Replicação de Máquinas de Estado em *containers* no Kubernetes: uma Proposta de Integração

Hylson Netto[1,2], Lau Cheuk Lung[1] , Miguel Correia[3], Aldelir Fernando Luiz[2]

[1]Departamento de Informática e Estatística – Universidade Federal de Santa Catarina

[2]Campus Blumenau – Instituto Federal de Educação, Ciência e Tecnologia Catarinense

[3]INESC-ID – Instituto Superior Técnico – Universidade de Lisboa – Portugal

hylson.vescovi@posgrad.ufsc.br, aldelir.luiz@blumenau.ifc.edu.br,
lau.lung@ufsc.br, miguel.p.correia@tecnico.ulisboa.pt

Resumo. *A virtualização de computadores permitiu um provisionamento dinâmico de recursos nos data centers e um consequente modelo de cobrança de serviços de acordo com utilização. A virtualização em nível de sistema por meio dos containers tornou mais dinâmica essa capacidade de provisionamento. Em aplicações que requerem garantias mais restritivas de acordo e ordenação, ferramentas de coordenação podem ser utilizadas em nível de aplicação para esse fim. Este artigo propõe uma integração do serviço de coordenação na arquitetura do sistema Kubernetes, um orquestrador de containers, com o objetivo de manter os containers os menores possíveis e oferecer à aplicação a capacidade de replicar estados automaticamente. Um protocolo que utiliza memória compartilhada disponível no Kubernetes foi desenvolvido para atuar na arquitetura, e uma avaliação foi realizada para demonstrar a viabilidade da proposta.*

Abstract. *Computer virtualization brought fast resource provisioning to data centers and also the deployment of a pay-per-use cost model. The system virtualization provided by containers has improved this versatility of resource provisioning. Applications that require more restrictive agreement and ordering warranties can use specific tools to achieve their goals. This paper proposes the integration of coordination services in the Kubernetes architecture in order to maintain containers as small as possible and offer automatic state replication. A protocol that uses shared memory available in Kubernetes was developed, and an evaluation was conducted to show the viability of the proposal.*

1. Introdução

A utilização de máquinas virtuais em *data centers* tornou prático o provisionamento dinâmico de recursos. Por consequência, foi possível implantar o modelo de custos sob demanda (*pay-per-use*). O número de usuários conectados à Internet cresce cada vez mais, fazendo surgir clientes em potencial para a variedade de serviços de computação disponíveis nos *data centers*. Esse conjunto de serviços disponíveis em *data centers* é conhecido como *cloud computing*, definido pelo Instituto Americano Nacional de Padrões e Tecnologia [Mell and Grance 2011] como "um modelo para permitir acesso ubíquo, conveniente e sob-demanda para recursos de computação compartilhados e configuráveis (por exemplo, redes, servidores, armazenamento, aplicações e serviços) que podem ser rapidamente provisionados e liberados com esforço de gerenciamento mínimo ou por um serviço de interação

com o provedor". No contexto de provisionamento, os *containers* popularizaram-se em 2013, tornando ainda mais rápida a capacidade de alocar e liberar recursos em *data centers*. A implementação de *containers* mais conhecida na atualidade é o Docker (`docker.com`).

O uso de *containers* pode causar um impacto positivo nos *data centers*. Para organizar esse desenvolvimento, várias empresas se reuniram, e em Julho de 2015 fundaram a *Cloud Native Computing Foundation* (`cncf.io`), cujos objetivos incluem estabelecer padrões para a operacionalização de *containers* em diversos provedores de nuvens. A Google utiliza *containers* há alguns anos e lançou, como um primeiro resultado da CNCF, o sistema Kubernetes para orquestrar *containers* Docker em *clusters*. O Kubernetes é *open-source*, tem sido desenvolvido colaborativamente e traz consigo conhecimento dos engenheiros do Borg [Verma et al. 2015], o atual gerenciador de *containers* da Google.

O Kubernetes possui a capacidade de replicar *containers* a fim de aumentar a disponibilidade da aplicação hospedada no *container*. Quando um *container* falha, o Kubernetes elimina o *container* faltoso e instancia outro a partir de uma imagem. Nesse processo, o estado da aplicação hospedada no *container* é perdido. Aplicações podem utilizar volumes externos para persistir o estado da aplicação, mas é preciso proteger estes volumes contra falhas. Além disso, no caso de replicar estados da aplicação, o acesso ao volume deve ser organizado pela aplicação para lidar com questões de concorrência.

Algumas aplicações requerem garantias mais restritivas de acordo e ordenação, como aplicações de trabalho cooperativo ou aplicações que realizam processamento replicado por razões de tolerância a faltas. Em ambientes de rede local, os algoritmos mais eficientes são derivados do Paxos [Lamport 1998]. O sistema ZooKeeper [Hunt et al. 2010] e o algoritmo Raft [Ongaro and Ousterhout 2014] são utilizados na prática para criar máquinas de estado replicadas [Schneider 1990]. Estas soluções podem ser utilizadas em nível de aplicação no Kubernetes, porém ocuparão espaço em todos os *containers* participantes da replicação, além de onerar a aplicação com o esforço de realizar a coordenação.

Para fornecer ao Kubernetes a possibilidade de lidar com estados replicados, a abordagem de integração [Lung et al. 2000] é apropriada pois está alinhada com a natureza colaborativa de desenvolvimento do Kubernetes. Integrar significa construir ou modificar um sistema existente, adicionando componentes para complementar funcionalidade. Outras abordagens são possíveis, como interceptação ou serviço, porém a integração é transparente ao usuário, sendo também a abordagem que apresenta melhor desempenho, segundo experiências realizadas no âmbito do CORBA [Lung et al. 2000]. Assim sendo, este trabalho apresenta duas contribuições. A primeira é uma arquitetura de suporte a replicação de máquinas de estado no sistema Kubernetes, utilizando um componente pré-existente no Kubernetes como memória compartilhada para orientar a coordenação entre *containers*. A segunda contribuição é um protocolo denominado ONSET (OrdeNação Sobre mEmória comparTilhada) que opera na arquitetura proposta.

O artigo está organizado da seguinte forma: a Seção 2 apresenta conceitos sobre *containers* e o sistema Kubernetes. Na Seção 3 é detalhada a proposta, em termos de protocolo e arquitetura. A Seção 4 apresenta a avaliação da proposta. Na Seção 5 são abordados os principais trabalhos relacionados e por fim a Seção 6 conclui o artigo.

