PAS-CA: Uma Arquitetura Auto-escalável para Ambientes Web de Alta Demanda

Marcelo Cerqueira de Abranches[1,2], **Priscila Solis**[1], **Eduardo Alchieri**[1]

[1]Departamento de Ciência da Computação – Universidade de Brasília (UnB)
[2]Diretoria de Sistemas e Informação – Controladoria-Geral da União (CGU)

Abstract. *This paper proposes a autoscalable architecture based on containers and a self-elasticity algorithm that uses the response time as threshold for requests in a Web system. The proposal promotes the optimization of the processing of requests through efficient allocation of containers. An evaluation was carried out with a workload characterized by a time series resulting from the requests of a Web system in production and of massive use. The results show that the proposal meets the performance requirements by allocating more efficiently the number of containers than other solutions already on the market.*

Resumo. *Este artigo propõe uma arquitetura auto-escalável baseada em containers e um algoritmo de auto-elasticidade que utiliza como limiar o tempo de resposta das requisições em um sistema Web de alta demanda. A proposta promove a otimização do processamento das requisições por meio da alocação eficiente de containers. Uma avaliação foi realizada com uma carga de trabalho caracterizada a partir de uma série temporal resultante das requisições de um sistema Web em produção e de uso massivo. Os resultados mostram que a proposta cumpre os requisitos de desempenho ao alocar com maior eficiência o número de containers quando comparado com outras soluções existentes no mercado.*

1. Introdução

A computação em nuvem é essencialmente um agrupado de computadores conectados em rede em locais geográficos iguais ou diferentes, operando em conjunto para atender clientes com diferentes necessidades e carga de trabalho sob demanda, com o uso de técnicas de virtualização [Zhang et al. 2010].

A tecnologia da nuvem se direciona a uma maior distribuição através de multi-nuvens e a inclusão de vários dispositivos, como no caso de IoT (*Internet of Things*) e a integração de rede no contexto de nuvem de borda e *fog computing*. As soluções de virtualização leve, como a utilização de *containers*, são benéficas para esta configuração com dispositivos menores. Além de oferecer benefícios sobre as máquinas virtuais tradicionais na nuvem em termos de tamanho e flexibilidade, os *containers* são especificamente relevantes para as aplicações de Plataforma como Serviço (PaaS) [Felter et al. 2015].

Recentemente, o uso dos *containers* de código aberto denominados Docker [Docker 2016] se consolidou como um padrão *de facto* para que as aplicações possam estender-se de uma plataforma para outra, funcionando como microserviços em servidores Linux e ambientes PaaS. Atualmente, muitos dos principais fornecedores de infraestrutura de nuvem utilizam o Docker, por exemplo Google Inc., IBM Corp., Red Hat Inc.,

Microsoft, Rackspace, VMware Inc. [Schwartz 2014]. Recentemente, a Google lançou o GKE (*Google Container Engine*) que utiliza Kubernetes, o qual automatiza o processo de provisionamento, execução e parada de máquinas virtuais e o processo de implantação de aplicações em um número variável de *containers* com base na demanda [Avran 2014].

Este trabalho tem como objetivo propor e avaliar o desempenho de uma arquitetura autoescalável e um algoritmo para auto-elasticidade baseado na alocação eficiente do número de *containers* para atender requisições em um sistema *web* de alta demanda na nuvem. O artigo está estruturado da seguinte forma: a Seção 2 apresenta a revisão bibliográfica e os trabalhos relacionados, a Seção 3 descreve a proposta da arquitetura que posteriormente é analisada experimentalmente na Seção 4. Finalmente, a Seção 5 apresenta as conclusões e trabalhos futuros.

2. Revisão Bibliográfica e Trabalhos Relacionados

2.1. Virtualização, Gerenciadores de Nuvem e Automação de Configuração

A virtualização utiliza *software* para emular as funcionalidades de *hardware*. Isto permite que vários sistemas operacionais e aplicações distintas sejam executados em um mesmo servidor [VMware 2016]. Uma máquina virtual é uma porção de software isolada, com um sistema operacional e aplicações associadas. Uma camada de software chamada *hypervisor*, que é responsável pela execução das máquinas virtuais e pelo gerenciamento do uso dos recursos físicos, implementa independência entre as máquinas virtuais.

Para provisão de funcionalidades destes ambientes, podem ser utilizados gerenciadores de nuvem [Sefraoui et al. 2012] os quais normalmente se comunicam com os *hypervisors*, ativos de rede e equipamentos de armazenamento por meio de APIs (*Application Program Interfaces*). As ferramentas de automação de configuração permitem que se defina e aplique a configuração desejada para um conjunto de servidores [Walberg 2008].

