além disto os *containers* apresentam um grande dinamismo para serem criados e destruídos, dado que apenas tem que iniciar ou destruir processos em seu espaço isolado. Portanto verifica-se que existem vantagens em se utilizar a tecnologia de *containers*. Esta foi a tecnologia escolhida para atender o processamento das cargas de trabalho *http* nesta solução.

B. Serviços Web e Balanceadores de Carga

Os sistemas web podem sofrer com a alta variabilidade da carga demandada, muitas vezes de forma inesperada. Neste caso, a infraestrutura que hospeda este tipo de serviço, deve estar preparada para o atendimento da demanda com alto desempenho, escalabilidade e com alta disponibilidade. Uma abordagem comum para disponibilizar estas características é o uso de arquiteturas distribuídas que podem rotear as requisições para diversos servidores, de forma transparente ao usuário. Esta abordagem além de melhorar a performance e a disponibilidade dos serviços, pode aumentar sua escalabilidade ao se adicionar e remover membros no *cluster*. [5]

Porém vários desafios se impõem para que um *cluster* distribuído de servidores possam funcionar de forma eficiente como se fosse um único servidor. Esses desafios vão desde o roteamento das requisições para os membros do *cluster*, métodos para escolha do membro que receberá a demanda e manutenção do estado da conexão. [7]

No balanceamento na camada de transporte, as requisições são distribuídas entre os membros do *cluster*. O balanceador distribui as conexões de clientes que conhecem o endereço IP do *cluster*, entre os diversos servidores que efetivamente respondem as requisições. Neste caso, como a distribuição é feita com base na camada 4, o balanceador escolhe um servidor, sem considerar o conteúdo ou o tipo da requisição. [1]

C. Caracterização da Chegada das Requisições

A correta caracterização da caraga de trabalho a que um sistema é submetido, é fundamental para a validação de um modelo que propõe dimensionar recursos para atendimento desta demanda.

Vários modelos matemáticos foram produzidos baseados nos modelos de Markov, que consideram que as fontes de carga possuem comportamento Poissioniano [2]. Porém [3], demonstrou que o comportamento dos acessos web possuem comportamento fractal. Portanto, a validação deste trabalho foi realizada utilizando uma carga que como será comprovado na seção V-B, possui comportamento autossimilar.

Os processos fractais podem ser do tipo monofractais ou multifractais. Os processos multifractais têm aplicação para a caracterização de tráfegos em alta frequência, observados em pequenas escalas de tempo, na faixa de milisegundos [17].

Estes processos não serão abordados neste trabalho, pois a caracterização da chegada de requisições http, baseia-se na coleta de dados de acesso obtidos em logs de servidor web, que fornecem valores na escala de segundos. Portanto este trabalho utilizará o processo monofractal para caracterização da chegada de requisições web.

D. Definição de Autossimilaridade

Para uma discussão mais detalhada sobre autossimilaridade em sistemas web, veja [3]. Dada uma série temporal estacionária, com média zero $X = (X_t; t = 1, 2, 3, ...)$, define-se uma série *m*-agregada $X^m = (X_k^m; k = 1, 2, 3, ...)$ com a soma da série original X pelos m blocos não sobrepostos. Diz que X é *H*-autossimilar se para todos *m* positivos, X^m tem a mesma distribuição que X na escala de m^H. Isto é,

$$X_t = m^{-H} \sum_{i=(t-1)m+1}^{tm} X_i, \forall m \in N$$

Se X é H-autossimilar, ele tem a mesma função de autocorrelação $r(k) = E(X_t - \mu)(X_{t+k} - \mu])/\sigma^2$, assim como a série X^m para todo m. Isto significa que a distribuição da série agregada é a mesma (exceto pela diferença na escala) da série original.

Como consequência, os processos autosimilares podem apresentar dependência de longa duração, fato que não é verificado em séries que com características poissonianas. Um processo com com dependência de longa duração tem função de autocorrelação $r(k) \sim k^{-\beta}$ com $k \to \infty$, onde $0 < \beta < 1$. Portanto a função de correlação deste processo segue a lei das potências (power law), que se difere das distribuições comumente usadas na modelagem para cargas de rede, que tem decaimento exponencial. Como estas funções apresentam decaimento lento, a soma dos valores de autocorrelação deste tipo de série, se aproxima do infinito. Isto tem algumas implicações. Primeiro, a variância da média de *n* amostras deste tipo de série não diminui proporcionalmente a $1/n$, mas sim proporcionalmente a $n^{-\beta}$. Segundo, o espectro de potência deste tipo de série, é hiperbólico tendendo para o infinito na frequênica zero, o que reflete a influência infinita da dependência de longa duração nos dados.

Uma dos atributos mais interessantes das séries auto similares, é o parâmetro de Hurst, que demonstra o grau de autossimilaridade de uma série, sendo este $H = 1 - \frac{\beta}{2}$. Portanto, uma série autossimilar com longa dependência, tem $\frac{1}{2} < H < 1$. Com $H \to 1$, tanto o grau de autossimilaridade, quanto a dependência de longa duração aumentam. [3]

E. Estimativa do Parâmetro H

Existem vários métodos para a estimativa do parâmetro de Hurst (H). Neste trabalho serão apresentados os métodos R/S e Kettani-Gubner

F. Análise R/S

Seja uma série infinita $X_k, k = 1, 2, ..., n$, com média $Y(n)$ e variância das amostras dadas por $S^2(n)$, então a amplitude reescalonada é dada por [17]:

$$\frac{R(n)}{S(n)} = \frac{1}{S(n)}[max_{0 \le t \le n}(Y(t) - tY(n)) - min_{0 \le t \le n}(Y(t) - tY(n))]$$

Segundo Hurst várias séries temporais satisfazem a seguinte relação empírica:

$$E[\frac{R(n)}{S(n)}] \approx cn^H, n \to \infty$$

onde c é uma constante finita, independente de n e S(n) é o desvio padrão. Deste modo, a declividade da reta ajustada para *(log(n), log(R(n)/S(n)))* fornece uma estimativa para o parâmetro de Hurst.

G. Método Kettani-Gubner

Para uma discussão mais detalhada sobre este método, veja [13]. Seja um processo extamente autossimilar de segunda ordem. Neste caso, o coeficiente de autocorrelação é descrito pela equação,

$$R(k) = 1/2(|k+1|^{2H}) - 2|k|^{2H} + |k-1|^{2H}$$

Para k=1 tem-se $R(1) = 2^{2H-1}$, isolando H, têm-se $\hat{H} = \frac{1}{2}[1 + \log(1 + \hat{R_n}(1))]$. Sob a suposição de que o processo seja ergódigo, é possível calcular o parâmentro H, com [2],

$$\hat{H_n} = \frac{1}{2}[1 + log(1 + \hat{R_n}(1))]$$

H. Controladores PID

Os controladores PID (Proporcional-Integral-Derivativo) são algoritmos de controle muito utilizados na industria. Os controladores PID possuem três coeficientes: proporcional, integral e derivativo. Esses coeficientes são variados de forma a se obter a resposta de controle ideal desejada para um processo.

Um controlador PID trabalha dentro de um sistema em malha fechada, onde é possível a leitura do estado atual de determinada variável que se deseja controlar, e de acordo com seu valor, uma ação é executada de modo que a variável chegue, e tente permanecer no nível desejado (ainda considerando distúrbios externos), nas próximas iterações de tempo [11].

Sendo assim, o PID deve ler o estado atual da variável e calcular a resposta da saída do atuador, por meio do cálculo dos componentes proporcional, integral e derivativo e então somar os três para calcular a saída. O componente proporcional depende da diferença entre o valor desejado (*setpoint*) e o valor atual da variável. Esta diferença é referida como erro. O componente integral soma o termo de erro ao longo do tempo. A resposta derivativa é proporcional a taxa de variação da variável do processo.

Para que o controlador PID produza os ajustes necessários ao sistema, os parâmetros de ganho K_p, K_i e K_d devem ser ajustados. Existem diversos métodos de ajuste destes parâmetros, como por exemplo O método manual (*guess and check*) e o método Ziegler-Nichols [11].

No método manual, os ganhos de cada um dos componentes é ajustado usando tentativa e erro. Para isso, ao ajustar os parâmetros, devem ser conhecidos os efeitos que cada parâmetro provoca na saída do controlador. Neste método, os termos K_i e K_d são ajustados para zero, e o termo K_p é aumentado até que a saída do ciclo comece a oscilar. A partir daí aumenta-se lentamente o termo K_i para reduzir o erro

estacionário. Neste ponto inicia-se o incremento do termo K_d, de modo a diminuir as oscilações na saída do ciclo [11]. A discussão sobre outros métodos de ajuste dos parâmetros foge ao escopo deste trabalho, já que foi utilizado o método de ajuste manual.

IV. Solução Proposta

O objetivo deste trabalho, é a provisão de um ambiente elástico de nuvem privada, auto escalável e com balanceamento de carga, para hospedagem de sistemas Web.

A solução propõe um ambiente de *cluster* baseado em *containers*, e um método de auto elasticidade que reage a aumentos no tempo de resposta do sistema, de modo a mantê-lo dentro de um limite. Para isso é utilizado um sistema de malha fechada com um controlador PID, que reage a variações no tempo de resposta do sistema aumentando ou diminuindo o número de *containers* no *cluster* de balanceamento de carga capaz de responder as requisições.

A. Ferramentas Utilizadas para Implementação

B. Docker

O *Docker* começou como projeto da empresa de PaaS (*Platform as a Service*) dotCloud em 2013 [15], propondo ser um integrador e facilitador para adoção da tecnologia de *containers* em produção e em larga escala. O Docker utiliza o LXC (linux *containers*) para isolar os containers do servidor, criando isolamento de processos, rede e privilégios. A limitação e contabilização de recursos (CPU, memória, espaço em disco e E/S) é feita por meio dos *cgroups*. Além disso a utilização do sistema de arquivos é feita de forma eficiente pois se baseia em *copy-on-write*, que permite que as alterações em um *container* sejam simplesmente uma atualização diferencial da imagem anterior.

Uma das grandes vantagens do *Docker* é a habilidade de encontrar, baixar e iniciar imagens de *containers* que foram criados por outros programadores de forma muito rápida e prática. O *Docker* viabiliza o uso da tecnologia de *containers* de forma prática, razão pela qual é utilizado nesta solução.

