Coordenação de *Containers* no Kubernetes: Uma Abordagem Baseada em Serviço

Hylson Vescovi Netto[1,2], Caio Pereira Oliveira[1], Aldelir Fernando Luiz[2], Lau Cheuk Lung[1], Luciana de Oliveira Rech[1], José Roque Betiol Júnior[1]

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

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

{hylson.vescovi,aldelir.luiz}@blumenau.ifc.edu.br,
{lau.lung,luciana.rech}@ufsc.br, {caiopoliveira,roque.betioljr}@gmail.com

Resumo. *Constantemente o paradigma de computação em nuvens tem sofrido modificações. Uma das principais consiste no uso da virtualização por contêineres, com vista para uma maior flexibilidade na infra-estrutura. A criação do gerenciador de contêineres denominado Kubernetes, por parte do Cloud Native Computing Foundation (CNCF), expressa os esforços para orientar tais modificações de maneira padronizada. O Kubernetes permite a replicação de contêineres, porém, não do estado das aplicações hospedadas. Neste sentido, este artigo versa sobre a replicação do estado de contêineres, em que é proposta uma arquitetura de sistema para efetuar a coordenação sob forma de serviço, a fim de acoplar-se à aplicação de maneira leve e simples. Como prova de conceito, foram realizados experimentos a fim de analisar o comportamento de aplicações com graus de consistência forte e eventual, cujos resultados demonstraram a viabilidade da proposta.*

Abstract. *The cloud computing paradigm have been modified. One of the most impacting changes is the usage of system level virtualization, for a better infrastructure management. The creation of Kubernetes, a containers management system, by the Cloud Native Computing Foundation (CNCF) express the efforts to guide the changes in a standard manner. Kubernetes can replicate containers, but not the state of the applications hosted in the containers. This paper presents an architecture for replicating state in containers providing coordination as a service, with a light and simple coupling to the application. Some experiments analyse the behavior of applications which observe eventual and strong consistency when reading data. The results shown that the proposal is feasible.*

1. Introdução

A profusão do paradigma de computação em nuvem (do inglês, *cloud computing* [Mell and Grance 2011]) no cotidiano das pessoas e organizações tem impelido quanto ao uso extensivo da tecnologia de virtualização [Popek and Goldberg 1974, Smith and Nair 2005], em face ao provisionamento dinâmico de recursos – um dos pilares fundamentais deste modelo de negócio –, o que portanto, vem de encontro à definição de nuvem computacional proposta pelo NIST [Mell and Grance 2011]. Um dos aspectos mais propulsores acerca da adoção da tecnologia de virtualização no âmbito das nuvens computacionais, decorre da permissibilidade de isolamento das cargas de trabalho executadas sobre tal ambiente, além da possibilidade da realização de algum controle sobre os recursos provisionados.

Embora a virtualização clássica (i.e., aquela onde o sistema de computação é virtualizado por completo) seja, indiscutivelmente, a tecnologia predominante no âmbito de provedores de nuvens computacionais (i.e., *data centers*), uma alternativa bastante atrativa na atualidade consiste na virtualização baseada em *containers* [Soltesz et al. 2007] – doravante referenciado por contêiner(es). A virtualização baseada em contêineres se caracterizada como uma tecnologia capaz de lançar instâncias de processamento isoladas umas das outras, isto é, numa mesma instância do sistema operacional, em que cada contêiner tem sua própria abstração de recursos. Deste modo, o isolamento não requer a execução das aplicações em separado, por diferentes instâncias de sistema operacional, tampouco o controle por algum monitor de máquinas virtuais, a exemplo do que ocorre com a virtualização clássica.

Dentre as tecnologias recentes para o suporte a virtualização por contêineres, uma das implementações mais populares na atualidade é o *Docker* [Bernstein 2014], a qual pode ser vista como uma extensão do LXC (i.e., *Linux Containers*) [Bernstein 2014]. Por outro lado, é digno de nota que a implementação nativa do *Docker* não provê mecanismos que possibilitam o gerenciamento/orquestração/integração dos contêineres num ambiente de *cluster* – um requisito desejável para o provimento de tolerância a faltas. Diante disso, alguns engenheiros do Borg [Verma et al. 2015] – o atual sistema de gerenciamento de contêineres do Google –, no âmbito do CNCF, trabalharam na proposição de um sistema denominado Kubernetes [Verma et al. 2015] – um sistema que visa o controle e gerenciamento do ciclo de vida de contêineres num ambiente de *cluster*.

Em suma, o Kubernetes replica os contêineres no intuito de melhorar a disponibilidade do ambiente virtualizado. Assim, aqueles contêineres que porventura apresentarem estado de falha serão recriados pelo Kubernetes, embora o estado da(s) aplicação(ões) em execução no(s) mesmo(s) não seja(m) recuperado(s). Porém, tal funcionalidade é passível de implementação a partir do uso de volumes externos. Todavia, é importante se ater ao fato de que o êxito na persistência do estado da aplicação só é obtido se os volumes puderem ter alguma proteção contra falhas, além de que a aplicação deverá exercer o controle sobre os acessos concorrentes ao volume em questão.

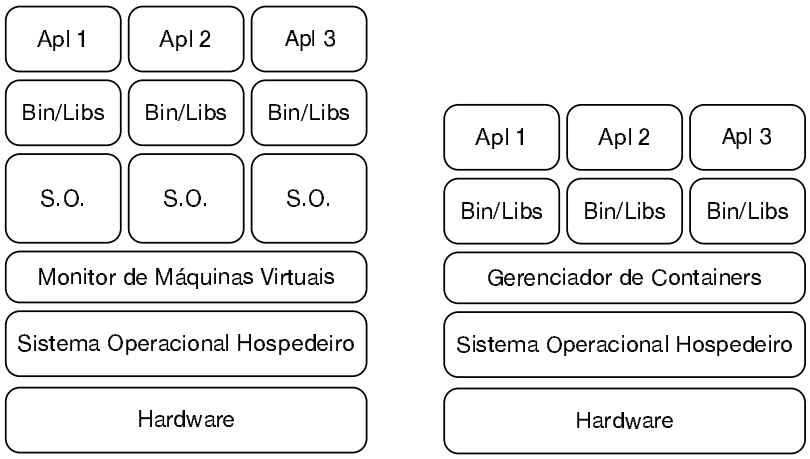
Diante do exposto, este trabalho visa a proposição de uma solução arquitetural para efetuar a coordenação da alteração de estados no Kubernetes, por meio de uma abordagem baseada em serviço. No que segue, o trabalho está organizado da seguinte maneira: a Seção 2 dedica-se a revisar a literatura acerca da tecnologia de virtualização baseada em contêineres; a Seção 3 descreve os detalhes de especificação empregados na proposta objeto deste trabalho, tais como arquitetura e algoritmos desenvolvidos; a Seção 4 apresenta os resultados experimentais obtidos a partir de um protótipo implementado; a Seção 5 estabelece relações com trabalhos da literatura, e por fim; na Seção 6 são evidenciadas as conclusões e alguns apontamentos de trabalhos futuros.

