Integração de Replicação Máquina de Estados no Kubernetes

Felipe Albuquerque, Eduardo Alchieri, Marcos Caetano e Priscila Solis

[1] Departamento de Ciência da Computação
Universidade de Brasília – UnB

Abstract. *State Machine Replication (SMR) is an approach widely used to implement fault-tolerant systems. In this approach, servers are replicated and client requests are deterministically executed in the same order by all replicas. In order to provide support to the development of fault-tolerant virtualized applications, this work proposes an architecture to integrate SMR (through the* BFT-SMART *library) with Kubernetes, a container orchestrator. The proposed solution provides high transparency to developers and final users. Some experiments conducted with a hashmap application developed over the proposed architecture show its practical feasibility.*

Resumo. *A Replicação Máquina de Estados (RME) é uma abordagem muito utilizada na implementação de sistemas tolerantes a falhas. Esta técnica consiste em replicar os servidores e fazer com que os mesmos executem deterministicamente, e na mesma ordem, o mesmo conjunto de requisições. Visando prover tolerância a falhas em ambientes virtualizados, este trabalho propõe uma arquitetura para integrar protocolos de RME através da biblioteca* BFT-SMART *ao Kubernetes, que é um orquestrador de contêineres. A solução proposta busca fornecer um elevado grau de transparência tanto para os desenvolvedores de aplicações virtualizadas quanto para os usuários finais. Alguns experimentos realizados com uma aplicação de hashmap desenvolvida sobre a arquitetura proposta mostram a sua viabilidade prática.*

1. Introdução

A Replicação Máquina de Estados (RME) [Schneider 1990] é uma abordagem muito utilizada na implementação de sistemas tolerantes a falhas [Lamport 1998, Schneider 1990, Castro and Liskov 2002]. Basicamente, esta técnica consiste em replicar os servidores e fazer com que os mesmos executem deterministicamente, e na mesma ordem, o mesmo conjunto de operações requisitadas por clientes, fornecendo um serviço de replicação com consistência forte (*linearizability*) [Herlihy and Wing 1990]. Para manter o determinismo da execução, as operações são ordenadas e executadas sequencialmente seguindo a mesma ordem em todas as réplicas.

A virtualização surgiu ainda nas décadas de 60 e 70, através da IBM que desenvolveu sistemas operacionais com suporte a técnica de virtualização [Goldberg 1973, Goldberg and Mager 1979]. Depois disso, o *hardware* ficou mais barato e a virtualização perdeu importância na década de 80, voltando a ganhar atenção a partir da década de 90 com o surgimento da linguagem de programação Java [Laureano and Maziero 2008]. Virtualização pode ser considerada uma técnica que favorece o desenvolvimento de aplicações tolerantes a falhas, visto que cada máquina virtual (ou contêiner) fica isolada, e vem sendo usada em trabalhos nesta área [Oliveira et al. 2016, Garfinkel and Rosenblum 2003,

Laureano et al. 2004, Jiang and Wang 2007, Dettoni et al. 2013, Netto et al. 2016]. Além de prover tolerância a falhas, estes trabalhos visam melhorar o aproveitamento dos recursos computacionais disponíveis para uma aplicação, tanto através do uso de máquinas virtuais [Dettoni et al. 2013] quanto através do uso de contêineres [Netto et al. 2016].

No entanto, estes trabalhos existentes apenas integram protocolos de consenso (ex.: Raft [Howard et al. 2015]) no ambiente virtualizado, sem considerar outros aspectos da RME como por exemplo recuperação e transferência de estado. Além disso, estas arquiteturas propostas não deixam os aspectos da replicação transparentes no sistema. Visando avançar em direção a uma solução abrangente de RME para ambientes virtualizados, este trabalho propõe uma arquitetura para integração do BFT-SMART no orquestrador de contêineres Kubernetes. O BFT-SMART [Bessani et al. 2014] é uma biblioteca completa para desenvolvimento de aplicações tolerantes a falhas através de RME, fornecendo suporte para a recuperação, a reconfiguração, a transferência de estados, dentre outros. Já o Kubernetes [Bernstein 2014] é um orquestrador de contêineres (ex.: *Docker* [Docker 2018]) de código aberto desenvolvido pela Google, fornecendo todas as funcionalidades necessárias para criar e gerenciar contêineres. A arquitetura proposta neste trabalho visa que esta integração seja transparente tanto para usuários quanto para desenvolvedores de aplicações, i.e., as aplicações são desenvolvidas para execução em um contêiner da maneira tradicional como se o mesmo não estivesse fazendo parte de um sistema replicado, e ao mesmo tempo toda a replicação fica escondida dos usuários finais que acessam o sistema como na forma tradicional.