C. Kubernetes

Kubernetes é um sistema desenvolvido pelo Google [9], e disponibilizado para a comunidade, que visa gerenciar o ciclo de vida de *containers* nos nós de um *cluster*.

Desta forma, o *Kubernetes* é um orquestrador de *containers*, sendo capaz de agendar o lançamento de *containers* entre os nós do *cluster*, assim como fazer o controle de admissão dos *containers*, balanceamento de recursos e prover escalabilidade ao ambiente. O *Kubernetes* também provê funcionalidades como descoberta de serviços entre os *containers*, publicação do serviço para acessos a partir de entidades fora do *cluster* e balanceamento de carga entre os *containers* [9]. Estas funcionalidades foram determinantes para a decisão de usar o Kubernetes na solução.

A infraestrutura de um *Kubernetes* é composta por nós do tipo *Master* que controlam os nós do tipo *Workers*, e estes rodam os *containers* . Todas as configurações do *cluster* ficam armazenadas de em um repositório de configurações distribuído, o *Etcd*. Os *PODs* são a unidade básica com a qual o *Kubernetes* trabalha. Os *containers* são agrupados em *PODs* e estes geralmente representam uma aplicação. Estes são criados por meio dos Replication Controllers que são utilizados para definir PODs que podem ser escalados horizontalmente. Os Replication Controllers também são responsáveis por manter o número desejado de PODs ativos no *cluster*.

D. Apache Spark, Flume, HAproxy e Redis

O *Apache Spark* é uma ferramenta de processamento distribuído, ideal para processamento de grandes bases de dados. Foi desenvolvido pela AMPLab (UC Berkeley), e realiza processamento de dados em memória. Sua estrutura básica de abstração são os *RDDs (Resilient Distributed DataSets)*, que são coleções de elementos que podem sofrer operações em paralelo, sendo possível a geração de novos RDDs a partir de transformações como *map, reduce, filter e join* em *RDDs*. [19] Esta ferramenta foi escolhida para integrar a solução pois com ela é possível realizar processamento de grandes bases em formato de texto, de forma escalável.

O *Apache Spark* oferece uma API chamada *Spark Streaming*, que permite o processamento em tempo real de dados, por meio da criação de estruturas chamadas *DStream* (*discretized stream*), que são representados como sequências de *RDDs*. A criação dos *DStreams* é feita por meio da classe *StreamingContext*, onde se configura a duração de cada janela de *DStreams* [20]. Neste trabalho, a duração de cada janela foi configurada como 5 segundos, o que é suficiente para a geração da série temporal em tempo real utilizada pela solução.

Neste contexto o *Spark Streaming* é utilizado para processar os logs do balanceador de carga, coletando o tempo de resposta de cada uma das requisições que o sistema atende, em tempo real. Esta informação do tempo de resposta é armazenada em formato de série temporal no servidor *Redis*, para uso no algoritmo de auto escalabilidade proposto neste artigo.

O recebimento das entradas de log em texto do balanceador de carga pelo *Spark*, ocorre por meio da ferramenta *Flume*. O *Flume* é um serviço que agrega, coleta e move grandes volumes de fluxo de dados. Para seu funcionamento é criada uma fonte que recebe os dados de interesse. Essa fonte (*Source*) é conectada a um canal (*Channel*), por onde os dados trafegarão em direção a um *Sink* [6].

Portanto, a função desta ferramenta na solução, é realizar o envio, em tempo real, dos logs de acesso do balanceador carga, por meio da criação de uma fonte do tipo *Syslog*, que recebe os logs do balanceador de carga, e um canal de memória que carrega os dados da fonte em memória. A partir daí, o *Spark* consome este fluxo de dados por meio de um *Sink* do tipo *Avro*, que trafega pela rede.

O balanceador de carga utilizado na solução, é o *HAproxy*, que pode atuar como balanceador de camada 4 ou 7, terminador SSL, proxy reverso entre outros [21]. Atualmente, este é o balanceador utilizado por *sites* como *Reddit, Stack Overflow/Server Fault, Instagram* entre outros, além de ter sido escolhido como balanceador da nuvem da Red Hat (*OpenShift*).

Esta adoção pela comunidade foi uma das razões de se ter escolhido esta ferramenta para compor a solução. Além disso, o *HAproxy* disponibiliza um *log* com informações viabilizam a implementação da proposta deste trabalho.

O log do *HAproxy* apresenta o seguinte formato:

May 18 06:24:25 10.125.7.229 haproxy[1078]: 10.125.8.252:43839 [18/May/2016:06:24:24.988] cherrypy cherrypy/10.125.7.227 0/0/2/26/28 200 169 - - —- 1/1/1/0/0 0/0 "GET /generate HTTP/1.0"

Nesta linha pode-se observar informações como o tempo de espera na fila do servidor de aplicação, método e código da resposta *http* entre outras. A parte com *0/0/2/26/28* contém as informações : *Tq '/' Tw '/' Tc '/' Tr '/' Tt*, onde O *Tr* é o tempo em milisegundos que o balanceador demora para receber uma resposta completa de uma requisição *http* ao servidor [21]. Portanto este valor representa o tempo total de processamento da requisição pelo *container*.

No contexto deste trabalho o *SparkStreaming*, processa as entradas do *log*, separa o campo *Tr* e o armazena no banco *Redis*, no formato de uma série temporal. O *SparkStreaming* também é responsável por converter o formato da data de cada linha, para o número de segundos desde 0 hora de cada dia, para suportar a criação da série temporal.

O *Redis* é utilizado para armazenar a série temporal, pois consegue prover baixa latência tanto para escrita como para leitura, por manter os dados como estruturas em memória [18].

A integração da solução com o *Redis*, ocorre por meio da biblioteca *Kairos*, que cria uma estrutura para armazenamento de séries temporais em bancos como *Redis*, *Mongo*, *SQL* ou *Cassandra* [12].

Esta biblioteca provê facilidades, como configuração do número de entradas a serem mantidas no banco e tamanho da unidade mínima de tempo de interesse da série. No caso deste trabalho são mantidos armazenados na série a cada momento os dados dos últimos 600 segundos, o que é suficiente para operação do algoritmo.

O *Kairos* também permite o cálculo de parâmetros estatísticos da série, utilizando tamanho de janelas configuráveis de tempo. Isso permite que seja calculada a média de tempo de resposta de uma aplicação nos últimos dois minutos por exemplo. A unidade mínima de tempo configurada nesta solução é 1 segundo. Isso permite boa flexibilidade para configuração dos tamanhos das janelas de tempo para os cálculos estatísticos, além de permitir a geração de gráficos com boa resolução de tempo para monitoramento e avaliação da solução.

E. Arquitetura da Solução

A arquitetura proposta é apresentada na Figura 1, em que é utilizado um controlador PID para manter o tempo médio de resposta de determinada aplicação próxima de um determinado limiar. A arquitetura proposta, chamada de PAS (PID *based Autoscaler*) opera com base na seguinte sequência:

1) É estabelecido um limiar (*setpoint*) de tempo médio de resposta desejado para as requisições. O monitor recebe o tempo de resposta das requisições que chegam no balanceador;
2) O monitor envia o tempo médio de resposta (ex: média dos últimos 200 segundos) para que o dimensionador PID calcule o número de *containers* necessários para atingir ou permanecer no *setpoint*. A média dos últimos 200 segundos foi definida, pois verificou-se experimentalmente que é uma quantidade suficiente para que valores muito diferentes da média não exerçam influência negativa no dimensionamento, assim como permite que o sistema seja capaz de reagir rapidamente a mudanças no padrão da média do tempo de resposta.
3) O Dimensionador PID executa o algoritmo 1 e informa o número desejado de *containers* ao *Kubernetes*. O número atual de *containers* é pesquisado usando a ferramenta do kubernetes *kubectl*, e o tempo médio de resposta do *cluster* é determinado usando a série temporal presente no *Redis*.
4) O *Kubernetes* cria ou remove novos *containers*, além de garantir que o ambiente permanecerá com o número desejado de *containers* até a próxima rodada do algoritmo (ex: depois de 10 segundos). Este valor de 10 segundos foi definido pois verificou-se que é suficiente para que o *Kubernetes* inicie novos *containers* e estes passem a responder as requisições no *cluster* de balanceamento.

Algoritmo 1: PAS

Entrada: Tempo médio de resposta do cluster, Número atual de *containers*
Saída: Número desejado de Containers

início
  Leia o limiar de tempo médio de resposta desejado para as requisições: $t\_ms\_desejado$
  Leia o número atual de *containers*: $n\_containers\_atual$
  Leia tempo de resposta médio do cluster em ms: t_ms_atual Calcule o erro: $e(t)$ = $t\_ms\_desejado$ - t_ms_atual Calcule a saída o controlador pid:

$$u(t) = K_p e(t) + K_i \int_0^t e(t)\delta t + K_d \frac{d}{dt} e(t) \quad (1)$$

  $n\_desejado\_containers$=$n\_containers\_atual$ + $u(t)$
fim
retorna *n_desejado_containers*

F. Operação da solução

Para viabilizar a operação do algoritmo que trabalha com dados dinâmicos, recebidos em tempo real, utiliza-se o fluxo de processamento mostrado na Figura 2. O *Haproxy* atua como balanceador de carga da solução. Seus *logs* são enviados ao *Flume* que envia os dados em um canal de memória e os envia ao *Spark Streaming*, desta forma disponibilizando as informações necessárias para operação do algoritmo de auto escalabilidade.

O *Haproxy* atua balanceando as requisições no modo *round robin* para os servidores *Docker/Kubernetes* que hospedam os

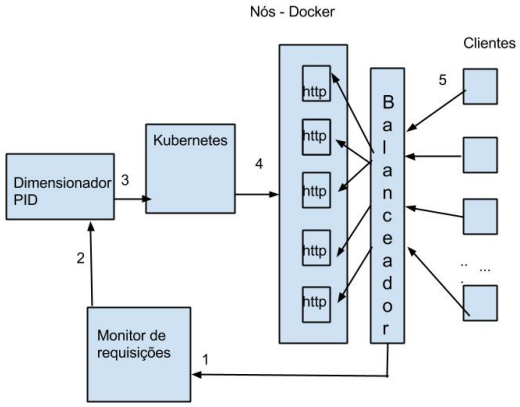

Figure 1. Arquitetura da Solução

containers. Quando o nó *Docker/Kubernetes* recebe a requisição, o serviço *Kubernetes proxy* é responsável por balancear a carga entre os *containers* de cada um dos servidores.