2.2. Containers

Containers são uma tecnologia para a criação de instâncias de processamento isoladas que permitem a virtualização no sistema operacional. Dois *containers* em execução no mesmo sistema tem sua própria abstração de camada de rede, memória e processos [Docker 2016]. Esse isolamento é feito via *Kernel namespaces*. Um *namespace* é uma abstração que permite que os processos dentro do *namespace* tenham sua própria instância isolada do recurso global. Mudanças em um *namespace* são percebidas apenas por processos que são membros daquele *namespace* [Linux 2016]. O *Linux* implementa *namespaces* para sistema de arquivos, processos, rede e usuários [Felter et al. 2015].

A limitação e contabilização de recursos dos *containers* é feita por meio dos *cgroups*. Os *cgroups* permitem a alocação de recursos por meio de grupos de processos definidos pelo usuário em um sistema [RedHat 2016]. Os *containers* têm uma maior portabilidade que as máquinas virtuais, ao serem configurados de forma genérica para qualquer sistema operacional baseado em Linux.

2.3. Carga de Trabalho

O comportamento dos acessos web apresenta comportamento monofractal [Crovella and Bestavros 1997].

Os processos monofractais ou autossimilares apresentam dependência de longa duração. Um processo com dependência de longa duração tem função de autocorrelação $r(k) \sim k^{-\beta}$ com $k \to \infty$, onde $0 < \beta < 1$. O parâmetro de Hurst indica o grau de autossimilaridade de uma série, dado por: $H = 1 - \frac{\beta}{2}$. 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 [Crovella and Bestavros 1997]. Existem vários métodos para a estimativa do parâmetro de Hurst (*H*). Um deles é o método de Kettani-Gubner [Kettani et al. 2002] que permite o cálculo do parâmetro *H* utilizando séries de tamanho menor do que as necessárias com outros métodos. Nesse método, o coeficiente de autocorrelação é descrito pela Eq.1.

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

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 parâmetro *H*, conforme a Eq.2.

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

No contexto deste trabalho, as séries autossimilares foram utilizadas para caracterização e geração do perfil da carga usado na avaliação do ambiente.

2.4. Controladores PID

Controladores PID (Proporcional-Integral-Derivativo) são um dos algoritmos de controle mais utilizados na indústria. O valor da variável de controle é lido do sistema *S*. Este valor é comparado com o valor desejado, denominado de *setpoint* e é gerado o termo de erro *e(t)*. O termo de erro é combinado nos componentes proporcional, integral e derivativo e é gerado um sinal de controle que atua no sistema *S* para controlar o valor da variável [Instruments 2015]. O componente proporcional depende da diferença entre o valor desejado (*setpoint*) e o valor atual da variável. Esta diferença é referida como erro. O ganho proporcional (*Kp*) determina a taxa de resposta de saída para o erro e o componente derivativo indica a taxa de variação instantânea deste. O componente integral soma o termo de erro ao longo do tempo.

O ajuste dos parâmetros de ganho K_p, K_i e K_d permite que o controlador PID produza os ajustes necessários. Existem diversos métodos de ajuste destes parâmetros, como o método manual, o método Ziegler-Nichols [Instruments 2015] e o ajuste utilizando o algoritmo *Coordinate Ascent*. 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 com tentativa e erro. 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].

O método *Coordinate Ascent* (CA) [Thrun et al. 2006] estabelece um laço que maximiza ou minimiza uma função de pontuação. Esta função define quão próxima do objetivo uma variável permaneceu dentro de uma iteração do algoritmo. Dada uma estimativa inicial de um conjunto de parâmetros que afeta o valor da função de pontuação, o

algoritmo modifica cada um destes parâmetros com base em um valor fixo e a partir desta modificação, verifica se esta alteração apresentou uma melhora na função de pontuação. Havendo melhora, este parâmetro é registrado, assim como o novo valor da função de pontuação e aumenta-se o passo de forma gradual. Caso contrário, o passo é diminuído gradualmente até que o intervalo de busca do parâmetro seja menor do que um valor mínimo pré-fixado. O pseudocódigo do Algoritmo 1 descreve o CA.

No contexto deste trabalho, um controlador PID é utilizado em um ambiente de malha fechada que permite manter o tempo médio de resposta de uma aplicação constante, independente da carga aplicada, por meio da variação do provisionamento de recursos (*containers*) destinado a atender as requisições.