2. Tecnologia de Virtualização Baseada em Containers

A virtualização consiste numa tecnologia que permite abstrair os recursos físicos de um sistema de computação, tendo como propósito o provimento de um ambiente computacional virtual capaz de melhorar o aproveitamento dos recursos disponíveis [Parmelee et al. 1972, Popek and Goldberg 1974, Smith and Nair 2005]. De um modo geral, a virtualização ocorre por meio de uma máquina virtual, que é caracterizada como um ambiente de software construído sobre as interfaces fornecidas pelos recursos físicos disponíveis [Smith and Nair 2005]. Neste sentido, a virtualização em nível de sistema, doravante

denominada por virtualização baseada em contêineres, surgiu no contexto do sistema operacional FreeBSD, como uma versão estendida do comando *chroot*, denominada *jail* [Bernstein 2014]. Oportunamente, a SUN MicroSystems realizou modificações sobre o *jail*, incorporando-o ao Sistema Operacional Solaris, tendo denominado tal sistema por *zone*.

Em suma, a virtualização baseada em contêiner é similar às tecnologias de virtualização tradicionais (p. ex.: *VMWare*, *Xen*, etc.), no sentido de permitir que diversas aplicações – Apl n das Figuras 1(a) e 1(b) – sejam executadas de maneira isolada num mesmo sistema de computação (p. ex.: S.O. e hardware). Ao passo que a virtualização tradicional requer a execução de um sistema operacional completo (i.e., o convidado) num sistema hospedeiro para prover um ambiente isolado (cfm. a Figura 1(a)), um contêiner compartilha o núcleo subjacente do sistema hospedeiro e isola os processos executados nele, dos demais processos em execução no sistema hospedeiro (vide Figura 1(b)). Isto é, instâncias de contêineres virtualizam o próprio sistema operacional em vez de virtualizar o hardware, de modo que um gerenciador de contêiner faz o papel de mediador de uso e acesso aos recursos disponíveis no sistema de computação subjacente.



(a) Máquinas Virtuais (b) Containers

Figura 1. Modelos de arquitetura de tecnologias de virtualização.

A despeito das similaridades verificadas nas tecnologias de virtualização tradicional e baseada em contêineres, a segunda apresenta maior dinamicidade em termos de ciclo de vida dos processos/aplicações, uma vez que cabe a eles a simples tarefa de apenas iniciar ou destruir processos em seu espaço isolado. Outrossim, quando comparado às máquinas virtuais tradicionais os contêineres se destacam por sua eficiência, pois, não somente a sua instanciação, mas também o provisionamento dos recursos ocorre de maneira bastante rápida.

Não obstante, a virtualização por contêineres também é impulsionada pela portabilidade verificada em tal tecnologia, bem como pelo uso racional dos recursos – uma aplicação num contêiner só consome recursos quando é lançada uma instância do mesmo. Por outro lado, um contêiner é caracterizado como uma máquina virtual sem estado, uma vez que as imagens pelas quais instanciam-se os contêineres são estáticas. Assim, quando um contêiner é encerrado, não apenas seu estado, mas também o daquelas aplicações que nele estavam em execução são perdidos.

Embora os pontos ora elucidados apontem para vantagens quanto ao uso da tec-

nologia de contêineres, as implementações atuais de contêineres não dispõem de suporte para o gerenciamento de instâncias num ambiente de *cluster* (p. ex.: *docker*[1], *shifter*[2]), um requisito desejável se considerada a disponibilidade de aplicações executadas em contêineres. Neste sentido, no intuito de melhorar a disponibilidade de aplicações em contêineres o Google criou o Kubernetes [Bernstein 2014].

Mais precisamente, o Kubernetes consiste num ambiente gerenciador de contêineres, com a finalidade de administrar o ciclo de vida de contêineres em nós num ambiente de *cluster*. Dentre as funcionalidades providas pelo Kubernetes, pode-se citar como principais: controle de admissão dos contêineres, balanceamento de recursos, descoberta de serviços entre os contêineres, publicação do serviço para acessos externos ao *cluster* e o balanceamento de carga entre os contêineres.

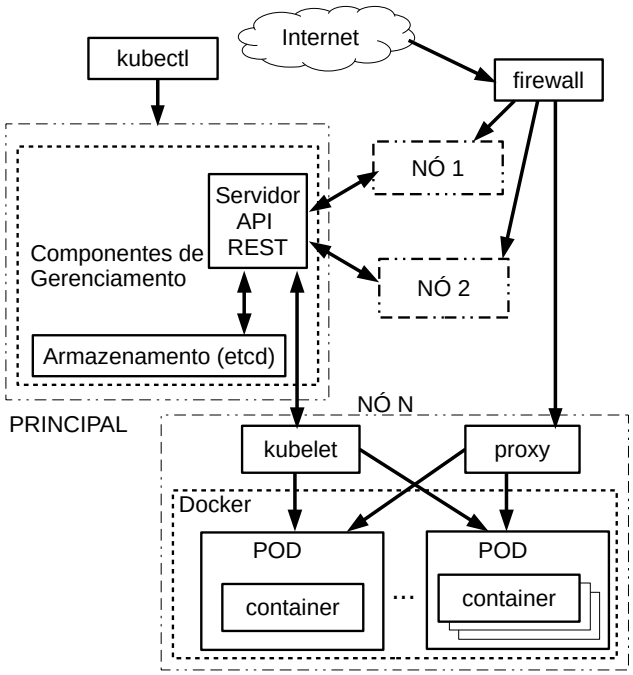


Figura 2. Arquitetura do Kubernetes

A Figura 2 descreve de maneira simplificada a arquitetura do Kubernetes. Note que o Kubernetes é composto por máquinas (virtuais ou físicas) denominadas nós. Os componentes POD consistem na unidade básica na qual o Kubernetes opera (i.e., uma aplicação, por exemplo), de modo que cada POD pode ter um ou mais contêineres. No contexto de um POD, os contêineres podem compartilhar recursos, por exemplo, um volume de dados externo. O *firewall* é responsável por despachar as requisições dos clientes para os nós presentes no *cluster* gerido pelo Kubernetes – cada requisição é entregue a apenas um nó.