Desta forma, acontecem dois níveis de balanceamento, um entre os nós *Docker/Kubernetes* em que o responsável pelo balanceamento é o *Haproxy*, e outro internamente dentro do nó *Docker/Kubernetes*, onde o serviço *Kubernetes proxy* é o responsável pelo balanceamento.

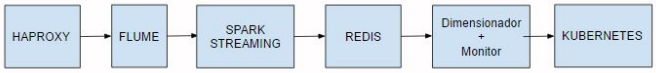

Figure 2. Fluxo de processamento entre o conjunto de ferramentas utilizadas

V. Resultados Experimentais

A. Ambiente

Para avaliação da solução foi configurado um *cluster Kubernetes* v1.1.2 no sistema operacional *CoreOS* (899.6.0 (2016-02-02)), virtualizado em *VMWare ESXi* 5.5.0. Este ambiente foi configurado para validação da solução. Um ambiente de produção poderia se beneficiar mais se o sistema *CoreOS* fosse instalado diretamente em máquinas físicas, pois seria eliminada a camada de virtualização.

O *cluster* foi constituído com os seguintes componentes: 1 nó *master* (4 vCPUs, 6 GB de RAM), 1 nó *etcd* (4 vCPUs, 6 GB de RAM) e 4 nós *workers* (4 vCPUs, 6 GB de RAM). As máquinas virtuais (*VMWare ESXi* 5.5.0) foram instaladas no sistema operacional Ubuntu 14.04.3 LTS, com as seguintes configurações e ferramentas: 1 nó *haproxy* 1.5.4 (4 vCPUs, 4G GB de RAM), 1 nó *Spark* 1.5.2 + *Redis* 2.8.4 (2 vCPU, 10 GB de RAM) e 1 nó *Flume 1.7.0* + dimensionador (2 vCPU, 4 GB de RAM).

Para validação do ambiente foi gerada uma imagem de *container* que roda o servidor web *cherrypy* 5.1.0. Foi configurado um *link* neste servidor. O *link* gera um *array* de tamanho aleatório entre 1.000 e 10.000 elementos a cada requisição.

A publicação do serviço no *Kubernetes* foi feita por meio da criação de um *Replicaction Controller* e a configuração de um serviço do tipo *NodePort*. O balanceador *Haproxy* foi configurado para balancear as requisições entre os nós *Docker/Kubernetes* usando os endereços IP dos nós e as portas publicadas pelo serviço do tipo *NodePort*. Cada *container* teve seus recursos de processamento e memória limitados a 18 MB de memória RAM e 24 ms de CPU.

O algoritmo PID utilizou os seguintes parâmetros: Kp=0.016, Ki=0.000012 e Kd=0.096, que foram definidos após vários testes com a carga de trabalho, e aplicação do método de ajuste manual, ou *guess and check*, conforme descrito na seção III-H

B. Carga de Trabalho

A geração de carga de trabalho foi feita usando a ferramenta *ab (apache bench)*. Para geração de uma carga com um perfil de requisições por segundo realista e com característica autossimilar, foi coletado um conjunto de acessos do portal da transparência (www.transparencia.gov.br), entre os dias 25/05/2015 a 25/06/2015. A série temporal capturada na escala de 1 segundo, representa o número de acessos naquele intervalo de tempo. A série foi caracterizada com o método Kettani-Gubner [13]. A autossimilaridade e longa dependência da série foi confirmada com o parâmetro de Hurst H=0,87. nas escalas de 1 segundo, 10 segundos e 100 segundos.

Uma amostra da série obtida utilizada para caracterizar a carga de trabalho, relativa a 2000 segundos pode ser vista na Figura 3. Esta carga foi tem característica autossimilar. Isto pode ser comprovado ao se agregar a série completa desta carga, em diferentes escalas de tempo, e estimar o parâmetro H. As figuras: 4, 5 e 6, mostram o perfil desta carga, durante 24 horas, em diversas escalas de tempo, assim como o parâmetro H estimado para cada escala.

Esta série é utilizada para gerar a cada segundo requisições simultâneas direcionadas ao endereço IP do balanceador de carga que distribui as requisições nos nós do *cluster Kubernetes*. A intensidade da carga foi ajustada em 3 níveis, multiplicando a série temporal por 1, 1.5 e 2 e preservando o mesmo índice de autossimilaridade. Estas cargas são referidas respectivamente nos experimentos como carga_1, carga_1.5 e carga_2.

C. Cenários de avaliação

1) Cenário 1: No primeiro cenário o limiar de tempo de resposta no balanceador (*setpoint*) foi estabelecido em 50 ms e foram aplicadas as cargas carga_1 e carga_1.5. A Figura 7 mostra o tempo de resposta do sistema ao ser submetido à carga_1. O gráfico mostra o ajuste provocado pela alocação de *containers* do algoritmo PAS e a estabilidade alcançada próximo do *setpoint* de 50 ms.

A Figura 8 mostra a alocação de *containers*. Observa-se que o número de *containers* no início do experimento é igual a 2. O sistema alocou o número necessário para que o tempo médio de resposta mostrado na Figura 7 atingisse o *setpoint*. No final da execução estavam alocados 26 *containers*.

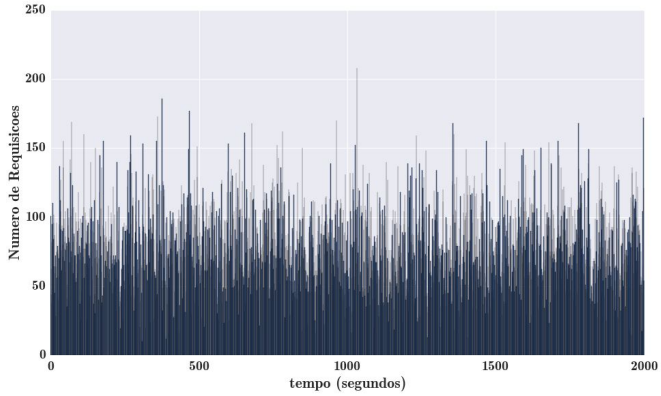

Figure 3. Amostra da carga utilizada nos testes, H=0,87

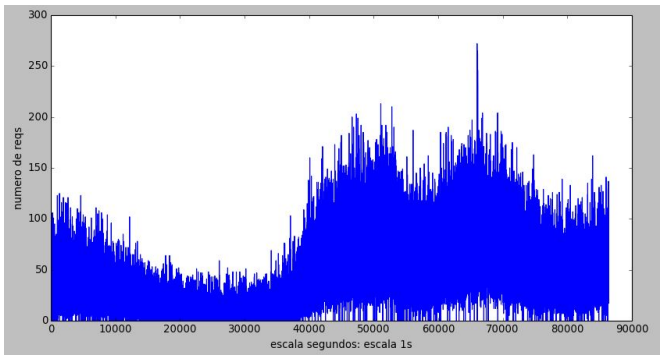

Figure 4. Carga, escala 1s, H=0,8738

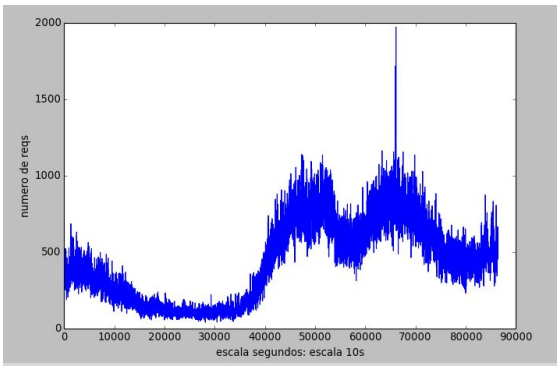

Figure 5. Carga, escala 10s, H=0,8795

A Figura 9 mostra o tempo médio de resposta com o sistema sendo submetido a carga_1.5 e o *setpoint* mantido em 50 ms. A Figura 10 mostra a alocação de *containers* durante este teste. Observa-se que a alocação de *containers* foi sendo ajustada de modo a manter o tempo de resposta médio do sistema próximo ao *setpoint*. No final da execução estavam alocados 52 *containers*. Neste caso, o sistema submetido a uma carga maior, alocou mais containers para manter o tempo de resposta dentro do limiar definido.

2) Cenário 2: O Cenário 2, compara o comportamento do algoritmo de escalabilidade proposto neste trabalho com o utilizado nativamente pelo *Kubernetes*, denominado de HPA [10].