Algoritmo 1: COORDINATE ASCENT

```
1  início
2      Enquanto a soma dos passos para cada parâmetro > limiar mínimo
3        Para cada parâmetro
4          aumente o parâmetro com o valor do passo e calcule o valor da
           pontuação atual
5        Se pontuação atual > pontuação anterior
6          mantenha o valor de parâmetro, aumente o passo e registre o valor da
           pontuação
7        Se pontuação atual < pontuação anterior
8          diminua o parâmetro com o valor do passo e calcule o valor da
           pontuação atual
9          Se pontuação atual > pontuação anterior
10           mantenha o valor de parâmetro, aumente o passo e registre o valor da
             pontuação
11         Se pontuação atual < pontuação anterior
12           diminua o passo
13 fim
14 retorna Parâmetros otimizados
```

2.5. Trabalhos Relacionados

O trabalho [Lorido-Botrán et al. 2012] compara os diversos métodos de auto-escalabilidade e elasticidade em um ambiente de nuvem. Estes métodos são separados nas categorias reativos e preditivos. Este estudo mostra a variedade de técnicas que estão sendo propostas em diferentes áreas de conhecimento.

O algoritmo PRESS [Gong et al. 2010] propõe a predição de carga de CPU. Um dos métodos apresentado utiliza o processamento de sinais usando-se transformadas rápidas de Fourier para extração de frequências dominantes para gerar séries temporais e diversas janelas de tempo.

O Haven [Poddar et al. 2015] apresenta uma proposta de dimensionamento elástico para ambientes de nuvem que se baseia no monitoramento de cargas de CPU e memória para cada máquina virtual. A partir de limiares previamente estabelecidos para consumo de CPU e memória, o algoritmo instancia novas máquinas virtuais e as insere em um *pool* de balanceamento de carga. O balanceador de carga é implementado com

SDN (*Software Defined Network*) e tem inteligência para realizar o encaminhamento da requisição para membros com melhores condições.

A proposta do presente trabalho se diferencia da abordagem do PRESS pois não realiza predição de carga e 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, aumenta recursos de memória e CPU para se ajustar a carga de trabalho, enquanto que na proposta deste 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 para 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, esta 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 do presente trabalho apresenta um método de dimensionamento do sistema a partir da observação do tempo de resposta das requisições, o que permite uma visibilidade global do comportamento do sistema. 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.

Os trabalhos anteriormente citados analisam ambientes que utilizam tecnologia de máquinas virtuais em que sistemas de escalabilidade devem levar em conta o tempo de provisionamento e configuração de máquina virtual, tempo que pode alcançar minutos, enquanto o provisionamento de *containers* pode alcançar segundos [Martin 2015]. Sistemas que utilizam máquinas virtuais trabalhando de forma reativa num laço de controle como o PAS podem incorrer em violações constantes de nível de serviço, motivo pelo qual técnicas de predição de carga e pré-provisionamento são comuns nestes ambientes ([Lorido-Botrán et al. 2012], [Poddar et al. 2015] e [Gong et al. 2010]).

Sistemas projetados para prover elasticidade em ambientes de máquinas virtuais, podem não ser adequados para escalar ambientes de *containers*. Os parágrafos seguintes analisam sistemas projetados para atuar com *containers* e posicionam a PAS-CA frente a esses sistemas.

O trabalho [Google 2016] propõe uma ferramenta nativa de auto-escalabilidade chamada de *Horizontal Pod Autoscaler* (HPA), a qual escala o ambiente a partir de limiares médios de consumo de CPU dos *containers* para atender às necessidades das aplicações. O HPA faz o uso do *cadvisor* (que deve estar funcional em todos os nós workers do Kubernetes) e do *heapster* para monitoramento do uso de CPU dos *containers*. O PAS-CA se diferencia do HPA no monitoramento pois não necessita do *cadvisor* e do *heapster*. O PAS trabalha de forma independente de qualquer componente de software de monitoramento ao monitorar os tempos de resposta em um ponto único, i.e., no balanceador de carga.