O *proxy* tem a finalidade de encaminhar as requisições aos PODs e estes, por sua vez, podem ser replicados ou não. O Kubernetes pode replicar contêineres a fim de aumentar a disponibilidade de aplicações. Quando um contêiner falha, o Kubernetes recria o contêiner a partir de uma imagem. Assim, quando os PODs são replicados, as requisições são distribuídas pelo critério do algoritmo *Round Robin*. Por sua vez, os PODs são gerenciados pelo

[1]`https://github.com/docker/docker`
[2]`https://github.com/NERSC/shifter`

componente denominado *kubelet*, o qual também é utilizado para enviar dados relacionados ao monitoramento de contêineres ao nó principal.

E finalmente, o nó principal do Kubernetes mantém os componentes de gerenciamento, em que as informações acerca do *cluster* são persistidas num componente de armazenamento denominado *etcd* [Saito et al. 2016] – componente que implementa todo o armazenamento do Kubernetes. Neste nó, os componentes interagem uns com os outros por meio de APIs REST, onde os mesmos usam o servidor da API para salvar e recuperar informações. Por fim, a interação de um usuário com o ambiente de *cluster* é realizado pela interface de comando *kubectl*.

Conforme já fora mencionado, é importante salientar que, a despeito dos mecanismos providos pelo Kubernetes, o estado da aplicação hospedada no contêiner não é preservado/restaurado quando este é desativado. Neste sentido, este é o espaço de projeto que se busca explorar no âmbito deste trabalho.

3. Coordenação via Serviço no Kubernetes

Num ambiente de *cluster* onde se emprega a redundância para fins de melhoria da disponibilidade (i.e., tolerância a faltas) das aplicações, as requisições/pedidos que alteram o estado da(s) aplicação(ões) precisam ser devidamente coordenados antes de sua(s) respectiva(s) execução(ões). Uma abordagem bastante útil para realizar tal tarefa consiste na replicação de máquinas de estado (RME) [Schneider 1990]. Em suma, pelo uso da RME, pedidos concorrentes são submetidos a um algoritmo de consenso distribuído (p. ex.: Paxos [Lamport 1998] e Raft [Ongaro and Ousterhout 2014]) e a execução destes ocorre na mesma ordem em todas as réplicas.

É sabido que protocolos para RME disponíveis podem ser utilizados para realizar a coordenação dos pedidos de armazenamento. Entretanto, protocolos de coordenação, por mais simples ou eficientes que sejam, requerem esforço da aplicação para implementar sua integração. Cui e outros [Cui et al. 2015] descrevem a complexidade quanto ao uso de *interfaces* de aplicações como ZooKeeper [Hunt et al. 2010], para coordenar a replicação.

Uma possibilidade de utilização de um algoritmo de coordenação advém da incorporação desse algoritmo num ambiente existente. A literatura menciona que a incorporação de coordenação num ambiente pode ser feita pelo menos de três maneiras [Lung et al. 2000, Felber and Narasimhan 2004, Bessani et al. 2005]: integração, interceptação e serviço (também denominado *middleware*). No contexto de um ambiente, a abordagem de **integração** consiste em construir ou modificar um componente do ambiente, a fim de acrescentar uma funcionalidade.

De outro modo, a **interceptação** requer que as mensagens enviadas aos destinatários sejam capturadas e mapeadas num sistema de comunicação de grupo. É importante frisar que esse processo é feito de forma totalmente transparente à aplicação. Um exemplo prático consiste no uso de *interfaces* do sistema operacional [Narasimhan et al. 1997] para interceptar chamadas de sistema (i.e., *system calls*) relacionadas à aplicação.

E por fim, a abordagem de **serviço** consiste em definir uma camada de *software* entre o cliente e a aplicação, de modo que essa camada se torna responsável por prover a coordenação das requisições/pedidos enviada(o)s à aplicação.

3.1. Proposta de Arquitetura

Nesta seção se apresenta a proposta elaborada para realizar a coordenação da alteração de estados no Kubernetes, por meio da abordagem de serviço. Para tanto, foi desenvolvida uma arquitetura de sistema, doravante denominada CAIUS (**C**oorden**A**ção via Serv**I**ço no K**U**bernete**S**). Para facilitar a compreensão acerca da arquitetura proposta neste trabalho, a Figura 3(a) ilustra a execução coordenada de um pedido no âmbito do Kubernetes.

A operação normal do protocolo se dá da seguinte maneira. Inicialmente, o cliente envia o pedido (1), que por meio de um *firewall* (2) é entregue a um nó do *cluster* Kubernetes. O *proxy* do nó encaminha o pedido a uma réplica do contêiner de coordenação (3). O contêiner que recebeu o pedido realiza a ordenação do mesmo, na camada de coordenação via serviço (4). Após a ordem de execução ter sido estabelecida, o cliente é respondido com uma confirmação de que o pedido será em algum momento executado. Na sequência, por meio do *firewall* (5), a resposta é entregue ao cliente (6). E finalmente, o pedido entra em uma fila, e assim que for possível o mesmo é executado nas réplicas de aplicação (7).