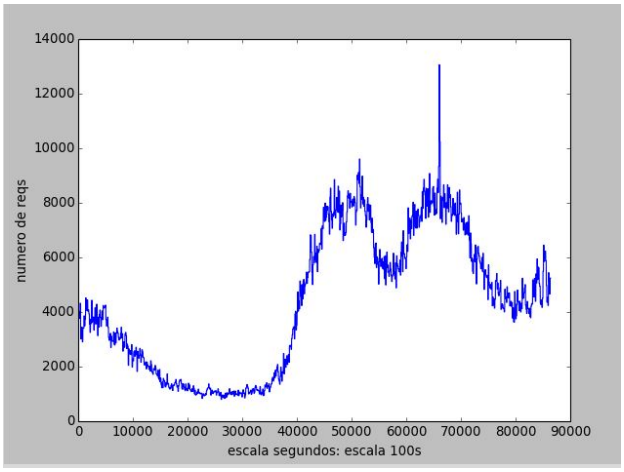

Figure 6. Carga, escala 100s, H=0,8532

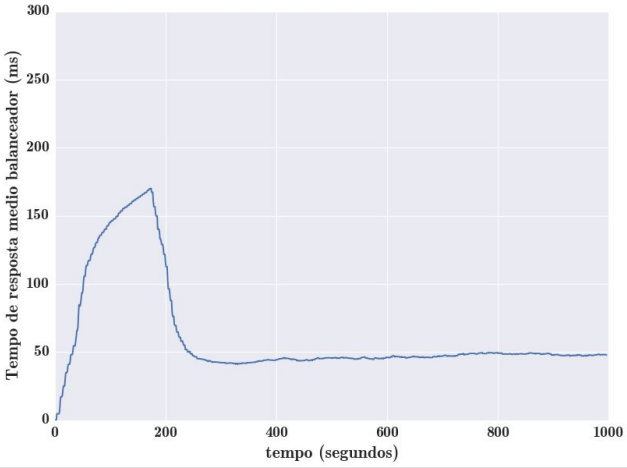

Figure 7. Tempo de Resposta (ms) x Tempo (s), carga carga_1, *setpoint* 50 ms

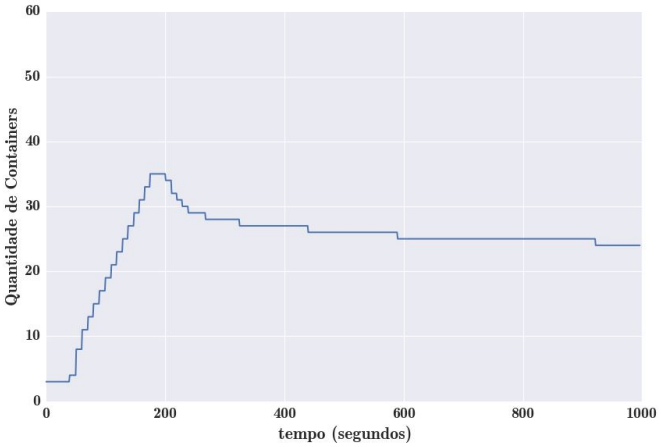

Figure 8. Número de containers x Tempo (s), carga_1, *setpoint* 50 ms

A comparação é feita da seguinte forma: o sistema configurado com o HPA_80 (configurado para escalar quando o

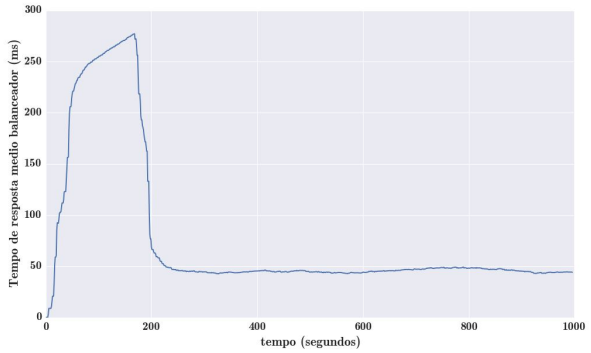

Figure 9. Tempo de Resposta (ms) x Tempo (s), carga carga_1.5, *setpoint* 50 ms

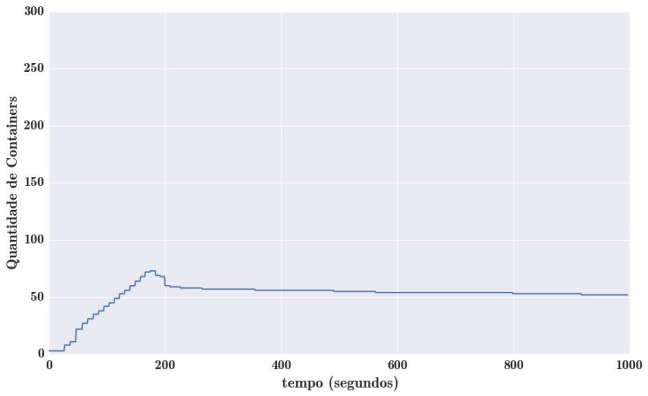

Figure 10. Número de *Containers* x Tempo (ms), carga carga_1.5, *setpoint* 50 ms

consumo médio dos *containers* estiver acima de 80 %) será exposto às cargas carga_1, carga_1.5 e carga_2 durante 1000 segundos. No fim da avaliação o tempo médio de espera das requisições na camada de aplicação (ponto de vista dos clientes) é observado.

A partir destes dados, foi definido um *setpoint* para ser utilizado com o algoritmo proposto (PAS) que permita comparar com o tempo de resposta próximo ao de referência do HPA. Os resultados serão comparados, verificando a quantidade média de *containers* utilizados durante os experimentos e os tempos de resposta obtidos na camada de aplicação dos clientes.

Como pode ser observado na Figura 11 os tempos médios de resposta na camada de aplicação com o algoritmo HPA para cada carga, foram: 66.18 ms para a carga_1, 110.12 ms para a carga_1.5 e 144.01 ms para a carga_2.

Para efeitos comparativos, os *setpoints* do PAS, foram configurados para entregar um tempo de resposta próximo aos obtidos com o HPA. Para isto foram configurados *setpoints* um pouco abaixo dos observados no HPA, dado que o *setpoint* é controlado no balanceador, portanto o tempo observado na camada de aplicação dos clientes deve ser maior. Os valores definidos para os *setpoints* são os seguintes: 50 ms (PAS_50) para a carga_1, 80 ms (PAS_80) para a carga_1.5 e 100 ms (PAS_100) para a carga_2.

As Figuras 11 e 12 mostram que para um tempo de resposta aproximado ao do HPA, o sistema PAS, alocou menos *containers* do que o HPA. O comparativo de alocação de *containers* mostra que: PAS_50 alocou 44.02 % do que foi alocado pelo HPA_80, PAS_80 alocou 36.07 % do que foi alocado pelo HPA_80, para a carga_1.5 e PAS_100 alocou 12.72 % do que foi alocado pelo HPA_80, para a carga_2.

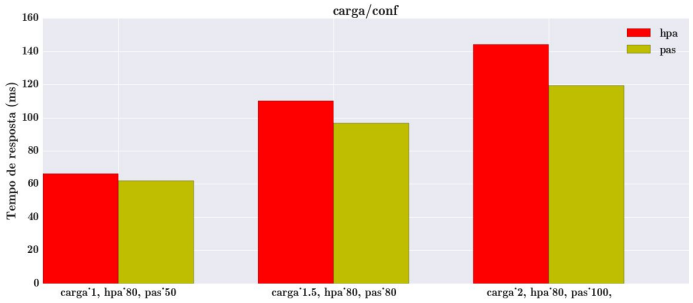

Figure 11. Comparativo entre HPA e PAS (tempo médio de resposta do sistema)

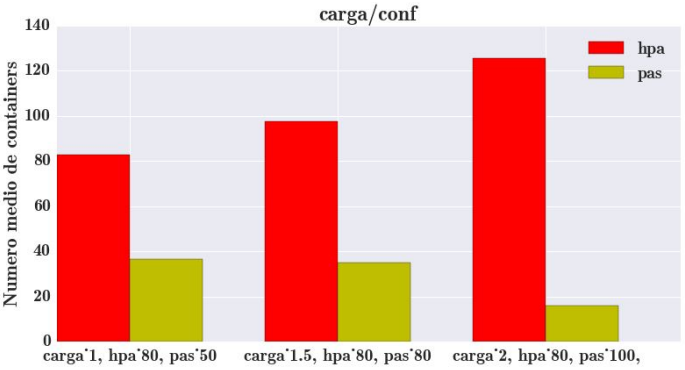

Figure 12. Comparativo entre HPA e PAS (número médio de containers)

D. Análise dos Resultados

Os resultados experimentais mostram que nos cenários avaliados, o algoritmo PAS otimiza a alocação do número de *containers* para manter os valores de *setpoints* dentro de um limiar estabelecido. O HPA aloca um número maior de *containers* para alcançar limiares equivalentes de tempo de resposta para as requisições na camada de aplicação. Este resultado evidencia que a alocação de um número maior de *containers* pode aumentar a complexidade do tempo de balanceamento de carga e não necessariamente produzir melhores tempos de resposta.

Os resultados obtidos mostram que o algoritmo PAS, proposto neste trabalho apresenta potencial para promover a otimização de alocação do número de *containers* em um ambiente de computação em nuvem.

VI. Conclusão e Trabalhos Futuros

Este trabalho apresentou o algoritmo PAS (PID *based Autoscaler*) e os resultados mostram um potencial para prover auto elasticidade a um ambiente de computação em nuvem baseado em *containers*. A comparação com a ferramenta de auto escalabilidade nativa do *Kubernetes*, mostra uma maior

eficiência da solução proposta para alocação da quantidade de *containers* com o objetivo de cumprir limiares pré estabelecidos de tempo de resposta na execução de requisições para aplicações Web. A proposta foi validada com um perfil de carga realista e autossimilar.

O uso de ferramentas como *spark* e *flume* possibilita que a arquitetura seja escalável, podendo vir a ser utilizada por *sites* com servidores com grande número de acessos, em que a série temporal resultante do número de requisições em diversas escalas de tempo tende a ter um alto grau de autossimilaridade.

A proposta possibilita um uso eficiente de recursos, o que pode permitir um menor custo de operação da infraestrutura instalada e dos serviços oferecidos em nuvem.

Os trabalhos futuros pretendem validar os resultados experimentais em um maior número de cenários e confronta-los com medições em ambientes de produção que permitam expandir a aplicação desta pesquisa para ambientes reais de computação em nuvem.

REFERENCES

[1] Mitchell Anicas. Mitchel Anicas an introduction to haproxy and load balancing concepts. https://www.digitalocean.com/community/tutorials/an-introduction-to-haproxy-and-load-balancing-concepts, 2014. [Accessado em 03/04/2016].

[2] Priscila América Solis Mendez Barreto. Uma metodologia de engenharia de tráfego baseada na abordagem auto-similar para a caracterização de parâmetros e a otimização de redes multimídia. 2010.

[3] Mark E Crovella and Azer Bestavros. Self-similarity in world wide web traffic: evidence and possible causes. *Networking, IEEE/ACM Transactions on*, 5(6):835–846, 1997.

[4] Docker. Docker the definitive guide to docker containers. https://www.Docker.com/sites/default/files/WP-%20Definitive%20Guide%20To%20Containers.pdf, 2016. [Accessado em 03/04/2016].

[5] F5. F5 load balancing 101 - nuts and bolts. https://f5.com/resources/white-papers/load-balancing-101-nuts-and-bolts, 2012. [Accessado em 03/04/2016].

[6] Flume. Flume flume user guide. https://flume.apache.org/FlumeUserGuide.html, 2016. [Accessado em 03/04/2016].