O trabalho [Kan 2016] usa um método reativo e um pró ativo para escalar *containers*. A métrica utilizada é requisições por segundo e o algoritmo utiliza a informação do número de requisições por segundo que um *container* consegue atender. Esta informação é utilizada na previsão do número de *containers* necessários para atender a demanda. Os autores afirmam que este valor pode ser configurado manualmente, entretanto, o número

de requisições por segundo que um *container* atende depende das condições de CPU e memória e isso pode variar em um ambiente real. O sistema adiciona ou subtrai quantidade fixa de *containers* de acordo com a demanda enquanto que o PAS-CA adiciona ou remove instâncias de acordo com o controlador PID que informa a quantidade de *containers*, que deve ser adicionada ou subtraída a cada iteração do algoritmo de modo a manter o tempo de resposta sob controle. Além disso o PAS-CA não precisa que seja determinado o número de requisições por segundo que um *container* pode atender, o que o torna mais robusto dado que esta métrica pode variar de aplicação para aplicação e com níveis de consumo de recursos dos *containers*. A avaliação do ambiente realizada em [Kan 2016], mostra apenas o número de *containers* variando conforme a demanda e não compara o sistema proposto com outras propostas e com outras métricas de desempenho, por exemplo, número de requisições não atendidas e variações no tempo de resposta da aplicação. No presente artigo o PAS-CA é comparado com a solução dominante de mercado [Google 2016] e avalia diversas métricas de desempenho (número de requisições não atendidas, variações no tempo de resposta da aplicação, quantidade média de *containers* alocados durante os experimentos, entre outras).

O trabalho [Müller et al. 2016] utiliza a métrica de operações por segundo como limiar para escalabilidade ao estabelecer de forma empírica o número de operações por segundo que um sistema consegue atender, o que também apresenta a dificuldade de se estabelecer um valor fixo da métrica em um ambiente dinâmico.

Os trabalhos [Amazon 2016a] e [Amazon 2016b] demonstram como construir e configurar aplicações auto escaláveis dentro do ambiente de nuvem da Amazon. Porém estes trabalhos utilizam ferramentas exclusivas da Amazon (por exemplo: *CloudWatch*). O PAS CA pode ser utilizado em qualquer provedor de nuvem pois utiliza ferramentas de código aberto em sua construção, além de ser independente das ferramentas de monitoramento de terceiros para prover auto escalabilidade. O trabalho [Jesheen 2016] descreve como prover um ambiente escalável utilizando *containers docker*, porém a escalabilidade é realizada de forma manual. O PAS-CA executa a escalabilidade de forma automática.

O trabalho [Solis 2016] propõe o uso de ferramentas de Big Data e teoria de controle para provisão de um ambiente auto escalável, onde o limiar para a escalabilidade é o tempo de resposta médio de uma aplicação. Neste trabalho somente utiliza-se a configuração dos parâmetros do PID de forma manual enquanto que o presente artigo implementa o PAS-CA que complementa a técnica ao empregar a técnica de otimização de parâmetros *Coordinate Ascent*.

3. Solução Proposta

O PAS-CA (PID *based Autoscaler*-CA) é uma arquitetura inovadora em nuvem para provisão de auto-elasticidade. É utilizado um ambiente baseado em *containers* e um método de alocação que reage aos aumentos no tempo de resposta do sistema, de modo a manter esse tempo abaixo de um limiar pré-estabelecido. É utilizado um sistema de malha fechada com um controlador PID que reage às variações no tempo de resposta do sistema em conjunto com um algoritmo de otimização de parâmetros do controlador PID.

O PAS-CA é apresentada na Figura 1 e funciona da seguinte forma: (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 balan-

ceador; (2) O monitor de requisições envia o tempo médio de resposta para o dimensionador PID que calcula o número de *containers* necessários para atingir ou permanecer no *setpoint*. (3) O PID executa o Algoritmo 2 e informa o número desejado de *containers* ao Kubernetes. (4) Kubernetes cria ou remove novos *containers*, além de garantir que o ambiente permanecerá com o número desejado até a próxima rodada do algoritmo (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.

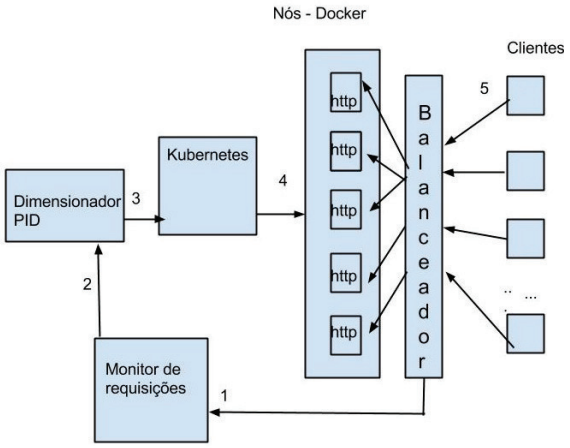

Figura 1. Arquitetura da Solução

Algoritmo 2: PAS

Entrada: Tempo médio de resposta do cluster, Número atual 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 do controlador PID:

$$u(t) = K_p e(t) + K_i \int_0^t e(t)\delta t + K_d \frac{d}{dt} e(t) \quad (3)$$