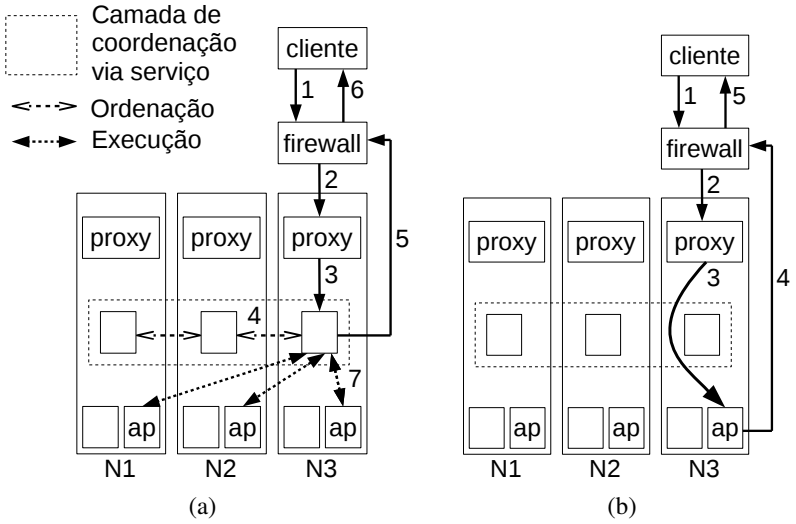


Figura 3. Kubernetes: incorporação de coordenação via serviço. (a) Atualizações coordenadas e (b) leituras diretamente na aplicação.

De outro modo, para os pedidos que efetuam apenas operações de leitura sobre estado da aplicação, é prevista uma otimização na execução do protocolo. Assim, os clientes que porventura vierem a executar operações de leitura sobre a aplicação, podem acessar diretamente as aplicações (Figura 3(b)), sem a necessidade de passar pelo serviço de coordenação. Neste caso, o cliente envia o pedido ao Kubernetes (1), que é entregue a um nó por meio do *firewall* (2). Na sequência, o *proxy* encaminha o pedido a uma das réplicas da aplicação (3), a aplicação executa o pedido (leitura de dados) e envia a resposta (4), que é entregue ao cliente (5) por meio do *firewall*.

Note pela Figura 3(a), que há uma separação das atividades/tarefas de ordenação e execução [Yin et al. 2003]. A separação destas tarefas em camadas distintas permite definir a consistência das operações, em função do número de réplicas da aplicação. Neste artigo, dois critérios de consistência são explorados para fins de análise, são eles: forte e eventual [Vogels 2009]. De maneira resumida, na consistência forte, depois que uma atualização de estado é concluída, qualquer acesso subsequente retornará o valor mais atual do estado. Um exemplo de aplicação que requer consistência forte é a edição colaborativa de documentos.

Sob outra perspectiva, a consistência eventual provê a garantia de que, após uma atualização do estado sem atualizações subsequentes, todos os leitores observarão o novo valor do estado. Assim, a janela de inconsistência pode ser determinada com base em fatores como latência de rede, carga de processamento e número de réplicas. Como exemplo, aplicações de rede social geralmente são satisfeitas com consistência eventual.

Na arquitetura proposta, pressupõe-se que a aplicação se encontra hospedada em contêineres. Diante disso, se houver apenas uma réplica da aplicação, toda operação de leitura retornará o último valor atualizado pelas operações de escrita coordenadas (Figura 3(b)) – esse cenário provê consistência forte à aplicação. Caso o número de réplicas da aplicação seja maior que um (1), que possivelmente é o caso, a fim de aumentar disponibilidade, o valor retornado pode ser o último valor completamente escrito (i.e., escrito em uma maioria de réplicas) ou um valor que esteja sendo escrito. Esse último cenário pode ocorrer em situações em que ocorram leituras e escritas simultâneas. Nesse caso, clientes da aplicação observam o estado sob uma consistência eventual.

3.2. Modelo de Sistema

Para a especificação da arquitetura, é pressuposto que o modelo de interação do ambiente é parcialmente síncrono [Dwork et al. 1988]. Para tanto, considera-se que clientes e réplicas são interconectados por uma rede em que os canais de comunicação são confiáveis (i.e., *reliable channels*), de modo que em algum momento, as mensagens são recebidas e entregues [Basu et al. 1996]. Em termos de faltas, são toleradas faltas por parada nas réplicas, em que réplicas podem parar de funcionar definitivamente, e assim deixar de realizar qualquer processamento ou comunicação.

Para fins de ordenação dos pedidos cuja finalidade é alterar o estado da aplicação, o CAIUS adota o Raft [Ongaro and Ousterhout 2014] como protocolo de consenso subjacente. Neste sentido, tão logo ocorre a ordenação de um pedido, o cliente é notificado pelo sistema por meio de um *ACK*. Os pedidos já ordenados entram numa fila de execução, cuja progressão se dá na medida em que uma maioria das réplicas da aplicação confirma a execução dos pedidos.

3.3. Controle de Fluxo de Execução

Para fins de compreensão, é importante salientar que o controle da execução dos pedidos nas réplicas de aplicação é feito pela réplica líder do Raft. No que segue, se apresenta uma explanação acerca do controle de fluxo realizado pela réplica líder. Note que a Figura 4 consiste num detalhamento da operação do líder, nos termos da Figura 3(a). Cada pedido recebido por um nó da camada de ordenação (passo 3) é ordenado pelo líder do protocolo subjacente de ordenação (Raft), em que este líder é que interage com as demais réplicas daquele protocolo (passo 4.1). Uma vez definida a ordem de um pedido, este é inserido numa fila de requisições (passo 4.2) e o cliente é notificado de que seu pedido será executado pelas réplicas da aplicação (passo 5). Para cada réplica de aplicação é criada uma *thread*, cuja finalidade destas é observar a fila de requisições (passo 7.1). Ao verificar um novo pedido na fila, a *thread* envia a respectiva requisição para a réplica de aplicação sob seu controle (passo 7.2). E finalmente, após a execução do pedido, a *thread* insere o resultado da execução no mapa de respostas (passo 7.3).

No que segue, são descritos alguns detalhes de operação do protocolo, aqui especificados pelos Algoritmos 1 e 2.