2. *Containers* e Kubernetes

Certos componentes do Linux tornam possível realizar virtualização em nível de sistema. O *control group* (*cgroups*) permite controlar o acesso de recursos pelos processos. Um componente chamado *namespaces* permite isolar processos e recursos hierarquicamente em subníveis. *Ambiente* é o conjunto de processos e recursos, e ambientes criados em níveis inferiores não pode acessar ambientes de níveis superiores[1]. Por fim, um componente que consiste de um sistema de arquivos em camadas permite que dentro do ambiente isolado seja instalado um sistema operacional (SO), mas que efetivamente sejam gravados apenas os arquivos que não existem no SO do *host*. Este ambiente isolado, gerenciável e com arquivos em camadas é atualmente denominado *container*. O *cgroups* está no *kernel* do Linux desde 2007, mas foi reprojetado em 2013, fato que potencializou a criação de gerenciadores de *containers*, como o Docker. *Containers* são máquinas virtuais cujo estado não é persistente, e são instanciados a partir de imagens. As imagens são pequenas, pois utilizam sistema de arquivos em camadas. Por isto, a criação de *containers* é muito rápida e o provisionamento de recursos torna-se mais eficiente em comparação às máquinas virtuais tradicionais.

Uma rede que interliga *containers* hospedados em diferentes máquinas e o monitoramento de *containers* são funcionalidades necessárias em um *cluster*. A Google criou o sistema Kubernetes (`kubernetes.io`) a fim de integrar soluções para permitir o gerenciamento completo. Engenheiros da Google compõem a equipe principal do Kubernetes, que herdou conceitos do Borg [Verma et al. 2015], o atual gerenciador de *containers* da Google. No Borg há um componente chamado *Alloc* que mantém na mesma máquina *containers* que tem alto acoplamento e por isso precisam compartilhar recursos. Considere o exemplo de um servidor web e a aplicação que monitora/analisa os *logs* produzidos pelo servidor web. Estas aplicações podem estar localizadas em *containers* diferentes, de forma a facilitar a substituição ou a atualização de uma das aplicações. Ambos *containers* compartilham os *logs*, que são produzidos pelo servidor web e consumidos pelo analisador de logs. No Kubernetes, o componente que mantém *containers* relacionados próximos é o POD, e possui objetivos semelhantes ao *Alloc*. A Figura 1 ilustra a arquitetura do Kubernetes.

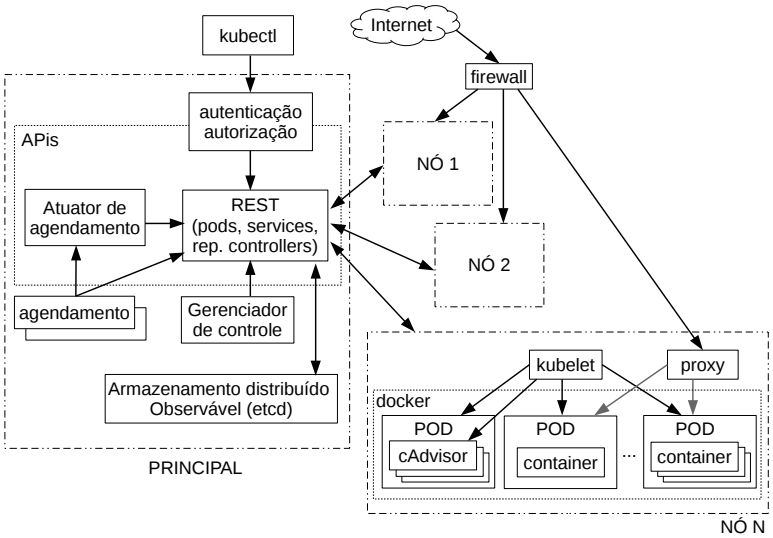

Figura 1. Arquitetura do Kubernetes.

[1]O comando `chroot` do Linux é um exemplo de implementação de espaço de nomes.

Um *cluster* Kubernetes é formado de nós, que são as máquinas (físicas ou virtuais) nas quais *containers* são hospedados. A máquina principal contém componentes gerenciais, e pode ser replicada para tolerar faltas de parada. O POD é uma unidade mínima de gerenciamento no Kubernetes, e pode conter um ou mais *containers*. PODs recebem endereço de rede e são alocados em nós. *Containers* que estão no mesmo POD compartilham recursos e são mantidos no mesmo nó. Clientes contactam o *cluster* por meio de um *firewall*, que geralmente distribui pedidos aos nós segundo uma estratégia de balanceamento de carga. O *proxy* recebe pedidos de clientes e encaminha os pedidos ao POD. Se o POD estiver replicado, um balanceamento de carga interno ao nó distribui aleatoriamente o pedido a uma das réplicas. O *kubelet* gerencia PODs, imagens, *containers* e outros elementos do nó. O *cAdvisor* monitora recursos utilizados pelo *container*. O *kubelet* repassa as informações de monitoramento à máquina principal, que deve atuar quando necessário.

As informações de controle do Kubernetes são armazenadas na máquina principal. O armazenamento de dados é observável, o que torna possível notificar clientes quando dados são modificados. A ferramenta *etcd* (`coreos.com/etcd`) implementa esse armazenamento no Kubernetes. Componentes acessam dados armazenados por meio de APIs REST. O *replication controller* mantém réplicas de PODs operantes, a despeito de falhas. O *kubectl* é uma *interface* de comando pela qual o operador pode criar PODs, verificar o estado do *cluster*, entre outras operações. O componente de agendamento determina em quais nós os PODs serão instanciados. O atuador de agendamento realiza a instanciação e a destruição de PODs. O gerenciador de controle atua em nível de *cluster*, realizando funções como descoberta e monitoramento de nós. Por fim, componentes de segurança permitem definir formas de autenticação e autorização de acesso do *kubectl* ao *cluster*.

A replicação de *containers* no Kubernetes utiliza monitoramento para eliminar ou criar *containers* e manter ativa a quantidade de réplicas definida. O balanceamento de carga realizado pelo *proxy* também favorece a disponibilidade da aplicação. Entretanto, existem aplicações que necessitam de garantias mais restritivas de acordo e ordenação. Aplicações de trabalho cooperativo ou aplicações que realizam processamento replicado por razões de tolerância a faltas são alguns exemplos. Estes requisitos podem ser alcançados com a utilização, em nível de aplicação (dentro dos *containers*), de ferramentas como o ZooKeeper [Hunt et al. 2010] e o algoritmo Raft [Ongaro and Ousterhout 2014]. Porém, retirar da aplicação a responsabilidade de realizar essa tarefa pode reduzir o tamanho da imagem do *container* e reduzir a complexidade no interior do *container*. O presente artigo é o primeiro passo nesta direção: integrar a coordenação de estados de *containers* ao Kubernetes.