O restante deste artigo está organizado da seguinte forma. A Seção 2 apresenta a fundamentação teórica para o trabalho, abordando os conceitos de máquinas virtuais e contêineres, além de explicar o orquestrador Kubernetes e discutir os conceitos envolvendo RME e a biblioteca BFT-SMART. A Seção 3 apresenta a arquitetura proposta para integração do BFT-SMART no Kubernetes. A Seção 4 apresenta e analisa alguns experimentos realizados. As conclusões do trabalho são apresentadas na Seção 5.

2. Fundamentação Teórica

Esta seção apresenta a fundamentação teórica do trabalho. Primeiramente, alguns conceitos envolvendo máquinas e contêineres são apresentados, bem como o orquestrador de contêineres Kubernetes. Por fim, os conceitos envolvendo Replicação Máquina de Estados (RME) são discutidos e a biblioteca BFT-SMART é apresentada.

2.1. Máquinas Virtuais

Uma máquina física real é formada por vários componentes físicos que fornecem uma interface para os sistemas operacionais e suas aplicações. Já uma máquina virtual nada mais é do que uma camada de virtualização (*software*) construída sobre esta interface e que fornece uma nova interface para os demais componentes do sistema. Uma máquina virtual também pode ser definida como uma cópia isolada e protegida de uma máquina real [Goldberg and Mager 1979].

Uma máquina virtual é constituída de três partes [Laureano and Maziero 2008]:

- O sistema hospedeiro, que engloba os recursos reais de *hardware* e *software*.
- A camada de virtualização, ou o sistema supervisor/monitor, que fornece as interfaces virtuais a partir da interface real.

- E o sistema convidado, que é executado sobre o sistema virtualizado.

Os sistemas supervisores comumente apresentam as seguintes propriedades que podem ser usadas pelas aplicações para questões de segurança e também para tolerância a falhas [Laureano and Maziero 2008]: equivalência – o ambiente provido pelo supervisor é quase idêntico ao da máquina real; controle de recursos – o supervisor faz o controle completo dos recursos da máquina real; eficiência – grande parte das instruções do processador virtual devem executar diretamente no processador real, sem a interferência do supervisor; isolamento – aquilo que é executado em uma máquina virtual fica isolado do resto do sistema; inspeção – o supervisor tem acesso e controle sobre o estado interno da máquina virtual; gerenciabilidade – o supervisor gerencia os recursos utilizados pelos sistemas convidados; encapsulamento – o supervisor pode fazer *checkpoints* das aplicações/sistemas convidadas a fim de permitir recuperação, migração e inspeção em caso de falhas; e recursividade – é possível executar um supervisor dentro de uma máquina virtual, produzindo uma nova camada de virtualização.

2.2. Contêineres

Contêiner é o nome dado a uma ambiente isolado, gerenciável e com arquivos em camadas que pode ser criado sobre um sistema operacional. A criação de contêineres é muito rápida e o provisionamento de recursos é mais eficiente quando comparado com máquinas virtuais. Neste trabalho é utilizado o sistema de virtualização baseado em contêiner conhecido como $Docker$ [Docker 2018].

Existem diferenças fundamentais na arquitetura de um sistema de virtualização completo e de um sistema de virtualização baseado em contêineres. Na arquitetura de virtualização completa, para que cada aplicação seja executada de forma isolada, faz-se necessário a instalação de um sistema operacional completo. Desta forma, os recursos serão desperdiçados quando comparados com abordagem de uso de contêineres. Nesta abordagem, o sistema gerenciador de contêiner é o mediador no uso e acesso aos recursos computacionais, não havendo a necessidade da instalação de um sistema operacional completo para que cada aplicação possa ser executada de forma isolada [Merkel 2014].