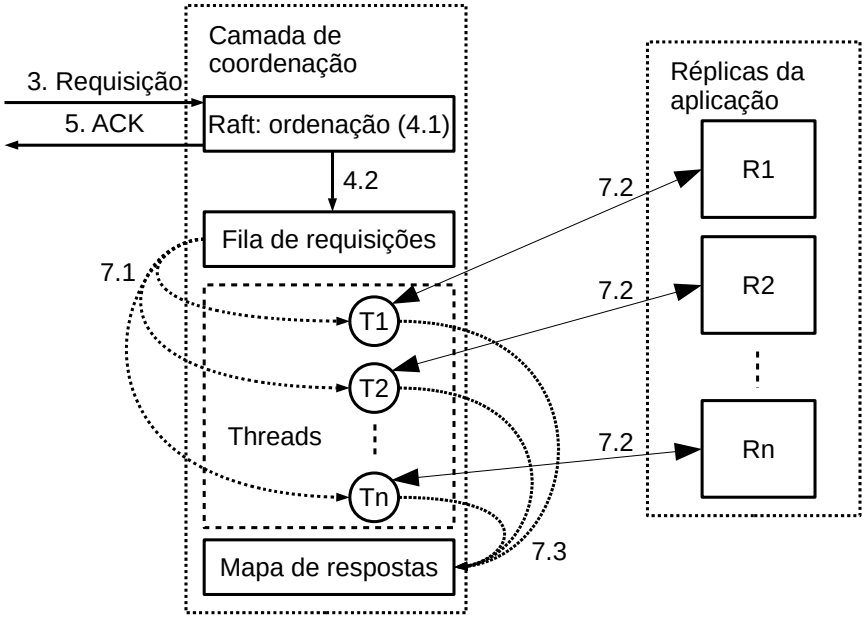


Figura 4. Estrutura interna do CAIUS

O serviço provido pela camada de coordenação é inicializado por meio da réplica líder do protocolo Raft, a partir da comunicação com as demais réplicas que fazem parte daquela instância do Raft (vide Algoritmo 1, linhas 1–10). A réplica líder do Raft contém uma fila para armazenar as requisições e um mapa para armazenar as respostas fornecidas às mesmas pelas réplicas da aplicação. Nesta réplica também são criadas as *threads* responsáveis por gerenciar a comunicação com cada nó da camada de aplicação (linhas 11-14 do Algoritmo 1). Os endereços das réplicas do Raft e da aplicação são obtidos pelas funções *onlineRaftReplicas* e *onlineAppReplicas*, respectivamente. Essas funções realizam chamadas à API do Kubernetes, em que é fornecida a *tag* dos contêineres para os quais deseja-se obter informações.

Algoritmo 1: Monitor de réplicas

```
1  answers[*,*] := {};
2  requestQueue := [];
3  lastAnswerdRequest := 0;
4  setOfRaftReplicas := {};
5  setOfApptReplicas := {};
6  while true do
7      for x | x in onlineRaftReplicas do
8          if x not in setOfRaftReplicas then
9              setOfRaftReplicas.append(x);
10             run raftReplica(x);
11     for x | x in onlineAppReplicas do
12         if x not in setOfAppReplicas then
13             setOfAppReplicas.append(x);
14             run appThread(x);
```

Note pelo Algoritmo 2, que durante a execução do protocolo as *threads* são mantidas num laço infinito, a fim de verificar a existência de novas requisições que chegam na

fila de requisições (Algoritmo 2, linha 4). Caso haja alguma nova requisição, esta é enviada pela *thread* para a respectiva réplica da aplicação sob o seu controle (linha 6), sendo que a resposta é adicionada no mapa que contém as respostas de cada requisição (linha 8). Assim, se a maioria das réplicas da aplicação responderam a requisição (linha 11), então essa requisição passa a demarcar, na fila de requisições, a última requisição respondida (linha 12).

Algoritmo 2: Controle de execução da aplicação

```
1  currentNode := x;
2  k := 0;
3  while true do
4      if len(requestQueue) > k then
5          k++;
6          request := requestQueue[k];
7          response := sendRequest(currentNode, request);
8          answers[k, currentNode] := response;
9          answerCount := len(answers[k]);
10         majority := floor(len(setOfReplicas) / 2) + 1;
11         if answerCount >= majority and lastAnswerdRequest < k then
12             lastAnswerdRequest := k;
```

4. Avaliação

No intuito de analisar o desempenho da arquitetura proposta, alguns experimentos foram realizados, cujos resultados são discutidos nesta seção. É importante salientar que a separação entre as camadas de ordenação e execução cria diferentes possibilidades de avaliação. Devido à semântica de execução das operações de atualização do estado no CAIUS (§ 3.2)[3] e da implementação adotada (§ 4.1), espera-se que o desempenho de acordo com o número de réplicas de ordenação ocorra segundo um padrão observado em trabalhos anteriores [Oliveira et al. 2016]. A execução de operações somente de escrita será realizada a fim de ratificar resultados já obtidos na literatura. Esse cenário de execução configura a especificação do primeiro experimento (e1).

Outra questão de interesse deste trabalho é aferir a diferença de desempenho ao variar o número de réplicas da aplicação, no contexto de operações de leitura do estado. O uso de uma ou mais réplicas de aplicação implica fornecimento de consistência forte ou eventual, para operações de leitura (§ 3). Um caso relevante, portanto, é o uso de apenas uma réplica de aplicação para verificar o comportamento da aplicação com consistência forte (e2). Outra possibilidade é o uso de três réplicas de aplicação, de forma a tolerar a falta, por parada, de uma das réplicas. O uso de um modelo de faltas por parada com ambiente assíncrono ($2f + 1$) permite maior disponibilidade (não impacta o desempenho durante situações de falta). Essa configuração (e3) provê à leitura do estado consistência eventual.

4.1. Ambiente e Implementação

O ambiente físico utilizado consistiu de um *cluster* composto por quatro microcomputadores, todos com as seguintes configurações: 1 (um) microprocessador Intel® Core™ i7

[3]O símbolo § significa referência a uma seção desse texto.

3.5GHz com 4 (quatro) núcleos e cache L3 8MB; 12GBytes de memória RAM, e; 1TB de armazenamento com 7200RPM. Para interconectar os elementos do *cluster* e cliente(s) foi utilizada uma rede de 10/100 Mbits, completamente isolada de tráfego externo. Como sistema operacional, adotou-se o Ubuntu 14.04.3 64 bits, *kernel* 3.19.0-42.