O Kubernetes usa um armazenamento observável, implementado pela ferramenta *etcd*, para persistir as informações sobre o gerenciamento dos *containers*. É possível utilizar o *etcd* para coordenar pedidos, reaproveitando o recurso que já está disponível no Kubernetes. O *etcd* pode ser usado como uma memória compartilhada, e protocolos de replicação de estados apoiados em memória compartilhada geralmente apresentam baixo número de mensagens de rede, em comparação aos protocolos que não usam memória compartilhada. O *etcd* utiliza o RAFT para distribuir o armazenamento, tornando a memória compartilhada confiável e disponível. Um trabalho recente [Toffetti et al. 2015] utiliza o *etcd* de forma distribuída para persistir estado e agregar confiabilidade a microserviços.

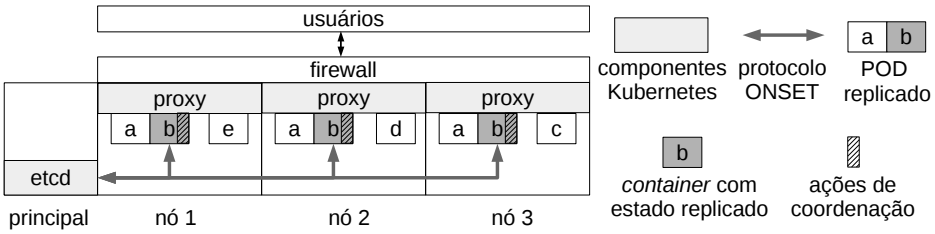

Figura 2. Arquitetura para Integração de Coordenação no Kubernetes.

3. Integração de Coordenação no Kubernetes

O Kubernetes é uma ferramenta *open-source*, o que possibilita o desenvolvimento de extensões de forma irrestrita. Proposta semelhante de modificação em arquitetura foi realizada no contexto do CORBA [Bessani et al. 2005], quando várias empresas reuniram-se para criar o padrão FT-CORBA. A proposta de incorporar coordenação de serviços na arquitetura Kubernetes considera a abordagem de integração [Lung et al. 2000], onde componentes podem ser modificados ou novos componentes podem ser adicionados à arquitetura.

A Figura 2 apresenta a proposta de integração de coordenação no Kubernetes. Nós são máquinas (físicas ou virtuais) do *cluster* Kubernetes. A máquina principal contém componentes de gerência do Kubernetes. Um *firewall* encaminha pedidos aos nós, e o *proxy* encaminha o pedido um *container* hospedado no nó. O *etcd* é o componente de armazenamento do Kubernetes, e será usado pelos *containers* como memória compartilhada. Nesta proposta, ações de coordenação encontram-se nos *containers* que possuem estado replicado.

3.1. Modelo de Sistema

O sistema é composto por *containers* replicados, identificados na Figura 2 pela letra b. Estes *containers* possuem estados replicados, devido à execução, na mesma sequência, de todos os pedidos que os *containers* recebem [Schneider 1990]. Assume-se que até f *containers* possam falhar por parada. Dessa forma, devem existir pelo menos $2f+1$ réplicas no sistema. A parada de nós também é tolerada, sendo que neste caso *containers* são realocados para outro nó automaticamente pelo Kubernetes.

A memória compartilhada (MC) é assumida como um repositório chave-valor que pode ser acessado por operações de leitura e escrita. *Containers* de estado replicado utilizam a MC como forma de comunicação. A MC possui capacidade de notificar clientes inscritos sobre eventos, como atualização de um valor e criação de novos valores. A MC também possui a capacidade de associar um identificador único sequencial a cada informação gravada. A MC pode ser distribuída em vários nós, no intuito de prover confiabilidade. A comunicação entre clientes e *containers* segue o modelo de sincronia parcial [Dwork et al. 1988], no qual existem períodos estáveis de comunicação síncrona. A comunicação entre os *containers* e a MC é assíncrona.

3.2. Interação entre componentes da arquitetura

O diagrama de sequência na Figura 3 mostra a interação do cliente com os *containers*. Considere b_i o *container* b replicado no nó i. No início, o cliente envia o pedido P, que será entregue a um dos nós pelo *firewall*. Dentro do nó, P é recebido pelo *proxy* e encaminhado a uma réplica do *container* b. No diagrama, b_1 recebeu P, o que significa que o *firewall* encaminhou P ao nó 1. A seguir, b_1 registra a chegada de P na memória compartilhada (MC), implementada pelo *etcd*. A MC notifica os demais *containers* replicados, que por

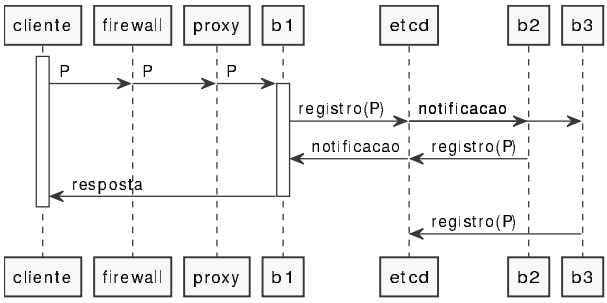

Figura 3. Interação na replicação de estados.

sua vez também farão seus registros de recebimento de P. A MC notifica b_1 sobre os registros feitos pelos demais *containers*. Quando b_1 acumula certo número de notificações, envia a resposta do pedido ao cliente (note que o registro de P por b_3 não é requisito para que b_1 responda ao cliente). A replicação de estados é garantida pois os *containers* replicados executam P na mesma ordem, por meio do protocolo de ordenação detalhado a seguir.

3.3. Protocolo de Ordenação

A ordenação de pedidos ocorre com suporte de memória compartilhada (MC), o que sugeriu o nome do protocolo: ONSET (OrdeNação Sobre mEmória comparTilhada). O ONSET está ilustrado na Figura 4. Qualquer réplica pode receber o pedido do cliente. Esta característica é importante pois o Kubernetes realiza balanceamento de carga automaticamente (em nível de nó, via componente *proxy*), e assim o protocolo torna-se compatível com esse recurso, não limitando o pedido ao recebimento apenas pelo líder. Caso a réplica receptora seja a réplica líder, uma ordem é associada ao pedido e um registro de chegada de pedido (RCP) é criado e inserido na MC. As demais réplicas serão notificadas e registrarão na MC seus respectivos RCPs, o que por sua vez fará com que a MC emita novas notificações. Quando uma réplica acumular $f + 1$ RCPs é possível executar o pedido. Apenas a réplica que recebeu o pedido via *proxy* responde ao cliente o resultado da execução.

Se uma réplica não-líder receber o pedido, essa réplica deve inserir o RCP na MC informando ordem zero, o que significa que o pedido está aguardando manifestação do líder para que seja definida a ordem do pedido. Quando o líder é notificado sobre esse pedido, o líder atribui uma ordem ao pedido e insere na MC o RCP do pedido, desta vez com uma ordem definida. A partir de então, o protocolo comporta-se de maneira semelhante ao caso no qual o líder recebe o pedido. Se o cliente não receber resposta da réplica após um tempo específico, o cliente deve reenviar o pedido. O reenvio poderá ser atribuído à mesma réplica ou a uma outra réplica, dependendo da política de balanceamento de carga utilizada.