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

fim

retorna *n_desejado_containers*

Para viabilizar a operação do algoritmo que pretende trabalhar com dados dinâmicos recebidos em tempo real, utiliza-se o fluxo de processamento mostrado na Figura 2 que descreve a integração e uso das ferramentas usadas na implementação da arquitetura da Figura 1. O *HAproxy* é um balanceador [Tarreau 2015] utilizado por *Reddit, Stack Overflow/Server Fault*, Instagram e Red Hat *OpenShift*. Seus *logs* são enviados ao *Flume* que agrega os dados em um canal de memória e os envia ao *Spark Streaming*. Para a disponibilização dos dados utilizados pelo dimensionador, o *Flume* realiza o envio em tempo

real dos *logs* de acesso do balanceador de carga. Esta informação do tempo de resposta é armazenada em formato de série temporal no servidor *Redis*. O *Redis* é utilizado para armazenar a série temporal. Com estes dados disponíveis no Redis, o PAS, nas versões manual e automática (CA) é executado e determina o número desejado de *containers*. O Kubernetes recebe a informação do número desejado de *containers* e utiliza a ferramenta *kubectl* para ajustar a alocação.

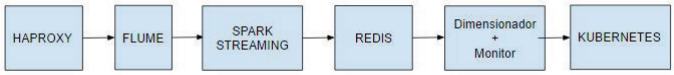

Figura 2. Fluxo de processamento entre o conjunto de ferramentas utilizadas

Para otimizar os parâmetros PID, o algoritmo *Coordinate Ascent* minimiza a soma do erro quadrático médio da série dos tempos médios de resposta em relação ao valor estabelecido no *setpoint*. São estabelecidos valores iniciais para os parâmetros *Kp, Ki* e *Kd* e um tamanho de passo inicial para cada um dos parâmetros. Também é estabelecido um valor inicial para o erro quadrático médio que é a função de pontuação, que neste caso é o quadrado da soma das diferenças dos tempos de resposta em relação ao *setpoint* dentro da janela de tempo. A janela foi definida em 2000 segundos pois verificou-se experimentalmente que esse tempo seria suficiente para ajustar cada um dos parâmetros do PID na função de pontuação.

4. Avaliação da Proposta e Resultados Experimentais

4.1. Ambiente

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. 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). Foram instaladas máquinas virtuais (*VMWare ESXi* 5.5.0) 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 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).

Os cenários de avaliação utilizam o ambiente Rubis [Rubis 2009], que é modelado como um clone do ebay (`www.ebay.com`). O RUBiS implementa as funcionalidades básicas de registro de produtos, venda, lances de leilão, navegação em produtos por região e categorias. A versão instalada do RUBiS foi a 1.4.3 obtida em [Squazt 2012]. O *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* do Kubernetes. Cada *container* teve seus recursos de processamento e memória limitados a 300 milicores e 300 MB de memória RAM.

4.2. Carga de Trabalho

Foi coletado um conjunto de acessos, entre os dias 25/05/2015 e 25/06/2015, do portal da transparência (`www.transparencia.gov.br`), na escala de 1 segundo. A série foi

caracterizada com o método Kettani-Gubner na qual foi confirmada a autossimilaridade com H=0,87, nas escalas de 1 segundo, 100 segundos e 600 segundos.

Esta série foi utilizada para gerar a cada segundo requisições simultâneas direcionadas ao endereço IP do balanceador de carga, o qual distribui as requisições entre os nós do *cluster* Kubernetes. A intensidade da carga foi multiplicada por vários índices preservando a autossimilaridade. Apesar de terem sido realizados testes com várias cargas, neste trabalho por restrição de espaço mostraremos os resultados para uma carga em que a série temporal é multiplicada por 3 (carga_3) e para uma carga variável que funciona da seguinte forma: o sistema é submetido a uma carga_1 (série multiplicada por 1) por 1000 segundos, em seguida a uma carga_2 (série multiplicada por 2) por 1000 segundos, depois a carga_3 por mais 1000 segundos, seguidas novamente pelas carga_2 e carga_1, por mais 1000 segundos cada. Esta carga variável também foi chamada de carga_1_2_3_2_1.

Quando configurado de forma manual, o controlador 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. Para a otimização dos parâmetros do PID com o algoritmo *Coordinate Ascent*, os valores para cada um dos limiares podem ser apreciados na Tabela 1. Para a descoberta dos parâmetros para os limiares de 80, 100 e 120 ms do PAS, o *Coordinate Ascent* foi executado da seguinte forma: (1) os parâmetros Kp, Ki e Kd foram configurados em 0.01; (2) o passo de atualização foi configurado para 0.001; (3) o limiar de convergência da soma dos passos foi configurado para 0.000001; (4) o sistema foi exposto à carga_3 e esperou-se a convergência do algoritmo para cada um dos casos.