No ambiente do *cluster* utilizou-se o Kubernetes 1.1.7, de modo que uma máquina atuou como nó principal e as outras três máquinas atuaram como nós de execução de contêineres. Como implementação de contêineres foi utilizado o Docker. O cliente foi executado no nó principal. É digno de nota que o CAIUS foi implementado a partir da linguagem Go, versão 1.4.2. Como protocolo subjacente de ordenação, adotou-se o Raft. Nos experimentos, a aplicação avaliada consistiu num repositório de texto, denominada *logger*. A imagem dos contêineres do CAIUS e do *logger* estão disponíveis no Docker Hub[4], sob as *tags raft* e *logger*. Os respectivos códigos-fonte e os resultados dos experimentos (datados de 19/12/2016) estão disponíveis no GitHub[5].

No arquivo de configuração YAML dos contêineres do CAIUS para o Kubernetes, foi definida uma variável de ambiente que corresponde à *tag* dos contêineres da aplicação a ser coordenada. Nestes experimentos, seu valor foi especificado como "logger".

4.2. Experimentos

Em relação aos experimentos, a obtenção das amostras ocorreu a partir de três cenários de experimentação, os quais foram, nomeadamente: *i*) somente escritores, *ii*) leitores observando consistência forte, e, *iii*) leitores sujeitos a consistência eventual. Nos três experimentos houve atuação dos escritores. O primeiro experimento (e1) contou com três réplicas da aplicação. No segundo experimento (e2) foi criada apenas uma instância da aplicação, para garantir consistência forte aos clientes que efetivaram leitura diretamente da aplicação (sem passar pela coordenação). Por fim, no terceiro experimento (e3) foram criadas três réplicas da aplicação. Os clientes realizaram a leitura diretamente em apenas uma réplica da aplicação. Essa réplica foi escolhida pelo *proxy* do Kubernetes, a cada pedido, segundo o escalonamento padrão *Round-Robin*.

Não foi utilizado um *firewall* em nível de *cluster*. Todas as escritas e leituras foram direcionadas ao primeiro nó do *cluster*. No segundo experimento, a aplicação foi instanciada (pelo escalonador do Kubernetes) no primeiro nó. Em todos os experimentos o líder do Raft foi criado no segundo nó. A operação de escrita foi caracterizada por 8000 requisições de escrita, realizadas simultaneamente por 16 clientes. Operações de leitura (em e2 e e3) consistiram de 80.000 requisições de leitura, realizadas por 256 clientes simultâneos. As requisições de escrita e leitura foram executadas com uso da aplicação *ab*[6]. Tais requisições eram do tamanho de 100 bytes, enquanto que a(s) resposta(s) eram de 5 bytes. Os clientes enviavam novas requisições apenas após o recebimento da resposta da requisição anterior.

Durante os experimentos, os recursos consumidos foram monitorados com a ferramenta *dstat*[7]. Cada experimento foi monitorado pelo período de 35 segundos, tendo sido o monitoramento iniciado instantes antes do início da execução daquele experimento.

[4] `https://hub.docker.com/r/caiopo`

[5] `https://github.com/caiopo`, repositórios *raft* e *pontoon*.

[6] *Apache Benchmark*, em `http://httpd.apache.org/docs/2.4/programs/ab.html`.

[7] `dag.wiee.rs/home-made/dstat/`

4.3. Resultados e Discussões

A atuação de múltiplos clientes resultou em latências semelhantes percebidas por cada cliente escritor, em todos os experimentos (Tabela 1, segunda coluna). Esses valores estão nos limites conhecidos por experiências anteriores com o uso da implementação considerada neste artigo (42ms para 16 clientes, em [Oliveira et al. 2016]).

Tabela 1. Resultados dos experimentos.

	escritores (16)			leitores (256)		
	latência (ms)			latência (ms)		
experimento	média	desv.pad.	vazão (req/s)	média	desv.pad.	vazão (req/s)
e1	39.2	1.85	407	-	-	-
e2	42.3	28.54	373	4.95	19.17	50969
e3	42.58	21.86	375	7.64	29.68	33537

A latência percebida foi menor para clientes acessando a aplicação sob consistência forte. Apesar de os desvios-padrão serem altos, é digno que nota que um teste estatístico de hipótese (*t* de *Student*, com intervalo de confiança e 99%) mostra que a relação mencionada é verdadeira. Porém, essa vantagem ocorreu devido ao fato de que todas as leituras foram realizadas no mesmo nó onde a aplicação foi instanciada.

Clientes leitores que atuam sob a consistência eventual têm suas requisições sujeitas à ação do balanceamento de carga promovido pelo *proxy* em nível de nó. Devido a isso, percebem maior latência e menor vazão.

É importante salientar que os experimentos visam, sobretudo, demonstrar o consumo de recursos observado acerca da coordenação efetuada sobre os contêineres. Diante disso, no primeiro experimento (Figura 5), observa-se o alto consumo de recursos (rede e CPU) no segundo nó. Isso se deve ao fato do líder do Raft estar instanciado nesse nó. Além disso, corrobora o fato de o CAIUS também estar sendo executado junto ao líder do Raft.