A Figura 4 ilustra execuções do ONSET em cenários com 3 réplicas ($S_{0..2}$). Na Figura 4(a), a réplica S_0 é líder, responsável por definir a ordem dos pedidos. Números de sequência (1..10) na Figura 4(a) orientam a descrição que segue. O cliente inicia enviando o pedido P à réplica S_0 (1). A réplica S_0, por ser líder, atribui uma ordem à P e armazena na MC (2) o RCP de P, ou seja, $RCP(S_0, P)$. A MC responde S_0 confirmando a gravação de $RCP(S0, P)$ (3) e notifica as demais réplicas. Quando S_1 recebe a notificação sobre $RCP(S_0, P)$ (4), S_1 cria o seu próprio $RCP(S_1, P)$ e envia à MC (5). A MC responde S_1 (6) e notifica as demais réplicas (7 e 8) sobre $RCP(S_1, P)$. Quando S_1 recebe a resposta da MC (6), S_1 possui $f + 1$ RCPs e então pode executar P (o retângulo azul representa execução). Quando S_0 recebe $RCP(S_0, P)$ (7), S_0 pode executar P e responder ao cliente. Quando S_2 recebe $RCP(S_1, P)$ (8), S_2 pode executar P. Na Figura 4(a), a notificação

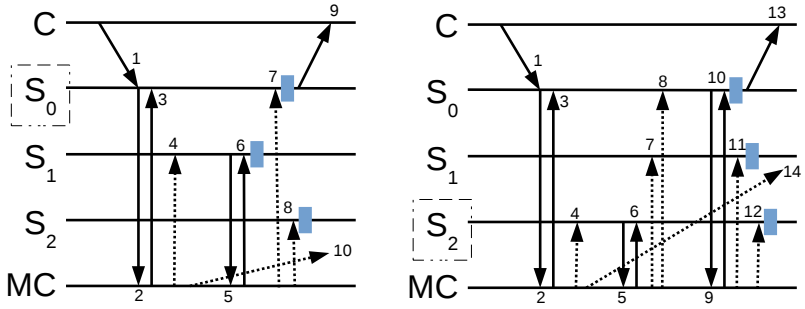

Figura 4. Protocolo ONSET: pedido enviado à uma réplica: (a) líder; (b) não-líder.

$RCP(S0, P)$ para S_2 foi intencionalmente atrasada (10) para mostrar que há casos onde S_2 não precisa enviar RCP à MC, e também para simplificar a ilustração do protocolo.

A Figura 4(b) mostra um cenário no qual um pedido P foi enviado à réplica não-líder S_0 (1). A réplica líder é S_2. Inicialmente, S_0 envia $RCP(S_0, P)$ à MC informando a ordem de P igual a zero. Pedidos com ordem zero permanecem na MC aguardando até que o líder atribua uma ordem. Quando a MC recebe $RCP(S_0, P)$ (2), responde à S_0 (3) e notifica as demais réplicas. Quando S_2 recebe $RCP(S_0, P)$ (4), atribui uma ordem a P e envia à MC $RCP(S_2, P)$ (5). Quando S_1 recebe $RCP(S_0, P)$, ações de comunicação não são executadas, pois S_1 não é líder. A MC responde S_2 (6) e notifica as demais réplicas. Ao receber $RCP(S_2, P)$ (8), S_0 envia $RCP(S_0, P)$ novamente à MC (9) (desta vez, com ordem definida pelo líder). Quando S_0 recebe a resposta da MC (10), S_0 possui $f+1$ $RCP's$, o que permite a S_0 executar P e responder ao cliente (13). Ao receber a notificação da MC, demais réplicas podem executar P (11 e 12). Na ilustração, a notificação de $RCP(S_0, P)$ foi intencionalmente atrasada, dispensando o envio de $RCP(S_1, P)$ pela réplica S_1.

O protocolo ONSET tolera faltas de paradas de réplicas. As Figuras 4(a) e 4(b) apresentam cenários onde a participação das réplicas S_2 e S_1, respectivamente, não influenciam no progresso do protocolo. Faltas de parada nas réplicas apresentariam comportamento semelhante aos casos de atraso representados na Figura 4.

3.3.1. Eleição

Protocolos que utilizam sequenciadores (fixo ou móvel) geralmente utilizam temporizadores para monitorar o funcionamento do líder. No ONSET, temporizadores são acionados no momento em que a réplica percebe a existência de um pedido, seja pelo recebimento direto do cliente ou através de notificações da MC. Se um temporizador expira e o líder ainda não definiu uma ordem para o pedido, a réplica envia à MC uma solicitação de eleição. A MC notificará demais réplicas, que registrarão solicitações de eleição, gerando notificações. Após $f + 1$ solicitações de eleição, uma réplica assume o papel de líder.

O critério de eleição para o novo líder é o seguinte: a primeira réplica que efetuou registro de eleição tornar-se-á o novo líder. Dessa forma, a réplica que possui um pedido atrasado há mais tempo provavelmente será a primeira a solicitar a eleição, e poderá tão logo possível tornar-se líder, ordenar o pedido e promover o progresso do protocolo. Uma observação relevante é que o primeiro líder só é eleito a partir do momento em que o sistema recebe o primeiro pedido. Para certificar-se de que todos os registros de eleição são

conhecidos da réplica (isso é importante para esquivar-se de possíveis inversões na rede de notificações da MC), ao acumular $f + 1$ solicitações de eleição a réplica lê da MC o conjunto completo de solicitações de eleição. Assim, é possível observar com exatidão (de maneira determinística) qual foi o primeiro registro de eleição. Esse critério torna-se possível graças ao identificador único sequencial gerado pela MC (Seção 3.1), cujo menor valor remete ao primeiro registro de eleição efetuado.

Outra característica importante do mecanismo de eleição adotado é que as réplicas não conhecem diretamente a réplica líder (há um certo anonimato), mas sabem de sua existência pela observação do progresso da ordenação dos pedidos. Essa característica é coerente com o ambiente de *containers*, no qual a criação e destruição de réplicas é mais frequente do que nos ambientes tradicionais de máquinas físicas ou virtuais. Apesar de não ser necessário para o funcionamento do protocolo, réplicas podem inferir quem exerce a liderança observando o autor do primeiro RCP de um pedido ordenado na MC.