[7] Katja Gilly, Carlos Juiz, and Ramon Puigjaner. An up-to-date survey in web load balancing. *World Wide Web*, 14(2):105–131, 2011.

[8] Zhenhuan Gong, Xiaohui Gu, and John Wilkes. Press: Predictive elastic resource scaling for cloud systems. In *Network and Service Management (CNSM), 2010 International Conference on*, pages 9–16. IEEE, 2010.

[9] Google. Google container cluster manager from google. https://github.com/kubernetes/kubernetes, 2016. [Accessado em 03/04/2016].

[10] Google. Google horizontal pod autoscaler. https://github.com/kubernetes/kubernetes/blob/release-1.2/docs/design/horizontal-pod-autoscaler.md, 2016. [Accessado em 03/04/2016].

[11] National Instruments. National Instruments explicando a teoria pid. http://www.ni.com/white-paper/3782/pt/, 2015. [Accessado em 20/08/2015].

[12] Kairos. Kairos time series data storage in redis, mongo, sql and cassandra. https://pypi.python.org/pypi/kairos, 2015. [Accessado em 03/04/2016].

[13] Houssain Kettani, John Gubner, et al. A novel approach to the estimation of the hurst parameter in self-similar traffic. In *Local Computer Networks, 2002. Proceedings. LCN 2002. 27th Annual IEEE Conference on*, pages 160–165. IEEE, 2002.

[14] Tania Lorido-Botrán, José Miguel-Alonso, and Jose Antonio Lozano. Auto-scaling techniques for elastic applications in cloud environments. *Department of Computer Architecture and Technology, University of Basque Country, Tech. Rep. EHU-KAT-IK-09*, 12:2012, 2012.

[15] Nick Martin. Nick Martin a brief history of docker containers' overnight success. http://searchservervirtualization.techtarget.com/feature/A-brief-history-of-Docker-Containers-overnight-success, 2015. [Accessado em 03/04/2016].

[16] Rishabh Poddar, Anilkumar Vishnoi, and Vijay Mann. Haven: Holistic load balancing and auto scaling in the cloud. 2015.

[17] Lócio Fernando Postai. Caracterização de tráfego em uma rede san-ip utilizando protocolo iscsi. 2011.

[18] Redis. Redis redis documentation. http://redis.io/documentation, 2015. [Accessado em 03/04/2016].

[19] Spark. Spark spark programming guide. https://spark.apache.org/docs/1.5.2/programming-guide.html, 2015. [Accessado em 03/04/2016].

[20] Spark. Spark spark streaming programming guide. https://spark.apache.org/docs/1.5.2/streaming-programming-guide.html, 2015. [Accessado em 03/04/2016].

[21] Willy Tarreau. HAProxy haproxy configuration manual. http://www.haproxy.org/download/1.5/doc/configuration.txt, 2015. [Accessado em 03/04/2016].

Anexo III

Artigo publicado no Congresso IEEE NCA 2016 - Cambridge/MA/EUA [39]

An Algorithm Based on Response Time and Traffic Demands to Scale Containers on a Cloud Computing System

Marcelo Cerqueira de Abranches
Departamento de Ciência da Computação
Universidade de Brasília
Brasília, Distrito Federal (61) 3107-6737/3658
Controladoria-Geral da União
Federal Government of Brazil
Brasília, Distrito Federal (61) 2020-6964
Email: marcelo.abranches@cgu.gov.br

Priscila Solis
Departamento de Ciência da Computação
Universidade de Brasília
Brasília, Distrito Federal (5561) 3107-6737/3658
Email: pris@cic.unb.br

***Abstract*—This paper proposes a cloud computing architecture based in containers and on an algorithm that intends to achieve an efficient allocation of processing resources to comply with response time requirements in a Web system. The algorithm is based on the characterization of web requests and on a PID (Proportional - Integral- Derivative) controller. The proposal was evaluated with a real time series obtained from an operational massive web system in a controlled infrastructure. The results show that the proposal achieves the expected response times allocating a lower number of containers than other related proposals.**

I. Introduction

Some of the most challenging and interesting topics on cloud computing environments are auto elasticity algorithms [13] and load balancing procedures. Several recent works address elasticity in cloud computing environments [13], [11], [5] [7]. Elasticity is a key feature in the cloud computing area and is the main characteristic that distinguishes this computing paradigm from the other ones such as grid computing or cluster computing.

The scalability describes the systems ability to reach a certain scale. Is the ability of the system to be enlarged as necessary, mainly to accommodate future growth adding more resources. Elasticity is a dynamic property that allows the system to scale on-demand in an operational system. Elasticity is the ability for clients to quickly request, receive, and release, many resources as needed. The elasticity implies in fluctuations, i.e., the number of resources used by a client may change over time.

The policy to implement elasticity can be manual or automatic. A manual policy means that the user is responsible for monitoring his virtual environment and applications and for performing all elasticity actions. Normally, the cloud provider provides interfaces with the system with this purpose. In automatic policy, the control is done by the cloud system, in accordance with user requirements, normally specified in the Service Level Agreement (SLA). Then, auto elasticy means to automatically adapt the environment, and even optimize resources according to the user demands.

978-1-5090-3216-7/16/$31.00

This work proposes an algorithm and an architecture to promote auto elasticity on a cloud computing environment based on the efficient allocation of resources. Our work is focused on processing power elasticity.

This paper makes the following contributions: first, we propose a cloud computing architecture, integrating several technologies to promote auto elasticity. Secondly, we analyse and characterize the web requests of a massive web system and propose an algorithm that using this typical workload allows to allocate processing resources to comply with QoS requirements, in our case, the response time. And finally, this paper evaluates the proposal in an experimental environment using several scenarios.

This work is organized as follows: section 2 presents the related work. Section 3 contains the literature review and the theoretical concepts used in our proposal. Section 4 describes the tools and technologies used in the proposed architecture. Also this section details the proposed auto scaling algorithm. Section 5 presents the experimental results. Finally, section 6 presents the conclusions and future work of this research.

II. Related Work

The work [11], compares different methods to obtain auto elasticity on a cloud computing environment. These methods can be classified into 2 categories: reactives and predictives. Those techniques are based on machine learning, queueing theory, control theory, temporal series analysis, among others.

The work [5] proposes an algorithm called PRESS to predict CPU loads by extracting consumption patterns and adjust resource allocation. Their approach uses two methods to perform online predictions: the first is based on the use of signal processing (Fast Fourier Transforms -FFT) to extract dominant frequencies. This frequencies are used to generate a time series and different time windows are compared. The Pearson correlation index is generated for various windows. If

it is obtained a Pearson correlation index greater than 0.85, the average value of the resources in each position of the time series is used to generate a forecast to the next window and the virtual machine's resources are adjusted. If a pattern is not identified, they propose an approach that uses a Markov chain with finite number of states to perform the forecast.

Another work that proposes auto elasticity is Haven [13]. This proposal is based on monitoring CPU and memory loads for each virtual machine in a load balancing pool. From CPU and memory thresholds previously established, Haven loads new virtual machines and insert them in the load balancing pool. In addition, the load balancer which is implemented with SDN (Software Defined Network), directs each request to the least loaded member of the pool.

The work [7] proposes and provides a tool called HPA (Horizontal Pod Autoscaler). This tool works by scaling the environment (number of containers) from CPU average thresholds of containers that serve a given application.

Our proposal is different from PRESS, as it does not perform load prediction, but rather, performs resource allocation or deallocation, based on the response time observed from an application behind a load balancer, prior to a workload characterization. Our proposal is reactive since it uses the variation of the average response time of an application to decide using control theory the allocation or deallocation of resources. Also, our proposal is different from PRESS that performs vertical scaling. Our proposal performs horizontal scalability, that is, new instances are allocated behind a load balancer. As our proposal, Haven also performs horizontal scalability but with virtual machines and our proposal uses containers [2].

Another difference of our proposal regarding to HPA is that while HPA performs CPU consumption measurements inside containers, our proposal performs measurements outside of containers, i. e., in the load balancer. This approach has the advantage to watch system performance, regardless of the level of resource utilization of containers. In Section 5 a comparison is made between the HPA and the solution proposed in this paper .

III. Theoretical Concepts and Literature Review

A. Containers

Container is a technology for creation of isolated processing instances and enables virtualization at the operating system level to provide protected processing portions and when running on the same system, they are not aware that are sharing resources as each one has its own network abstraction layer, memory and processes[2].

Containers have a great portability, because they can run on any operating system based on Linux. Virtualization depends on a hypervisor to achieve similar portability. Virtualization via hypervisors consumes more resources than containers. If a container is not performing any task, it is not consuming resources on the server [2]. Besides that, containers are very dynamic to be created and destroyed, as they just have to start or destroy processes in its isolated space.

B. Web Servers and Load Balancers

In our proposal, the infrastructure hosting the web system must be prepared to meet the demand with high performance, scalability and high availability. However there are many challenges to be addressed so that a cluster of distributed servers can function efficiently as if it were a single server. These challenges range from the routing of requests to the members of the cluster, methods for choosing the member to receive the workload and methods to maintain the connection status.[4].

In load balancing at the transport layer, the requests are distributed among the members of the cluster based on informations like IP addresses and ports. The load balancer distributes client connections, which must know the IP address cluster, among the various servers that effectively respond the requests. In this case, as the load balancing process is based on layer 4, the server is selected regardless of the content or the type of request. [1]. When load balancers are in layer 5, the distribution of workloads is based on the contents of the requests.

C. PID Controllers

PID controllers (Proportional - Integral- Derivative) are control algorithms that are widely used in industry. Examples of its applications are temperature control environments, and drone control. PID controllers have three coefficients : proportional , integral and derivative. These coefficients are varied in order to obtain the desired optimum control response for a given process.

A PID controller works in a closed loop system where it is possible to read the current state of a particular variable that is being controlled, and according to its value, an action is performed so that the variable of interest converges and remains on a desired level (even considering external disturbances), for the next iterations of time. [8]

Thus, the PID controller should read the current state of the variable and calculate the output response, by calculating the proportional, integral and derivative components and then adding the three for calculating the control output. The proportional component depends on the difference between the desired value (*setpoint*) and the current value of the variable. This difference is referred to as an error. The integral component adds the error term over time. The derivative response is proportional to the rate of change of the process variable.