4.3. Avaliação de Resultados

Esta seção apresenta a comparação dos resultados do PAS (com ajuste manual) e do PAS-CA (com ajuste automático) com o HPA, ferramenta comercial desenvolvida pela Google. Para cada teste, o experimento foi repetido com 10 amostras diferentes. O intervalo de confiança foi definido em 95%, com 9 graus de liberdade. Em todos os testes foi utilizado o ambiente Rubis. Os limites de alocação de recursos para estes testes foram configurados como 300 *millicores* de processamento e 300 MB de memória RAM. Estas configurações foram estabelecidas para permitir que o Rubis sempre fosse iniciado em boas condições de consumo de CPU e processamento. O número mínimo de *containers* em todos os testes foi 3 para permitir que sempre houvesse pelo menos um *container* executando em cada nó.

As configurações do HPA, PAS e PAS-CA assim como as nomenclaturas utilizadas, podem ser consultadas na Tabela 1. Para o HPA foram configurados limites de processamento médio máximo desejado em 20% e 50%. Essas configurações de limiares de 20% e 50% foram definidas pois o HPA escalando com 20% de processamento permite bom desempenho de tempo de resposta, mantendo os *containers* em um nível seguro de processamento. o HPA escalando com 50% de processamento permite uma alocação menor de *containers*, piorando os tempos de resposta das aplicações, porém ainda com um nível seguro de processamento, sem que hajam travamentos dos *containers*. Limiares acima de 50% causam altos níveis de processamento ao submeter o ambiente às cargas utilizadas neste trabalho, o que origina travamento dos *containers*, *reset* de conexões e *timeouts*.

Para o PAS e PAS-CA é configurado o limite de tempo de resposta e os parâmetros

do PID (*Kp*, *Ki* e *Kd*). Para as configurações dos algoritmos PAS 80, PAS 100 e PAS 120, são usados os parâmetros encontrados de forma manual. Já para as configurações PAS 80 CA, PAS 100 CA e PAS 120 CA são usados os parâmetros calculados pelo *Coordinate Ascent*. Os limiares configurados para o PAS foram escolhidos para trazerem tempos de resposta compatíveis aos das configurações utilizadas para o HPA.

Algoritmo	CPU Livre	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

Tabela 1. Configuração dos Algoritmos

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, percentagem de requisições não atendidas, número médio de *containers* alocados durante os testes e eficiência média na alocação dos *containers*. 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 por esse tempo. A eficiência média na alocação de *containers* foi definida pela Eq. 4, em que *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.

$$E = 1/(Nc * T) \tag{4}$$

As numeração das colunas das tabelas 2 e 3, apresentam a seguinte descrição: (1) intervalo de confiança para o tempo médio de resposta em milisegundos na camada de aplicação, (2) intervalo de confiança para a alocação média de *containers*, (3) intervalo de confiança para a percentagem de requisições não atendidas; (4) intervalo de confiança para a eficiência do algoritmo; (5) intervalo de confiança para o tempo médio de resposta em milisegundos medido no balanceador.

Os resultados dos experimentos realizados com a carga_3 por um tempo de 1200 segundos são mostrados na Tabela 2. Observa-se que o tempo de espera dos clientes, do PAS 80 CA foi próximo ao HPA 20, porém o PAS 80 CA apresentou um intervalo de confiança menor, o que demonstra a previsibilidade de comportamento em relação ao tempo de resposta para a carga_3. Verifica-se também que o HPA 50 alocou menos *containers*, porém apresentou maior tempo de resposta.

Nestes testes, a criação e destruição de *containers* executada pelos algoritmos de auto elasticidade não representou problemas críticos em relação ao não atendimento de requisições, visto que os intervalos de confiança para porcentagem de requisições não

atendidas apresentam valores baixos. Observa-se que as versões do PAS CA tenderam a perder menos requisições do que as versões otimizadas manualmente.

A tabela 2 também mostra que as configurações das versões do PAS conseguiram manter o tempo médio de resposta no balanceador próximo aos limiares desejados. É possível observar também que os algoritmos HPA tenderam a ter intervalos de confiança maiores do que os do PAS, o que demonstra uma maior previsibilidade no atendimento de requisitos de tempo de resposta do PAS-CA. O resultado mostra que as versões PAS-CA obtiveram melhores tempos de resposta que as versões otimizadas com parâmetros manuais, além de eficiência melhor ou igual. Também para um tempo de resposta equivalente ao obtido pelo HPA 20, o PAS 80 CA obteve melhor eficiência.