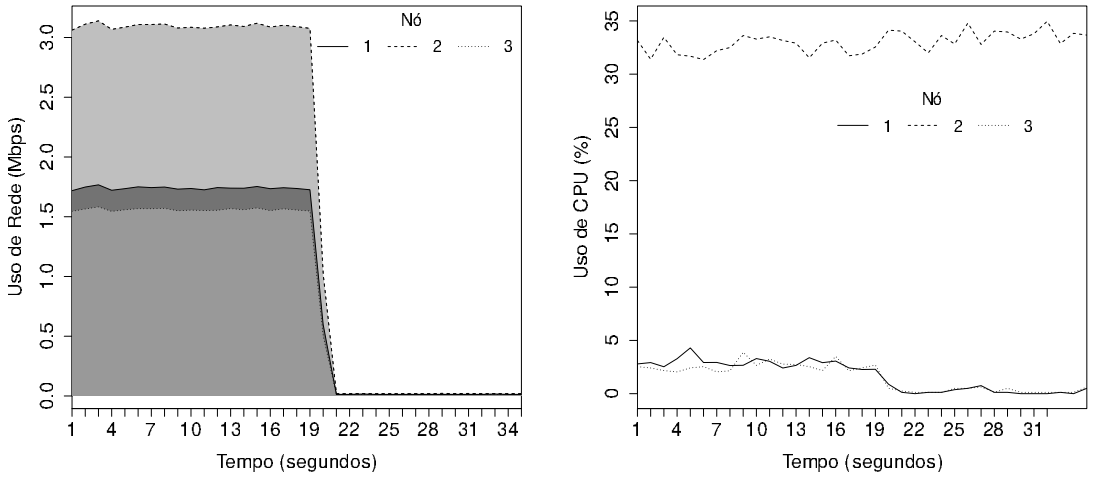
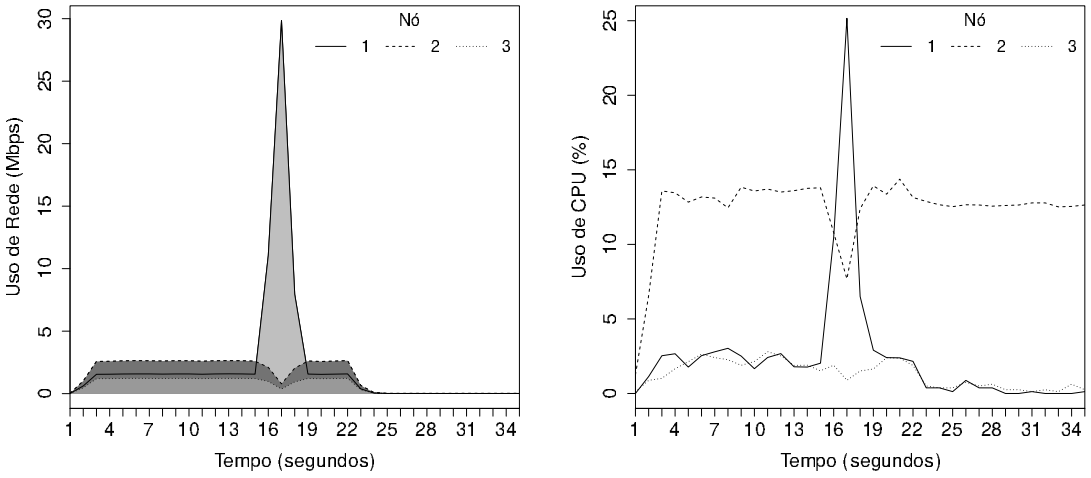


Figura 5. Experimento 1: consumo de (a) rede e (b) CPU.

De outro modo, no segundo experimento (Figura 6), o consumo de recursos é evidenciado na atuação dos clientes leitores. Entre os instantes 14s e 19s os clientes realizam a

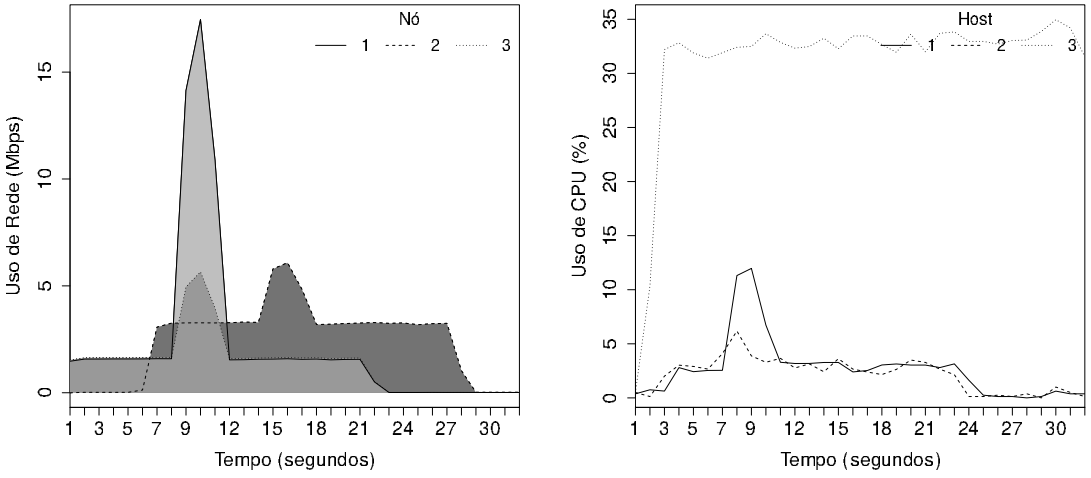
leitura de dados diretamente na réplica da aplicação. Esta é a razão pela qual ocorre o pico de consumo no primeiro nó.



(a) (b)

Figura 6. Experimento 2: consumo de (a) rede e (b) CPU.

E finalmente, o terceiro experimento (Figura 7) demonstra que o consumo de rede também manifesta um ponto de destaque no instante em que há a atuação de clientes leitores. Entretanto, ao que se percebe, o primeiro nó forneceu muito mais respostas do que os demais nós. Conforme mencionado sobre o aspecto de latência, o consumo predominou no primeiro nó devido ao fato de todos os pedidos de leitura terem sido realizados nesse nó.



(a) (b)

Figura 7. Experimento 3: consumo de (a) rede e (b) CPU.

5. Trabalhos Relacionados

A literatura dispõe de várias ferramentas que visam auxiliar na implementação de RME. Por exemplo, o BFT-SMaRt [Bessani et al. 2014] é disponível e implementa funcionalidades como transferência de estado e reconfiguração. O BFT-SMaRt já fora avaliado no cenário de contêineres [Torresini et al. 2016], onde aferiu-se novamente (corroborando [Felber and Narasimhan 2004]) a superioridade de desempenho dos contêineres em relação às tradicionais máquinas virtuais.

Em se tratando da atividade de coordenação, a literatura descreve que a incorporação de tal funcionalidade em ambientes pode ser feita por meio de integração, interceptação ou serviço [Lung et al. 2000]. O CRANE [Cui et al. 2015] torna transparente a coordenação realizada com o Paxos [Lamport 1998], por meio de interceptação, em nível de *sockets*, dos pedidos. No CAIUS a coordenação é acoplada ao Kubernetes por meio de um serviço, de modo que ela permanece transparente sob o ponto de vista da aplicação.

No contexto do Kubernetes, a RME já foi integrada ao mesmo, com vistas a prover a coordenação transparente para o usuário e reduzir o tamanho dos contêineres da aplicação [Netto et al. 2016]. Especificamente, a atuação do Raft no Kubernetes também já foi alvo de [Oliveira et al. 2016], onde se observou uma diferença de aproximadamente 17.4% de desempenho ao executar o Raft em ambiente virtualizado pelo Kubernetes (com uso do Docker) em comparação ao mesmo sendo executado diretamente numa máquina física. Porém, é digno de nota que a sobrecarga imposta pela virtualização é compensada pelas vantagens do gerenciamento que o Kubernetes proporciona ao ambiente.