4. Avaliação

A avaliação desta proposta foi realizada com a execução do protocolo ONSET sobre a arquitetura apresentada (Figura 2). Testes preliminares com o ONSET revelaram intensa atividade no disco do computador principal. Estas atividades em disco são oriundas do armazenamento realizado pelo *etcd*. Supôs-se então que a hospedagem do *etcd* em um diretório montado em memória RAM poderia favorecer o experimento, pois memória RAM é um dispositivo que possui latência e variabilidade de tempo de acesso menores do que disco rígido. Para investigar essa possibilidade, apresenta-se a primeira hipótese:

Hipótese 1: *memória compartilhada (MC) armazenando dados em disco virtual alocado em memória RAM trará resultados mais estáveis que execução da MC usando disco rígido. Ainda, o desempenho será pelo menos 50% melhor, comparado ao uso de disco rígido.*

Alocação de recursos no contexto de *containers* é menos custosa do que alocação usando máquinas virtuais tradicionais. Além disso, o protocolo ONSET possui poucas mensagens, em comparação a outros protocolos, por utilizar memória compartilhada como meio de comunicação entre réplicas. Dessa maneira, é razoável supor que a vazão geral do sistema (quantidade de pedidos executados por segundo) não será bruscamente afetada quando o número de réplicas for incrementado. Pode-se então enunciar a seguinte hipótese:

Hipótese 2: *usando containers, é possível utilizar um número de réplicas pelo menos 4 vezes superior ao cenário clássico de 3 réplicas, para tolerar faltas de parada. Haverá uma diminuição na vazão do sistema, mas esta será gradual, na medida em que o número de réplicas cresce, e não maior que 20% para cada incremento de f, na fórmula $n = 2f+1$.*

4.1. Experimentos

Experimentos foram conduzidos para avaliar as hipóteses apresentadas. Para avaliar a Hipótese 1, o computador principal do *cluster* Kubernetes foi dotado de um disco virtual de 4GB, montado em memória RAM, com uso do comando `mount` do Linux, e o sistema de arquivos `tmpfs`. Foi realizada uma execução do experimento com o *etcd* utilizando o disco rígido, e outra execução com o *etcd* utilizando o disco virtual. A fim de avaliar a Hipótese 2, o número de réplicas foi variado de 1 a 15, seguindo a regra de tolerância a faltas de parada: $n = 2f + 1$, onde n é o número de réplicas presentes no sistema e f é o número máximo de faltas toleradas. Adicionalmente, o número de clientes que realizam pedidos foi variado

de 1 a 64, dobrando-se o número de clientes a cada variação. Foram medidas a latência dos pedidos, em milissegundos, e a duração de cada cenário de execução do experimento, em segundos. Para quantidades de clientes superiores a 2, múltiplas *threads* simularam o acesso simultâneo de vários clientes. Os pedidos consistiram em requisições no estilo REST, com tamanhos que não excederam 50 bytes, e as respostas dos pedidos não excederam 100 bytes. Cada cliente efetuou 50 requisições ao sistema, considerando que um cliente apenas envia um próximo pedido após receber a resposta do pedido atual.

O protótipo do ONSET foi desenvolvido na linguagem de programação Go (`golang.org`), versão 1.4.2, visando uma futura integração no Kubernetes, que também é desenvolvido em Go. Um *cluster* Kubernetes foi montado para avaliar a execução do ONSET em *containers*. A aplicação mantém um contador compartilhado sobre o qual podem operar 3 comandos: *get* para retornar o contador; *inc* para incrementar o contador; e *dou*, uma operação que divide o contador por 2, se o contador for maior que 30. O temporizador que monitora o líder foi configurado para 1 segundo[2]. A aplicação está hospedada em uma imagem de *container* disponível no repositório `hub.docker.com`, sob o localizador `replicatedcalc`. O programas utilizados nos *containers* (réplicas) e no experimento (clientes) estão disponíveis no endereço `hylson.com/kubernetesSBRC2016`.

Os experimentos foram realizados em um *cluster* com quatro computadores com o sistema Kubernetes instalado, e um quinto computador atuando como cliente. Todos os computadores possuem configuração Intel i7 3.5Ghz, QuadCore, cache L3 8MB, 12GB RAM, 1TB HD 7200 RPM. Uma rede ethernet 10/100MBits isolada de tráfego externo conectou os computadores durante a realização dos experimentos. *Containers* que executam em máquinas distintas comunicam-se por meio de uma rede virtual, implementada neste trabalho com o Flannel (`github.com/coreos/flannel`). Os computadores possuem o sistema operacional Ubuntu Server 14.04.3 64 bits, com *kernel* versão 3.19.0-42.

4.2. Resultados e Discussões

A Figura 5 apresenta as latências médias percebidas pelos clientes, em função do número de réplicas executadas em *containers* no *cluster* Kubernetes. Na Figura 5(a) o experimento foi realizado utilizando memória compartilhada sobre um disco virtual em RAM, enquanto a Figura 5(b) refere-se à execução da memória compartilhada usando o disco rígido. Há uma diferença perceptível de desempenho quando o número de réplicas e clientes é pequeno. Especificamente, para poucos clientes (até 16) a utilização de memória RAM proporciona um desempenho melhor em relação ao cenário que utiliza disco rígido. Entretanto, a partir de 32 clientes, a diferença de latência percebida pelos clientes diminui, considerando as duas estratégias (disco virtual e rígido). Foram feitos testes estatísticos de hipótese sobre algumas medidas para aferir a diferença. Por exemplo, para as medidas de latências com 1 réplica e 64 clientes, a diferença entre o método que usa disco virtual em RAM e disco rígido é estatisticamente significativa, apesar de não ser aparente nos gráficos da Figura 5.

Além dos testes estatísticos, foram construídos gráficos específicos de alguns cenários, para melhor visualização. A Figura 6 mostra as latências percebidas por 64 clientes ao acessar 1 réplica (as latências de eleição de líder não estão representadas) e por 32 clientes ao acessar 15 réplicas. Os gráficos do tipo *boxplot* representam a mediana na linha central do retângulo, por isso foram também impressas as médias, representados pela linha de menor

[2]Este valor pode ser variado a fim de verificar limites.