In order to produce the necessary adjustments to the system, the PID controllers use gain parameters K_p K_i and K_d that should be adjusted. There are several methods for adjusting these parameters, such as the manual method (guess and check)) and the Ziegler -Nichols method [8].

In the manual method, the gains of each component are adjusted using trial and error. For this, the effects that each parameter causes the controller output must be known. In this method , the terms K_i and K_d are set to zero, and the term K_p is increased until the cycle output starts to oscillate. From there, the term K_i should be slowly raised to reduce the steady error. At this point the term K_d is incremented, in order to decrease the oscillations at the cycle output. The discussion on

other methods of parameter adjustment is beyond the scope of this paper, since we used the manual adjustment method.

IV. The Proposed Solution

Our proposal is based on a cloud computing environment based on containers and a method of auto elasticity to comply with a required system response time. For this purpose, we use a closed loop system with a PID controller, which responds to changes to system response time by increasing or decreasing the number of containers in a load balancing cluster that process web requests.

In the next subsections, we describe the tools that were integrated and the implementation of the algorithm to provide the features that enable our proposal.

A. Docker

Docker started as a project of the PaaS company (Platform as a Service) dotCloud in 2013 [12] proposing to be an integrator and facilitator for adoption of containers in production environments and in large scale. Docker uses kernel features to isolate the containers from the server, creating isolated processes, network and privileges. The limitation and accounting of resources (CPU, memory, disk space and I/O) is made through the use of cgroups. Also the use of the file system is done efficiently because it is based on copy-on-write, which allows changes to a container to be simply a differential update of the previous image.

One of the greatest advantages of Docker is the ability to find, download and start images of containers that were created by other developers very quickly and conveniently.

B. Kubernetes

Kubernetes is a system developed by Google [6], and made available to the community, which aims to manage the life cycle of containers in the nodes of a cluster. Thus, Kubernetes is an orchestrator of containers, being able to schedule the launch of containers between the nodes of a cluster, to do admission control of containers, resource balancing and provides scalability to the environment. Kubernetes also provides features such as service discovery between containers, service publication for access from outside the cluster and load balancing between containers [6].

The infrastructure of a Kubernetes cluster is composed of master nodes that control worker nodes, which run the containers. All settings of the cluster are stored in a distributed configuration repository, called Etcd. PODs are the basic unit within Kubernetes. Containers are grouped in PODs and these generally represent an application. These are created using Replication Controllers which are used to define PODs that can be scaled horizontally. Replication Controllers are also responsible for maintaining the desired number of PODs active in a cluster.

C. Apache Spark, Flume, HAproxy and Redis

Apache Spark is a distributed processing tool, ideal for processing large databases. It was developed by AMPLab (UC Berkeley) and performs data processing in memory by default.

Its basic structure of abstraction are the RDDS (Resilient Distributed DataSets), which are collections of elements that can undergo operations in parallel, making it possible to generate new RDDS from transformations such as map, reduce, filter and join on RDDS [16]. This tool was chosen to integrate the solution because it allows the solution to perform processing of large databases in text format, in a scalable way.

Apache Spark offers an API called Spark Streaming, which allows real-time data processing, through the creation of structures called DStreams (discretized streams), which are sequences of RDDs. The creation of DStreams is made by the StreamingContext class where you can configure the duration of each window of DStreams [16]. In our proposal, the duration of each window was set to 5 seconds and the motivations for this are further detailed in the next section.

In our proposal, the Spark Streaming is used to process the load balancer logs, collecting the response time of each of the requests that the system serves, in real time. This response time information is stored in a time series format on a Redis server, so that it can be used by the auto scaling algorithm proposed in this article.

Spark receives the log entries of the load balancer in text format, using the Flume tool. Flume is a service that aggregates, collects and moves large volumes of data flow. For its operation it creates a source that receives the data of interest. This source is connected to a channel, where the data will travel toward a sink [3].

Therefore, the function of this tool in the solution, is to send, in real time, the load balancer access logs to Spark, through the creation of a source of type Syslog, which receivers the Load Balancer logs, and a memory channel that carries the source data in memory. From there, Spark consumes this data stream through an Avro Sink, which travels over the network.

The load balancer used in the solution is HAproxy, which can act as a layer 4 or 7 load balancer, SSL terminator, reverse proxy and other [17]. Currently, this is the load balancer used by web sites as Reddit, Stack Overflow, Server Fault, Instagram among others, and has been chosen as the cloud load balancer of Red Hat's OpenShift. HAProxy has the following log format:

May 18 06:24:25 10.125.7.229 haproxy[1078]: 10.125.8.252:43839 [18/May/2016:06:24:24.988] cherrypy cherrypy/10.125.7.227 0/0/2/26/28 200 169 - - —- 1/1/1/0/0 0/0 "GET /generate HTTP/1.0"

In the above line, there are informations such as the waiting time in queue at the application server, *http* method and response code, among others. The part with *0/0/2/26/28* contains the information: *Tq '/' Tw '/' Tc '/' Tr '/' Tt*, where *Tr* is the time in milliseconds that the load balancer waits until it receives a complete response of a web request to the server [17]. So this represents the total time of the request processing by the container.

In our proposal, the SparkStreaming processes the log entries and separates the field *Tr* and stores it in the Redis database, in a time series format. SparkStreaming is also responsible for converting the format of the date of each line to the number of seconds from 0 hour of every day, to support the creation of time series.

Redis is used to store the time series, because it can provide low latency both for writing and reading, as it keeps the data as memory structures [14]. The integration of the solution with Redis is made through the use of the Kairos library, which creates a structure for storing time series in databases such as Redis, Mongo, SQL or Cassandra [9]. This library provides features, such as setting the number of entries to keep in the database and the minimum time unit of interest of a series. In the case of this work we keep stored in the series data of the last 600 seconds, which is sufficient for the algorithm operation.

Kairos also allows the calculation of statistical parameters of the series, using configurable time windows. This allows, for example, to compute average response time of an application in the last two minutes. The minimum unit of time set in this solution is 1 second. This allows good flexibility for configuring the time windows for statistical calculations and enables the generation of graphics with good time resolution for monitoring and evaluation of the solution.

D. The Cloud Architecture

The proposed architecture is shown in Figure 1 and described in the PAS pseudocode in this section. The PID controller is used to maintain the average response time of a particular application within a certain threshold. Our proposed architecture, hereby called as PAS (PID based Autoscaler) operates based on the following sequence:

1) Establishment of an average time threshold (setpoint) desired to the system to answer requests. The monitor receives the response time of the requests arriving at the load balancer;
2) The monitor sends the average response time (the average of the last 200 seconds) so that the PAS algorithm calculates the number of containers needed to reach the setpoint. The average of the last 200 seconds was defined because it was found experimentally that this value is adequate since avoids that outlier values influence on the operation of the system.
3) PAS runs algorithm 1 and inform the desired number of containers to Kubernetes. The current number of containers is obtained using a Kubernetes tool called kubectl, and the average response time of the cluster is determined using the time series present in the Redis database.
4) Kubernetes creates, mantains, or removes new containers, and ensures that the environment will remain with the desired number of containers until the next round of the algorithm (in this case, after 10 seconds). This value of 10 seconds was chosen because it was found that it is sufficient for Kubernetes to load new containers.

E. Solution Operation

In order to allow the operation of the algorithm which works with dynamic data received in real time, we use the processing flow shown in Figure 2. Haproxy acts as the load balancer of the solution. Its logs are sent to Flume that puts the data in a memory channel and sends it to Spark Streaming,

Algoritmo 1: PAS

Input: Average response time of the cluster, Current number of containers
Output: Desired number of containers

begin
- Read the desired threshold of average response time of the requests: $t\_ms\_desired$
- Read the current number of containers: $n\_containers\_current$
- Read the average response of the cluster in ms: $t\_ms\_current$
- Calculate the error: $e(t)$ = $t\_ms\_desired$ - $t\_ms\_current$
- Calculate the PID output:

$$u(t) = K_p e(t) + K_i \int_0^t e(t)\delta t + K_d \frac{d}{dt} e(t) \quad (1)$$

- $n\_desired\_containers$=$n\_containers\_currrent$ + $u(t)$

end
return *n_desired_containers*

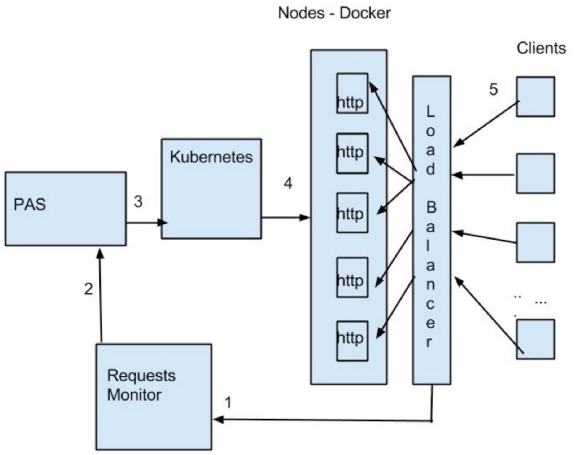

Figure 1. PAS Architecture

thus providing the information necessary for operation of the algorithm.

Haproxy balances the requests in round robin mode between the Docker/Kubernetes nodes hosting the containers. When the Docker/Kubernetes node receives the request, the Kubernetes proxy performs load balancing between the containers of each server.

So, two levels of load balancing are performed, one between the Docker/Kubernetes nodes, where Haproxy performs the load balancing, and other internally within the Docker/Kubernetes nodes where the service Kubernetes proxy performs the load balancing.

We have developed in this research a set of specific codes to customize the interaction between the tools, for example, to generate the time series with system response times and for the creation and destruction of containers, among others.

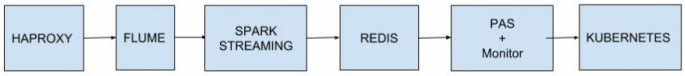

Figure 2. Processing Flow Between the Set of Tools

V. Experimental Results

A. Environment and Evaluation Scenarios

To evaluate the solution we configured a Kubernetes v1.1.2 cluster on the top of Coreos (899.6.0 (2016-02-02)) operating system, virtualised with VMWare ESXi 5.5.0. This environment was configured for solution validation. A production environment could benefit more if the system Coreos was installed directly on physical machines because the virtualization layer would be eliminated.