6. Conclusões

A virtualização em nível de sistema (contêineres) é uma tendência nos provedores de nuvem. De maneira similar, ocorre o desenvolvimento de um ambiente para a gerência de contêineres (o Kubernetes) com vistas à adoção do mesmo por um grupo de provedores em nuvem. Dentre as aplicações que rodam em nuvem, algumas delas requerem sincronismo de dados, quando a aplicação mantém o estado da aplicação replicado. Este trabalho apresentou o CAIUS, uma arquitetura que provê coordenação a aplicações que necessitam ter o estado replicado. O CAIUS foi avaliado com aplicações que observaram as semânticas forte e eventual, duas semânticas bastante praticadas em aplicações da Internet.

O CAIUS é flexível ao ponto de poder ser modificado para utilizar outros algoritmos de consenso/ordenação, além do Raft. Por exemplo, o EPaxos [Moraru et al. 2013] não utiliza líder e pode ser uma alternativa que faz melhor uso do balanceamento provido pelo Kubernetes. Nesse caso, o CAIUS seria ativo em todas as réplicas da camada de coordenação, e não apenas no líder. Outra possibilidade de investigação futura é verificar os impactos da variação do número de réplicas que atuam na coordenação e o número de réplicas da aplicação. Adicionalmente, a realização de mais experimentos considerando o balanceamento de carga em nível de *cluster* (provido pelo *firewall*) torna-se fundamental para instanciar um cenário mais próximo da realidade em provedores, nos quais as escritas e leituras de dados podem ser realizadas em quaisquer réplicas da camada de coordenação e de aplicação, respectivamente.

Agradecimentos

Este trabalho foi parcialmente financiado pela FAPESC/IFC, projeto nº 00001905/2015. Infelizmente, durante a elaboração deste trabalho o quarto autor, nosso orientador, colega e amigo Lau Cheuk Lung faleceu devido a súbitas complicações de saúde. Aproveitamos essa oportunidade para prestar uma pequena homenagem a um grande pesquisador e professor, que influenciou muitos alunos e colegas durante sua carreira.

Referências

Basu, A., Charron-Bost, B., and Toueg, S. (1996). Simulating reliable links with unreliable links in the presence of process crashes. In *Proceedings of the 10th International Workshop on Distributed Algorithms*, pages 105–122.

Bernstein, D. (2014). Containers and cloud: From LXC to docker to Kubernetes. *IEEE Cloud Computing*, 1(3):81–84.

Bessani, A., Sousa, J., and Alchieri, E. E. (2014). State machine replication for the masses with bft-smart. In *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 355–362.

Bessani, A. N., Lung, L. C., and da Silva Fraga, J. (2005). Extending the UMIOP specification for reliable multicast in CORBA. In *Proceedings of the nternational Symposium on Distributed Objects and Applications*, pages 662–679.

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.

Dwork, C., Lynch, N. A., and Stockmeyer, L. (1988). Consensus in the presence of partial synchrony. *Journal of ACM*, 35(2):288–322.

Felber, P. and Narasimhan, P. (2004). Experiences, strategies, and challenges in building fault-tolerant CORBA systems. *Transactions on Computers*, 53(5):497–511.

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

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

Lung, L. C., da Silva Fraga, J., Farines, J.-M., and de Oliveira, J. R. S. (2000). Experiências com comunicação de grupo nas especificações fault tolerant CORBA. In *Simpósio Brasileiro de Redes de Computadores*.

Mell, P. and Grance, T. (2011). The NIST definition of cloud computing. Technical report, National Institute of Standards and Technology Gaithersburg.

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.

Narasimhan, P., Moser, L. E., and Melliar-Smith, P. M. (1997). Exploiting the internet inter-ORB protocol interface to provide CORBA with fault tolerance. In *Proceedings of the 3rd USENIX Conference on Object-Oriented Technologies and Systems*, pages 6–15.

Netto, H. V., Lung, L. C., Correia, M., and Luiz, A. F. (2016). Replicação de máquinas de estado em *containers* no kubernetes: uma proposta de integração. In *Anais do Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuidos*.

Oliveira, C., Lung, L. C., Netto, H., and Rech, L. (2016). Evaluating raft in docker on kubernetes. In *Proceedings of the International Conference on Systems Science*, pages 123–130.

Ongaro, D. and Ousterhout, J. (2014). In search of an understandable consensus algorithm. In *Proceedings of the USENIX Annual Technical Conference*, pages 305–319.

Parmelee, R. P., Peterson, T. I., Tillman, C. C., and Hatfield, D. J. (1972). Virtual storage and virtual machine concepts. *IBM Systems Journal*, 11(2):99–130.

Popek, G. J. and Goldberg, R. P. (1974). Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421.

Saito, H., Lee, H.-C. C., and Hsu, K.-J. C. (2016). *Kubernetes Cookbook*. Packt Publishing, Aachen, DE.

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

Smith, J. E. and Nair, R. (2005). The architecture of virtual machines. *Computer*, 38(5):32–38.

Soltesz, S., Pötzl, H., Fiuczynski, M. E., Bavier, A., and Peterson, L. (2007). Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In *Proceedings of the 2nd ACM SIGOPS European Conference on Computer Systems*, pages 275–287.

Torresini, E., Pacheco, L. A., Alchieri, E. A. P., and Caetano, M. F. (2016). Aspectos práticos da virtualização de replicação de máquina de estado. In *Workshop on Cloud Networks & Cloudscape Brazil, Congresso da Sociedade Brasileira de Computação*.

Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., and Wilkes, J. (2015). Large-scale cluster management at Google with Borg. In *Proceedings of the 10th European Conference on Computer Systems*, pages 18:1–18:17.

Vogels, W. (2009). Eventually consistent. *Communications of the ACM*, 52(1):40–44.

Yin, J., Martin, J.-P., Venkataramani, A., Alvisi, L., and Dahlin, M. (2003). Separating agreement from execution for Byzantine fault tolerant services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 253–267.