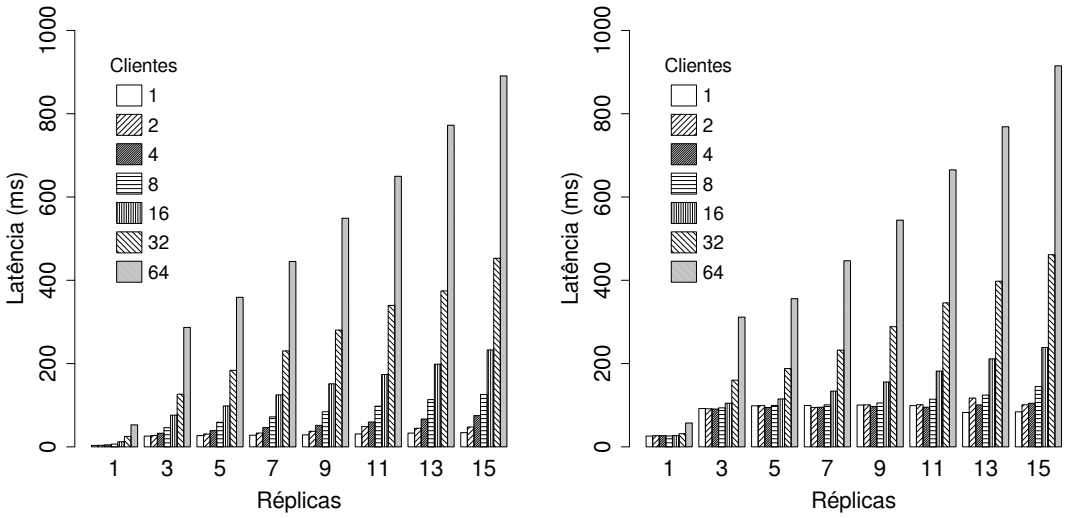

Figura 5. Latência com memória compartilhada em (a) disco virtual RAM, (b) disco rígido.

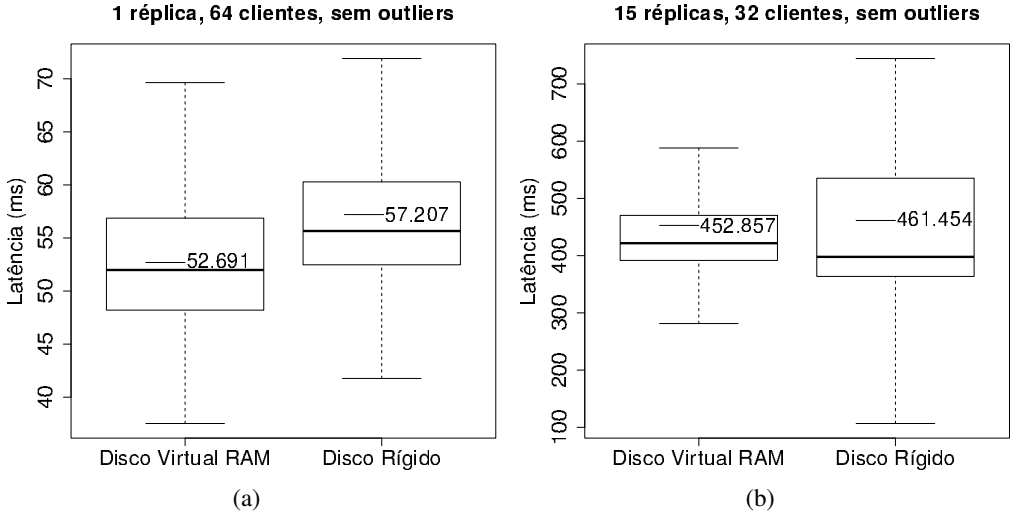

Figura 6. Latências para (a) 1 réplica e 64 clientes, e (b) 15 réplicas e 32 clientes.

comprimento, ao lado do valor da média. Em ambos cenários, a latência média com uso de disco virtual mostra-se menor do que a latência média com uso de disco rígido. Os testes estatísticos comprovam que no caso da Figura 6(a) é possível afirmar um melhor desempenho médio com uso de disco virtual em RAM sobre o disco rígido. Porém, no caso apresentado na Figura 6(b), os testes estatísticos não permitem afirmar a superioridade no uso do disco virtual sobre o disco rígido. Assim sendo, considera-se a Hipótese 1 como falsa por não haver evidências suficientes que comprovem o ganho de desempenho em todos os casos considerados. Os valores de média e desvio-padrão, bem como os testes estatísticos aplicados no exemplo, encontram-se disponíveis em `hylson.com/kubernetesSBRC2016`.

O líder do sistema apenas é eleito após a chegada do primeiro pedido. As latências percebidas pelos clientes destes primeiros pedidos podem ser consideradas medidas *outliers*, pois apresentarão valores superiores à média por conter em si o tempo de eleição do líder. Como exemplo, a Figura 7 mostra, para a execução de 3 réplicas usando memória compartilhada em disco rígido, pontos isolados de grande latência (*outliers*). Esses pon-

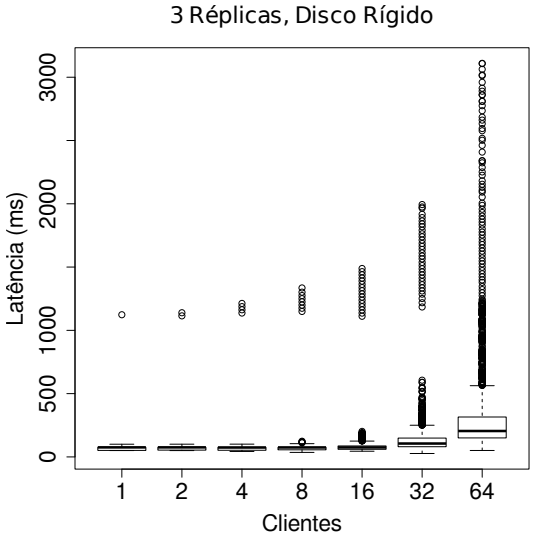

Figura 7. Latências: destaque da eleição de líder (*outliers*).

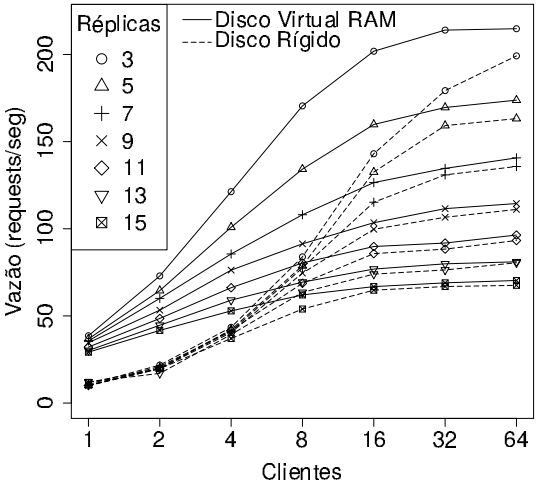

Figura 8. Vazão do sistema.

	Incremento de réplica (quantidades anterior/novo)						
	1/3	3/5	5/7	7/9	9/11	11/13	13/15
Mídia da MC	Percentual de redução da vazão						
Disco Virtual	84,7	16,42	15,05	13,48	12,58	11,7	10,13
Disco Rígido	74,93	8,51	6,8	8,98	7,27	6,86	6,81

Tabela 1. Percentuais médios de reduções de vazão ao incrementar número de réplicas.

tos em destaque carregam consigo, além do tempo de ordenação e execução do pedido, a duração da eleição do líder, que ocorre na chegada dos primeiros pedidos. No detalhe da Figura 7, é possível notar que para 4 clientes há 4 pontos em destaque, correspondente aos pedidos destes 4 clientes que perceberam maior latência devido à eleição do líder.