The cluster was built with the following components: 1 master node (4 vCPUs, 6 GB of RAM) , 1 Etcd node (4 vCPUs, 6 GB of RAM) and 4 worker nodes (4 vCPUs, 6 GB of RAM). Ubuntu 14.04.3 LTS Virtual machines (VMWare ESXi 5.5.0), with the following settings and tools: 1 haproxy 1.5.4 node (4 vCPUs , 4G GB of RAM) , 1 Spark 1.5.2 + Redis 2.8.4 node (2 vCPU 10 GB of RAM) and 1 Flume 1.7.0 + PAS node (2 vCPU , 4 GB RAM)

We defined 4 evaluation scenarios. For the scenarios 1, 2 and 3, we generated an image of a container that runs the web server Cherrypy 5.1.0. We configured a link on this server. The link generates in each request an array of random size between 1,000 and 10,000 elements.

The service publication in Kubernetes was made through the creation of a Replicaction Controller and the configuration of a service of the type NodePort. The HAProxy load balancer was configured to balance requests between the Docker/Kubernetes nodes using the IP addresses of the nodes and ports published by the service of the type NodePort. Each container had its processing and memory resources limited to 18 MB of RAM and 24 millicores of CPU.

Scenario 4 was evaluated with a more elaborate Web system than the arrays generator of scenarios 1, 2 and 3. The evaluation was made using the workload Rubis [15], which is modeled to be a clone of eBay (www.ebay.com). Rubis implements the basic features of ebay: product registration, sale, bidding, browsing products by region (United States) and categories. The installed version of Rubis 1.4.3 was obtained in https://github.com/sguazt/RUBiS.

In the tests we used the PHP version of Rubis and a MySQL 5.5 database. MySQL was installed in a virtual machine with Ubuntu 14.04.1 (16 vCPU and 4 GB of RAM). MySQL has been configured to allow caching of Rubis tables. These high settings of CPU and RAM of the MySQL virtual machine were carried out to ensure that there would be no bottlenecks in access to the application database, since the purpose of the tests is to test Web service auto scaling. The database was populated from the dump obtained in http://download.forge.ow2.org/rubis/rubis_dump.sql.gz.

As in the configuration described for scenarios 1, 2 and 3, the service publication in Kubernetes was made through the creation of a Replicaction Controller and the configuration of a NodePort service. The HAproxy load balancer was configured to balance requests between the Docker/Kubernetes nodes using the IP addresses of the nodes and ports published by the NodePort service. Each container had its processing and memory resources limited to 500 MB of RAM and 160 millicores of CPU.

The PID in the PAS algorithm utilized the following parameters: Kp = 0.016 , Ki = 0.000012 and Kd = 0.096, which was set after several tests with the workloads, adjusting the parameters using the manual method, or *guess and check* , as described in section III-C.

B. Workload

The workload generation was made using "ab" tool (apache bench). In order to generate a load with a realistic profile we collected a set of accesses of the Portal da Transparência (www.transparencia.gov.br), between May/2016 and June/2016. The captured time series in the range of 1 second is the number of accesses on that time interval. The series was characterized with Kettani-Gubner method [10]. The self-similarity and long dependence of the series was confirmed with the Hurst parameter H = 0.87 in the scales of 1 second, 10 seconds and 100 seconds.

A sample of the obtained series can be seen in Figure 3. This series is used to generate in every second, simultaneous requests directed to the IP address of the load balancer which distributes requests to the nodes of the Kubernetes cluster. The workload intensity was set at 3 levels by multiplying the time series by 1, 1.5 and 2, and preserving the same self-similarity index. These loads are referred in the experiments as load_1 , load_1.5 and load_2 .

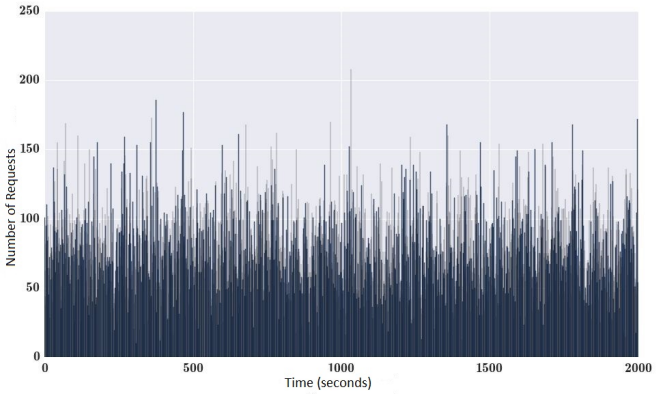

Figure 3. Workload Sample, H=0.87

1) Scenario 1: In this scenario the response time threshold at the load balancer (*setpoint*) was set at 50 ms and we applied load_1 and load_1.5. Figure 4 shows the system response time while under load_1. The graph shows the adjustment caused by the allocation of containers performed by the PAS algorithm and stability achieved close the setpoint of 50 ms.

Figure 5 shows the allocation of containers. The number of containers at the beginning of the experiment was equal to 2. The system allocated the required number of containers so that the average response time shown in Figure 4 reached

the setpoint. At the end of the run there were 26 allocated containers .

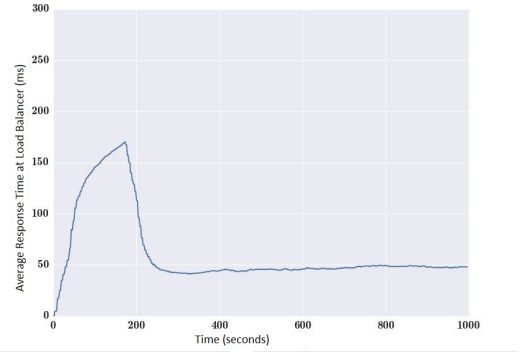

Figure 4. Response Time (ms) x Time (s), load_1, *setpoint* 50 ms

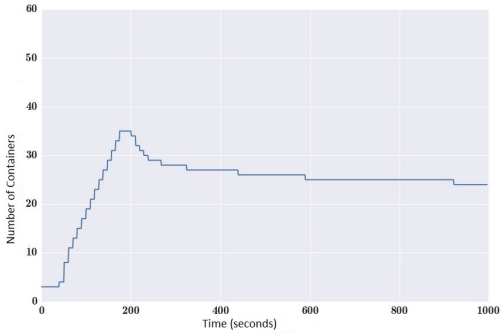

Figure 5. Number of Containers x Time (s), load_1, *setpoint* 50 ms

Figure 6 shows the average response time of the system while under load_1.5 and the setpoint kept at 50 ms. Figure 7 shows the containers allocation during this test. It is worth noting that the container allocation was adjusted to keep the average response time of the system near the setpoint. At the end of the run, 52 containers were allocated. In this case, the system under a greater load, allocated more containers to keep the response time within the defined threshold.

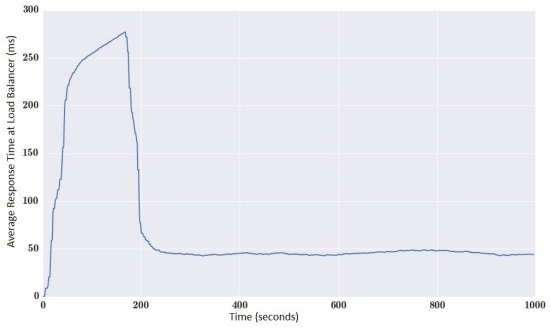

Figure 6. Response Time (ms) x Time (s), load_1.5, *setpoint* 50 ms

2) Scenario 2: In this scenario, the response time threshold in the load balancer (*setpoint*) was set at 50 ms and we applied load_1 for 1000 seconds, then load_1.5 for another 1000 seconds (starting at second 1001), and then we applied load_1 for more 1000 seconds (starting at second 2001). The

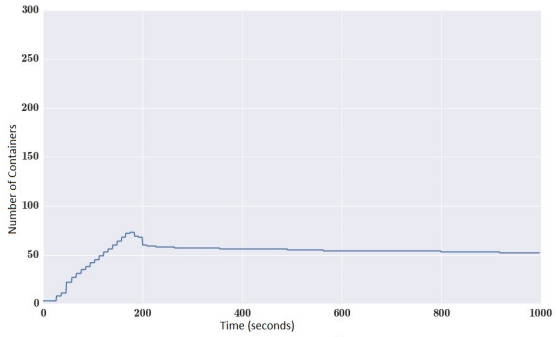

Figure 7. Number of Containers x Time (ms), load_1.5, *setpoint* 50 ms

purpose of this test was to evaluate the behavior of the system during sudden intensity changes.

As can be seen in figure 8 and 13, the system is capable to adjust itself increasing the number of containers when the intensity increases and decreases. It is observed that after exposing the system to load_1.5 at second 1000, the response time remains slightly over 50 ms, and after applying load_1 at second 2000 the response time remains slightly below 50 ms. Better results could be obtained for this case, readjusting the parameters of the PID controller.

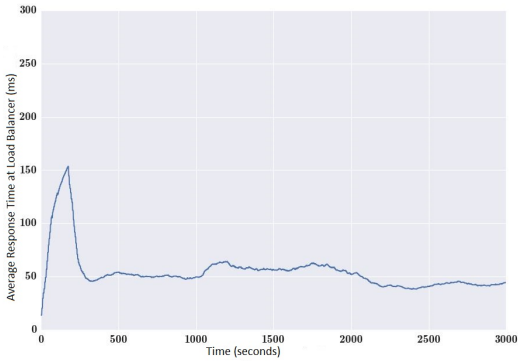

Figure 8. Response Time (ms) x Time (s), Variable load, *setpoint* 50 ms

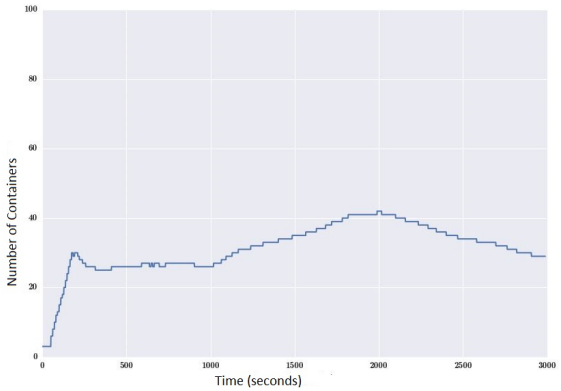

Figure 9. Response Time (ms) x Time (s), Variable load, *setpoint* 50 ms

3) Scenario 3: Scenario 3 compares the proposed algorithm in this paper with the HPA [7].