	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

Tabela 2. Resultados dos testes para carga_3 com 95% de confiança

Para verificar a diferença entre o PAS e o HPA com a carga variável (carga_1_2_3_2_1), foram realizados testes para as configurações PAS 80 CA e HPA 20. Os resultados obtidos estão sumarizados na Tabela 3. Nesta tabela, verifica-se que o tempo de resposta médio na camada de aplicação e a alocação média de *containers* durante os testes, foram melhores para o PAS 80 CA. Verifica-se também que os dois algoritmos obtiveram tempos médios de resposta próximos, dentro do intervalo de confiança, entretanto, o intervalo de confiança obtido para os tempos de resposta apresenta-se menor para o PAS 80 CA, o que indica uma maior previsibilidade de desempenho em relação ao HPA 20. O PAS 80 CA também alocou um número menor de *containers*.

	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

Tabela 3. Resultados dos testes para carga variável com 95% de confiança

Conforme os dados da tabela 3, o PAS 80 CA apresentou melhor eficiência que o HPA 20. Este comportamento é coerente com os dados obtidos nos testes anteriores, já que o PAS 80 CA teve tempos de resposta próximos aos do HPA 20, porém alocou menos *containers*. A percentagem de requisições não atendidas, foi baixa para tanto para o PAS 80 CA quanto para o HPA 20, o que demonstra a robustez dos dois algoritmos para cargas com mudanças abruptas de intensidade. A versão PAS 80 CA obteve tempos médios de resposta no balanceador menores que o HPA 20, o qual estava configurado para manter este valor próximo a 80 ms, porém verifica-se que este valor ficou próximo de 70 ms. Isto pode ser explicado pelo fato de que nestes testes com carga variável, o sistema passa 2000 segundos sendo submetido a carga_1, e foi observado nos diversos testes que nessa carga o tempo de resposta esteve abaixo de 80 ms, mesmo sem que o sistema seja escalado.

4.4. Testes de média zero

Para comparação dos resultados foram realizados testes de média zero para verificar estatisticamente os resultados dos diferentes sistemas dentro de um intervalo de confiança [Jain 1990]. Os testes realizados nesta seção foram configurados para apresentarem intervalo de confiança de 95%, usando-se a distribuição *t student* com 9 graus de liberdade. Os dois sistemas a serem comparados são o HPA 20 e o PAS 80 CA pois são as configurações com maiores requisitos de desempenho dentre os cenários avaliados. Foram comparados os tempos de resposta, alocação de *containers* e eficiência para a carga_3 e para a carga variável.

O resultado do teste para o tempo de resposta é mostrado na Tabela 4 (as linhas representam o "sistema 1" e as colunas o "sistema 2"). Para a carga_3, o intervalo mostra valores negativos e positivos quase simétricos, o que indica a equivalência dos dois sistemas. Para a carga variável o intervalo mostra tempos de resposta equivalentes, porém o HPA 20 mostra uma tendência para tempos de resposta maiores. A Tabela 5 apresenta a alocação de *containers* em que para ambas as cargas, o HPA 20 fez uma alocação maior de *containers*. A Tabela 6 compara a eficiência entre os algoritmos. Para ambas as cargas o PAS 80 CA foi mais eficiente. Todos os resultados nesta seção são coerentes com os resultados apresentados na seção anterior, o que valida a precisão do método de avaliação utilizado neste trabalho.

	carga_3		**carga variável**	
Sistema 1/Sistema 2	**HPA 20**	**PAS CA 80**	**HPA 20**	**PAS CA 80**
HPA 20	(0.0, 0.0)	(-20.77, 22.44)	(0.0, 0.0)	(-0.5108585, 5.26436)
PAS 80 CA	(-22.44, 20.77)	(0.0, 0.0)	(-5.26436, 0.510859)	(0.0, 0.0)

Tabela 4. Testes média zero para tempo de resposta

	carga_3		**carga variável**	
Sistema 1/Sistema 2	**HPA 20**	**PAS CA 80**	**HPA 20**	**PAS CA 80**
HPA 20	(0.0, 0.0)	(0.2767679, 0.414394)	(0.0, 0.0)	(0.0589784, 0.097679)
PAS 80 CA	(-0.4143941, -0.276768)	(0.0, 0.0)	(-0.0976786, -0.058978)	(0.0, 0.0)