2.3. Kubernetes

Com a demanda crescente por soluções de conteinerização, foi necessária a criação de softwares que facilitam o gerenciamento de contêineres. Esses softwares são denominados orquestradores de contêineres e têm como objetivo automatizar tarefas de implantação, atualização, controle de acesso e proporcionar algumas funcionalidades como escalabilidade, tolerância a falhas e balanceamento de carga.

O Kubernetes [Bernstein 2014] é um orquestrador de contêineres criado a partir da experiência obtida pela Google em outros sistemas, como o Borg [Burns et al. 2016] e o Omega [Schwarzkopf et al. 2013]. O Kubernetes é um software que gerencia contêineres distribuídos sobre um *cluster* de máquinas heterogêneas, que são divididas entre mestre e nós (Figura 1). O mestre é a máquina que disponibiliza os serviços do *cluster* através de uma API e é responsável por coordenar suas atividades, executando por exemplo tarefas relacionadas como o escalonamento e atualizações. Por outro lado, os nós são máquinas que funcionam como os trabalhadores do *cluster* e executam os contêineres que provêm seus serviços. Quando um novo serviço é criado no *cluster*, o mestre espalha contêineres

pelos nós, em unidades denominadas *Pods*, levando em consideração fatores como os requisitos de *hardware* da aplicação e a carga de cada nó.

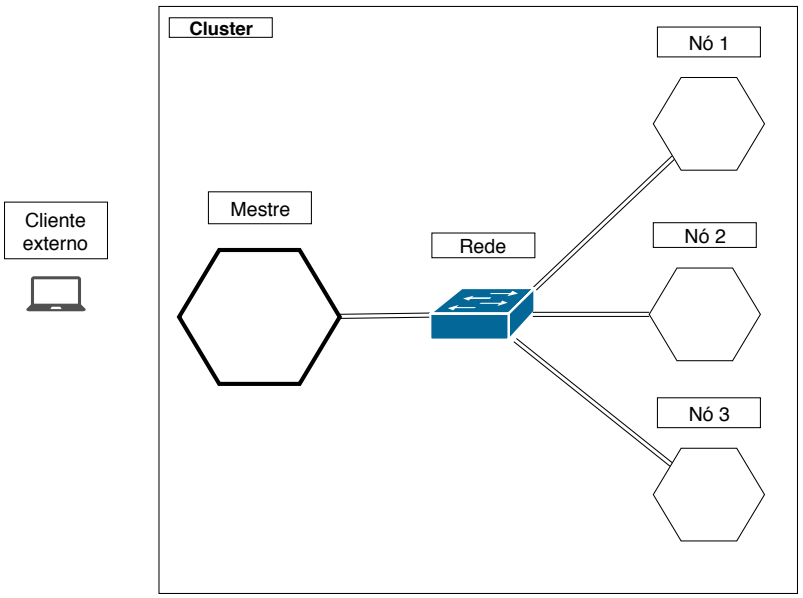


Figura 1. Representação de um *cluster* com seu mestre e seus nós.

***PODs*.** *Pods* geralmente são formados por um conjunto de contêineres altamente acoplados. Assim, podem ser vistos como recipientes que comportam grupos de contêineres que devem trabalhar em conjunto para entregar um serviço. Por exemplo, em um servidor Web que necessita de um banco de dados, um *Pod* poderia conter dois contêineres (um para o servidor Web e outro para o servidor de banco de dados).

Os contêineres presentes em um Pod são vistos externamente como uma entidade única. Eles dividem o mesmo domínio de rede, possuindo um mesmo endereço IP e compartilhando todo o conjunto de portas, como se fossem um único contêiner ou máquina. Isso permite que contêineres de um mesmo *Pod* acessem uns aos outros através da interface de *loopback* (*localhost*) e ao mesmo tempo proíbe que portas iguais sejam utilizadas em mais de um *Pod*. Além disso, contêineres de um mesmo Pod têm acesso à volumes compartilhados associados ao Pod e processos presentes neles podem se comunicar através de mecanismos de comunicação entre processos.