The comparison follows this procedure: the system configured with the HPA_80 (configured to scale when the average consumption of the containers of a given Replication Controller is above 80 %) is exposed to load_1, load_1.5 and load_2 during 1000 seconds. At the end of the test the average waiting time of the requests at the application layer (customer perspective) is observed.

From these data, it was defined a *setpoint* to use with the algorithm (PAS) for comparing with the response time close to the HPA reference. The results will be compared by checking the average amount of containers allocated during the experiments and the average response times achieved in the customer application layer.

As can be seen in Figure 10 the average response time in the application layer with the HPA algorithm for each load were: 66.18 ms for load_1, 110.12 ms for load_1.5 and 144.01 ms to load_2 .

For comparative purposes, the PAS *setpoints* was configured to deliver an average response time close to those obtained with the HPA. For this, we configured *setpoints* slightly below those observed in the HPA, as the *setpoint* is controlled in the load balancer, so the time measured in the customer application layer should be slightly higher. The values set for *setpoints* are: 50 ms (PAS_50) for the load_1 , 80 ms (PAS_80) for load_1.5 and 100 ms (PAS_100) for load_2 .

Figures 10 and 11 show that for response times near the value obtained by HPA, PAS system allocated less *containers* than HPA. The comparative of the *container* allocation shows that: PAS_50 allocated 44.02 % of which was allocated by the HPA_80, PAS_80 allocated 36.07 % of which was allocated by the HPA_80 to load_1 .5 and PAS_100 allocated 12.72% of which was allocated by the HPA_80 to load_2 .

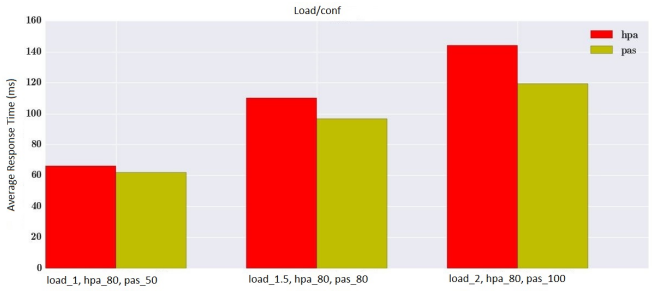

Figure 10. Comparison between HPA and PAS (Average Response Time of the System)

4) Scenario 4: In scenario 4 we evaluated the behavior of the PAS algorithm in the Rubis environment. We used loads_1.5. To generate request variability at every second the accesses to the links is divided as follows: 10 percent home page access, 10 percent of queries to the list of products with random category and random region, 40 percent of visits to random products and 40 percent of queries to random user profiles.

In this test the *setpoint* is set to 50 ms and it is applied load_1.5. Figure 13 shows the container allocation during the test, needed to control the application response time (*setpoint* = 50 ms) for load_1.5, and figure 12, shows the response

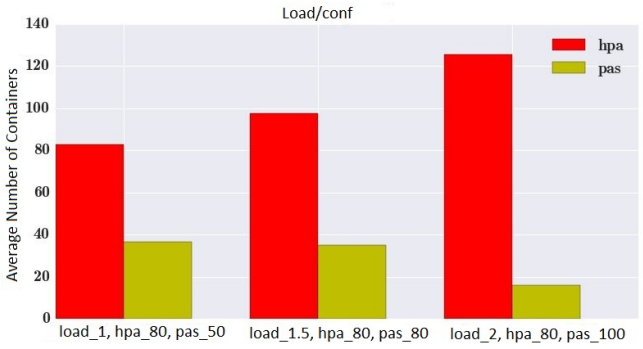

Figure 11. Comparison between HPA and PAS (Average number of containers)

time controlled near the *setpoint*. As can be seen, even with a much more varied workload than the array generator, PAS can control the average response time of the application close to the threshold .

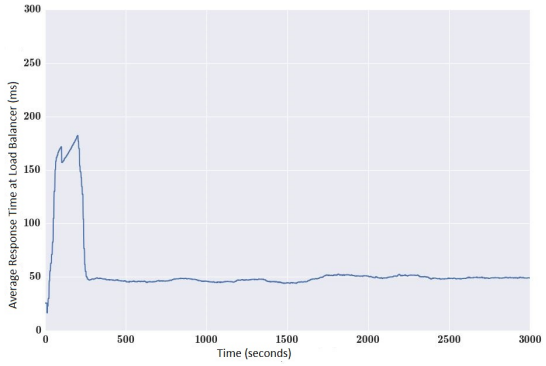

Figure 12. Rubis Response Time (ms) x Time (s), load_1.5, *setpoint* 50 ms

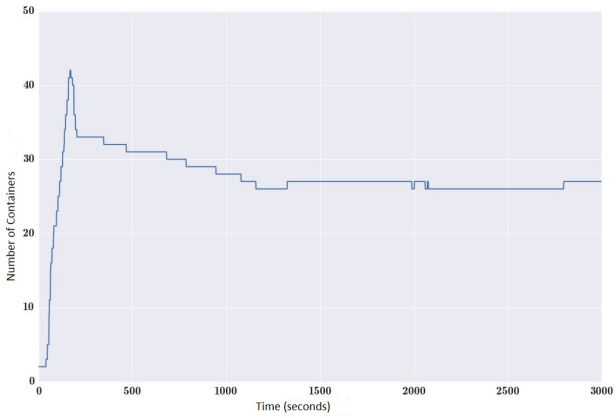

Figure 13. Rubis Number of Containers x Time (s), load_1.5, *setpoint* 50 ms

C. Analysis of Results

The experimental results show that for all the scenarios our proposal is efficient for resource allocation. In the scenario (V-B3), the PAS algorithm optimizes the allocation of the

number of containers to hold the values of *setpoints* within a given threshold. The HPA allocates a greater number of containers to achieve equivalent threshold response times for requests in the application layer. This result shows that the allocation of a larger number of containers can increase the complexity of load balancing time and not necessarily produce better response times.

The obtained results show that the PAS algorithm proposed in this work has the potential to promote an optimization of the number of allocated containers in a cloud computing environment.

The tested scenarios show that the PAS algorithm is a viable alternative to promote auto elasticity to comply with a required response time. Furthermore, performing measurements outside the container, allows PAS to be a generic tool for providing auto elasticity in cloud systems.

VI. Conclusion and Future Works

This paper presented an algorithm based on response time to scale containers on a Cloud Computing system. The proposal defines a cloud computing architecture based on containers and uses a PAS algorithm (PID based Autoscaler) to optimize resource allocation.

The proposal was evaluated in 4 scenarios using different workloads characterized from real world applications. The results shows that our proposal has the potential application to provide auto elasticity in cloud computing systems based on containers. The comparison with the native tool of Kubernetes, the HPA shows a higher efficiency for the PAS proposal.

In future work we intend to improve the PAS algorithm with sophisticated methods for setting the PID parameters. Furthermore the algorithm will be tested with other container orchestrators, such as Mesos and Docker Swarm, to verify that PAS can be a generic tool to provide auto elasticity in cloud computing environments based on containers.

References

[1] Mitchell Anicas. Mitchel Anicas an introduction to haproxy and load balancing concepts. https://www.digitalocean.com/community/tutorials/an-introduction-to-haproxy-and-load-balancing-concepts, 2014.

[2] Docker. Docker the definitive guide to docker containers. https://www.Docker.com/sites/default/files/WP-%20Definitive%20Guide%20To%20Containers.pdf, 2016.

[3] Flume. Flume flume user guide. https://flume.apache.org/FlumeUserGuide.html, 2016.

[4] Katja Gilly, Carlos Juiz, and Ramon Puigjaner. An up-to-date survey in web load balancing. *World Wide Web*, 14(2):105–131, 2011.

[5] Zhenhuan Gong, Xiaohui Gu, and John Wilkes. Press: Predictive elastic resource scaling for cloud systems. In *Network and Service Management (CNSM), 2010 International Conference on*, pages 9–16. IEEE, 2010.

[6] Google. Google container cluster manager from google. https://github.com/kubernetes/kubernetes, 2016.

[7] Google. Google horizontal pod autoscaler. https://github.com/kubernetes/kubernetes/blob/release-1.2/docs/design/horizontal-pod-autoscaler.md, 2016.

[8] National Instruments. National Instruments explicando a teoria pid. http://www.ni.com/white-paper/3782/pt/, 2015.

[9] Kairos. Kairos time series data storage in redis, mongo, sql and cassandra. https://pypi.python.org/pypi/kairos, 2015.

[10] Houssain Kettani, John Gubner, et al. A novel approach to the estimation of the hurst parameter in self-similar traffic. In *Local Computer Networks, 2002. Proceedings. LCN 2002. 27th Annual IEEE Conference on*, pages 160–165. IEEE, 2002.

[11] Tania Lorido-Botrán, José Miguel-Alonso, and Jose Antonio Lozano. Auto-scaling techniques for elastic applications in cloud environments. *Department of Computer Architecture and Technology, University of Basque Country, Tech. Rep. EHU-KAT-IK-09*, 12:2012, 2012.

[12] Nick Martin. Nick Martin a brief history of docker containers' overnight success. http://searchservervirtualization.techtarget.com/feature/A-brief-history-of-Docker-Containers-overnight-success, 2015.

[13] Rishabh Poddar, Anilkumar Vishnoi, and Vijay Mann. Haven: Holistic load balancing and auto scaling in the cloud. 2015.

[14] Redis. Redis redis documentation. http://redis.io/documentation, 2015.

[15] Rubis. Rubis rubis: Rice university bidding system. http://rubis.ow2.org/, 2009.

[16] Spark. Spark spark programming guide. https://spark.apache.org/docs/1.5.2/programming-guide.html, 2015.

[17] Willy Tarreau. HAProxy haproxy configuration manual. http://www.haproxy.org/download/1.5/doc/configuration.txt, 2015.