Tabela 5. Testes média zero para número médio de *containers*

	carga_3		**carga variável**	
Sistema 1/Sistema 2	**HPA 20**	**PAS CA 80**	**HPA 20**	**PAS CA 80**
HPA 20	(0.0, 0.0)	(-6.53e-05, -6.5e-05)	(0.0, 0.0)	(-8.95e-05, -8.9e-05)
PAS 80 CA	(6.53e-05, 6.5e-05)	(0.0, 0.0)	(8.95e-05, 8.9e-05)	(0.0, 0.0)

Tabela 6. Testes média zero para eficiência

5. Conclusões e Trabalhos Futuros

Este trabalho apresentou a PAS-CA, uma arquitetura de auto escalabilidade para computação em nuvem que propõe o uso de *containers* e a provisão da auto elasticidade com base em uma métrica que impacta diretamente a percepção do usuário, o tempo de resposta da aplicação. Foram integradas diversas ferramentas de *Big Data*, orquestradores de *containers* e técnicas de controle. Os resultados experimentais mostram que nos cenários avaliados a proposta otimiza a alocação do número de *containers* para manter o tempo

de resposta das aplicações dentro de um limiar estabelecido. Foi feita uma comparação da proposta com uma ferramenta da Google, o HPA, e os resultados mostram que a PAS-CA tem potencial para ser uma alternativa de escalabilidade em sistemas de nuvem. Nos cenários avaliados, a PAS-CA obteve melhor eficiência na alocação de *containers* que o HPA. Trabalhos futuros pretendem integrar outras métricas de interesse para provisão de auto-elasticidade, como limiar de CPU e memória e a avaliação da proposta em um sistema em produção.

Referências

Amazon (2016a). Aws, tutorial: Scaling container instances with cloudwatch alarms. `http://docs.aws.amazon.com/AmazonECS/latest/developerguide/cloudwatch_alarm_autoscaling.html`. [Acessado em 29/03/2017].

Amazon (2016b). Deploying elastic beanstalk applications from docker containers. `http://docs.aws.amazon.com/elasticbeanstalk/latest/dg/create_deploy_docker.html`. [Acessado em 29/03/2017].

Avran, A. (2014). Google announces cloud container engine using kubernetes. `https://www.infoq.com/news/2014/11/google-cloud-container-engine`. [Acessado em 16/11/2016].

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

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

Felter, W., Ferreira, A., Rajamony, R., and Rubio, J. (2015). 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.

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

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

Jain, R. (1990). *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling*. John Wiley & Sons.

Jesheen, H. (2016). Auto scaling with docker. `https://botleg.com/stories/auto-scaling-with-docker/`. [Acessado em 29/03/2017].

Kan, C. (2016). Docloud: An elastic cloud platform for web applications based on docker. In *Advanced Communication Technology (ICACT), 2016 18th International Conference on*, pages 478–483. IEEE.

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.

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

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). A brief history of docker containers' overnight success. http://searchservervirtualization.techtarget.com/feature/A-brief-history-of-Docker-Containers-overnight-success. [Acessado em 03/04/2016].

Müller, C., Truong, H.-L., Fernandez, P., Copil, G., Ruiz-Cortés, A., and Dustdar, S. (2016). An elasticity-aware governance platform for cloud service delivery. In *Services Computing (SCC), 2016 IEEE International Conference on*, pages 74–81. IEEE.

Poddar, R., Vishnoi, A., and Mann, V. (2015). 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.

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

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

Schwartz, J. (2014). Are containers the beginning of the end of virtual machines? https://virtualizationreview.com/articles/2014/10/29/containers-virtual-machines-and-docker.aspx. [Acessado em 23/11/2016].

Sefraoui, O., Aissaoui, M., and Eleuldj, M. (2012). Openstack: toward an open-source solution for cloud computing. *International Journal of Computer Applications*, 55(3).

Solis, M. A. P. (2016). 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.

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

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

Thrun, S., Montemerlo, M., Dahlkamp, H., Stavens, D., Aron, A., Diebel, J., Fong, P., Gale, J., Halpenny, M., Hoffmann, G., et al. (2006). Stanley: The robot that won the darpa grand challenge. *Journal of field Robotics*, 23(9):661–692.

VMware (2016). What is virtualization. http://www.vmware.com/solutions/virtualization.html. [Acessado em 16/11/2016].

Walberg, S. (2008). Automate system administration tasks with puppet. *Linux Journal*, 2008(176):5.

Zhang, Q., Cheng, L., and Boutaba, R. (2010). Cloud computing: state-of-the-art and research challenges. *Journal of internet services and applications*, 1(1):7–18.