A Figura 8 apresenta a vazão do sistema em função do número de clientes, réplicas e forma de uso da memória compartilhada. Quando o número de réplicas cresce ocorre uma redução na vazão, mas esta perda diminui na medida em que o número de réplicas cresce, até a quantidade de réplicas avaliada no experimento (15). A Tabela 1 mostra a variação média (considerando a média de todos os números de clientes medidos no experimento) do percentual de vazão entre réplicas. Há uma grande redução na vazão quando a replicação é introduzida (mudança de 1 para 3 réplicas). Porém, nas mudanças subsequentes, a variação não é maior de 20%, conforme suposto na Hipótese 2. Portanto, pode-se considerar a Hipótese 2 verdadeira, pois foi possível experimentar um alto número de réplicas no sistema, e houve uma gradual redução na vazão durante o aumento do número de réplicas.

5. Trabalhos Relacionados

Tolerar faltas de parada em ambientes de rede local é um tema que possui bastantes estudos disponíveis na literatura. Entretanto, a aplicação de tolerância a faltas no contexto de *containers* ainda é incipiente. A seguir serão descritos alguns trabalhos recentes [Bolosky et al. 2011, Marandi et al. 2012, Moraru et al. 2013, Silva et al. 2013, Ongaro and Ousterhout 2014, Cui et al. 2015] e suas relações com a presente proposta.

Gaios [Bolosky et al. 2011] apresenta uma implementação de Paxos otimizada de forma alcançar um desempenho próximo do hardware. São toleradas faltas de parada e um

conjunto limitado de faltas Bizantinas. Gaios não persiste operações antes de executá-las e retornar o resultado ao cliente, com o objetivo de aumentar o desempenho. Porém, isso limita o número de faltas simultâneas que podem ocorrer pra que o estado não seja perdido ou corrompido. Além disso, a semântica resultante dessa otimização não é igual àquela usada pelos tradicionais bancos de dados. A presente proposta realiza a persistência das operações na memória compartilhada antes de executá-las e enviar a resposta ao cliente. É possível que uma operação proposta no Gaios nunca alcance quórum e não seja executada. No ONSET, todo pedido deve ser registrado na memória compartilhada por uma maioria de réplicas antes que a resposta seja enviada ao cliente. Desde que uma réplica consiga "contaminar" [Défago et al. 2004] a memória compartilhada com o pedido, este será executado em algum momento. A contaminação nesse caso é útil pois acelera o retorno de resultados para o cliente, caso o cliente perca a conexão com o sistema e precise re-enviar o pedido.

Multi-Ring Paxos [Marandi et al. 2012] é um protocolo projetado para ser escalável, aumentando a vazão do sistema à medida em que recursos são adicionados. O consenso é particionado em grupos, e cada grupo executa uma instância do algoritmo Ring Paxos[Marandi et al. 2010]. *Learners* que participam de mais de um grupo reúnem os resultados de diferentes grupos de maneira determinística. Por exemplo, um *learner* que participa dos grupos $g1$ e $g2$ reunirá as respostas usando uma estratégia *round-robin*: $r_1 1$, $r_1 2$, $r_2 1$, $r_2 2$, onde $r_i j$ é a resposta de ordem i advinda do grupo j. Para garantir o progresso de *learners* que participam em mais de um grupo, cada grupo controla o tempo que fica sem enviar mensagens. Após um tempo máximo, o grupo envia uma mensagem $SKIP$, informando que não possui novas instâncias de consenso, permitindo o progresso do sistema. O coordenador de cada anel monitora a taxa na qual mensagens são geradas, a fim de sugerir ajustes no monitoramento. A falha de um nó (*acceptor* ou *coordinator*) requer a formação de um novo anel. O Multi-Ring é um protocolo de *multicast* atômico, enquanto o ONSET utiliza uma única difusão de pedidos para ordenar pedidos e manter o estado replicado.

O EPaxos [Moraru et al. 2013] é basedo no Paxos, e tem por objetivos otimizar latência em rede amplamente distribuídas (WAN) e balancear carga entre réplicas. O consenso é alcançado com apenas 1 *round* de comunicação, quando pedidos concorrentes não interferem entre si. Neste caso, o protocolo funciona de maneira semelhante aos quóruns tradicionais [Malkhi and Reiter 1998]. Uma interferência entre dois pedidos A e B existe se a execução serial de A e B não for equivalente à execução serial de B e A. A aplicação é responsável por definir quais comandos interferem entre si. Quando pedidos concorrentes possuem interferência entre si, 2 *rounds* de comunicação são necessários. Como exemplo, considere que os pedidos A e B são enviados por diferentes clientes em um primeiro *round*. Um destes comandos será primeiramente executado por uma maioria de réplicas. Quando o comando B chegar a uma réplica X integrante da maioria de réplicas que executou A, esta réplica X irá informar a necessidade de executar A antes de ser possível a execução de B. O segundo *round* informará que $A \rightarrow B$, ou seja, a necessidade de executar A antes de B. Ocorre balanceamento de carga pois qualquer réplica pode atuar como líder de um pedido. Assim que possível, o líder notifica o cliente sobre a confirmação do pedido, e notifica as demais réplicas assincronamente. A latência é otimizada pois apenas um quórum de réplicas é suficiente para deliberar consenso. O EPaxos foi projetado para otimizar latência em WAN, mas o contexto de *containers* ainda é restrito a redes de área local (LAN).

RegPaxos [Silva et al. 2013] utiliza virtualização e compartilhamento distribuído de dados para criar uma memória compartilhada confiável. Dessa maneira, o número de ré-

plicas é reduzido de de $3f + 1$ para $2f + 1$ sem a utilização de componentes confiáveis complexos ou custosos em nível de software ou hardware. A presente proposta não tolera faltas bizantinas nem utiliza virtualização para criar a memória compartilhada, mas o ONSET é um protocolo que possui formato semelhante ao RegPaxos, exceto que no ONSET apenas uma réplica responde para o cliente. Isto foi assim definido porque o ONSET tolera apenas faltas de parada, dispensando a necessidade de enviar múltiplas respostas ao cliente.

Raft [Ongaro and Ousterhout 2014] é um protocolo de consenso desenvolvido para ser de fácil entendimento e, consequentemente, ser uma melhor base para o desenvolvimento de sistemas distribuídos. Os diferenciais do Raft são: i) o fluxo de entradas no *log* (registro de novos pedidos) segue apenas do líder para as demais réplicas; ii) o uso de temporizadores aleatórios na eleição de líder e nos mecanismos de *heartbeat* torna mais simples a resolução de conflitos; iii) uma nova abordagem permite com que o sistema continue operando normalmente mesmo durante mudanças de configuração. Raft possui desempenho e resultados equivalentes ao Paxos, porém devido ao objetivo de ser um protocolo de fácil compreensão, tornou-se amplamente utilizado. O ONSET também opta pela simplicidade ao utilizar uma memória compartilhada pré-existente no ambiente de comunicação entre as réplicas. O Raft mantém o sistema operante durante mudanças de configuração e requer um protocolo de mudança de visão. O ONSET também requer um protocolo de mudança de visão e continua operando durante mudanças de configuração, pois a memória compartilhada persiste os registros de chegada nas réplicas, independente de as réplicas permanecerem no sistema ou não. O registro das chegadas dos pedidos na memória compartilhada contém um número que indica o total de réplicas participantes, cuja atualização é suficiente para mudar o critério de maioria e assim alterar a condição da deliberação do consenso.

CRANE [Cui et al. 2015] é um protocolo de replicação de máquinas de estado que lida com a questão de não-determinismo provocado pela execução de pedidos em servidores com múltiplas *threads*. CRANE questiona a complexidade de usar interfaces de aplicações como ZooKeeper [Hunt et al. 2010] para coordenar a replicação. Assim, propõe tornar transparente a replicação, por meio de interceptação dos pedidos realizados em *sockets*, e da ordenação dos pedidos usando uma instância do Paxos. O não-determinismo causado pela execução em múltiplas *threads* é parcialmente resolvido por uma solução anterior dos mesmos autores, denominada PARROT. Um algoritmo chamado *time bubbling* orquestra a ordenação e a execução *multithread*, inserindo tempos de espera (”bolhas”) e preenchendo instantes de execução que estariam ociosos na ausência de requisições. Como as *bolhas* possuem tamanho fixo, o sincronismo global é garantido. CRANE usa um método de interceptação [Lung et al. 2000] para deixar a replicação de estados transparente, enquanto ONSET utiliza integração. Ambas abordagens tornam transparente a utilização de coordenação para a aplicação, mas a abordagem de integração registra desempenho melhor que a interceptação, conforme experiências realizadas no âmbito do padrão CORBA [Lung et al. 2000].

6. Conclusões

Este artigo apresentou uma arquitetura de integração de serviços de coordenação na plataforma Kubernetes, um sistema orquestrador de *containers*. Para operacionalizar a coordenação, foi desenvolvido o protocolo ONSET, que realiza ordenação com suporte da memória compartilhada já disponível no Kubernetes. Um protótipo foi desenvolvido na mesma linguagem de desenvolvimento do Kubernetes, para facilitar uma futura transição das ações de coordenação para um novo componente a ser integrado na arquitetura oficial do produto.

Experimentos demonstraram a viabilidade da proposta, a possibilidade em escalar réplicas em quantidades geralmente não consideradas ao utilizar máquinas (físicas ou virtuais) como réplicas, além da característica de gradual degradação de vazão durante o aumento no número de réplicas do sistema. O uso de *containers* pode provocar grande impacto nos *data centers*, e parte das aplicações que migrarem para essa plataforma poderão se beneficiar de serviços de coordenação oferecidos por protocolos como o ONSET. Empresas como a Google já utilizam *containers* há anos, e esse conhecimento está sendo disponibilizado agora também sob forma de *software* de código aberto.

Agradecimentos

Miguel Correia é bolsista CAPES/Brasil (projeto LEAD CLOUDS). O trabalho foi parcialmente financiado pela FCT UID/CEC/50021/2013 e pelo CNPq 455303/2014-2. Este trabalho foi também parcialmente financiado com recursos da FAPESC/IFC, processo/projeto Nº 00001905/2015, no qual Aldelir Luiz e Hylson Netto são integrantes.

Referências

Bessani, A. N., Lung, L. C., and da Silva Fraga, J. (2005). Extending the umiop specification for reliable multicast in corba. In *7th Int. Symposium on Distributed Objects and Applications*, Agia Napa, Cyprus.

Bolosky, W. J., Bradshaw, D., Haagens, R. B., Kusters, N. P., and Li, P. (2011). Paxos replicated state machines as the basis of a high-performance data store. In *USENIX NSDI*.

Cui, H., Gu, R., Liu, C., Chen, T., and Yang, J. (2015). Paxos made transparent. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 105–120. ACM.

Défago, X., Schiper, A., and Urbán, P. (2004). Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421.

Dwork, C., Lynch, N., and Stockmeyer, L. (1988). Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323.

Hunt, P., Konar, M., Junqueira, F. P., and Reed, B. (2010). Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference*, volume 8, page 9.

Lamport, L. (1998). The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169.

Lung, L. C., Fraga, J. S., Farines, J.-M., and Oliveira, J. R. S. (2000). Experiências com comunicação de grupo nas especificações fault tolerant corba. In *Simpósio Bras. de Redes de Computadores*, Belo H. - MG.

Malkhi, D. and Reiter, M. (1998). Byzantine quorum systems. *Distributed Computing*, 11(4):203–213.

Marandi, P. J., Primi, M., and Pedone, F. (2012). Multi-ring paxos. In *Dependable Systems and Networks (DSN), 42nd Annual IEEE/IFIP International Conference on*, pages 1–12.

Marandi, P. J., Primi, M., Schiper, N., and Pedone, F. (2010). Ring paxos: A high-throughput atomic broadcast protocol. In *International Conf. on Dependable Systems and Networks (DSN)*, pages 527–536. IEEE.

Mell, P. and Grance, T. (2011). The nist definition of cloud computing. Technical report, National Institute of Standards & Technology, Gaithersburg, MD, United States. SP 800-145.

Moraru, I., Andersen, D. G., and Kaminsky, M. (2013). There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth Symposium on Operating Systems Principles*, pages 358–372. ACM.

Ongaro, D. and Ousterhout, J. (2014). In search of an understandable consensus algorithm. In *Proceedings of USENIX Annual Technical Conference*, pages 305–320.

Schneider, F. B. (1990). Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319.

Silva, M. R. X., Lung, L. C., Luiz, A. F., and Magnabosco, L. Q. (2013). Tolerância a faltas bizantinas usando registradores compartilhados distribuídos. In *Simpósio Brasileiro de Redes de Computadores*, Brasília,DF.

Toffetti, G., Brunner, S., Blöchlinger, M., Dudouet, F., and Edmonds, A. (2015). An architecture for self-managing microservices. In *Workshop on Automated Incident Management in Cloud*, pages 19–24. ACM.

Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., and Wilkes, J. (2015). Large-scale cluster management at google with borg. In *European Conference on Computer Systems*, page 18. ACM.