***Deployments*.** Através de *deployments* é possível criar, gerenciar e monitorar conjuntos de *Pods* idênticos para prover um mesmo serviço. Dessa forma, serviços se tornam elásticos (podendo aumentar ou diminuir de acordo com a demanda), atualizações podem ser aplicadas em grande escala, a carga do serviço pode ser distribuída ao longo de múltiplos *Pods*, etc. No momento da criação de um *deployment*, são definidos alguns parâmetros de implantação, os quais incluem: número de *Pods* que devem ser criados; número mínimo de *Pods* que devem estar disponíveis em um dado momento (em execução); quantidade de *Pods* que podem ser instanciados além do número máximo definido (utilizado em casos especiais, como durante atualizações); e quantas versões devem ser mantidas para permitir *rollback*.

Serviços. Os *Pods* criados em um *cluster* Kubernetes só podem ser acessados de dentro do próprio *cluster*. Para tornar uma aplicação visível externamente, são utilizados serviços. Apesar de parecer sem propósito tornar os *Pods* visíveis somente dentro do *cluster*, isso permite que modificações ocorram de forma transparente ao usuário. Por exemplo, a quantidade de *Pods* pode ser aumentada ou diminuída, pode ser feito balanceamento de carga ou ainda atualizações podem ser executadas sem que o usuário perceba qualquer interferência no serviço prestado.

Serviços podem ser expostos de quatro formas diferentes: ClusterIP – quando *Pods* são expostos dessa forma, eles recebem um IP interno do *cluster* e ficam visíveis somente dentro dele; NodePort – o serviço é exposto através de uma porta disponível no mestre e nos nós do *cluster* (nesse caso, o serviço é acessado através do IP do nó e de sua porta); LoadBalancer – o serviço é exposto através de um balanceador de carga externo provido por serviços em nuvem; e ExternalName – mapeia o serviço através de um nome, como `servico.cluster.com`.

2.4. Replicação Máquina de Estados

A Replicação Máquina de Estados (RME) consiste em replicar os servidores e coordenar as interações entre os clientes e estas réplicas, com o objetivo de que as mesmas apresentem a mesma evolução em seus estados. O modelo básico de programação de uma RME envolve duas primitivas:

- *invoke(operation)*: utilizada pelos clientes para invocar operações no serviço implementado pela RME.
- *execute(operation)*: implementada pelos servidores replicados, sendo utilizada sempre que uma operação deve ser executada pela RME.

A primitiva *invoke(operation)* é utilizada para enviar uma requisição *operation* para as réplicas, as quais executam a primitiva *execute(operation)* para processar *operation* e retornar a resposta a ser enviada para o cliente.

A implementação destas primitivas deve atender aos seguintes requisitos, que definem o determinismo de réplicas:

1. Estado inicial: todas as réplicas corretas devem iniciar a partir de um mesmo estado.
2. Determinismo: todas as réplicas corretas, que executam uma mesma operação sobre um mesmo estado, realizam a mesma mudança em seus estados e produzem a mesma resposta para o cliente.
3. Coordenação: todas as réplicas corretas executam a mesma sequência de operações.

Para garantir o item *(1.)* basta iniciar todas as réplicas com o mesmo estado (i.e., iniciar todas as variáveis que representam o estado com os mesmos valores nas diversas réplicas). Para prover o item *(2.)*, é necessário que as operações executadas pelos servidores sejam determinísticas. Finalmente, para que as réplicas executem a mesma sequência de operações afim de garantir o item *(3.)*, é necessária a utilização de um protocolo de difusão atômica [Hadzilacos and Toueg 1994]. Difusão atômica, ou difusão com ordem total, consiste em fazer com que todos os processos corretos, membros de um grupo, entreguem todas as mensagens difundidas neste grupo na mesma ordem.

Um resultado importante em sistemas distribuídos é que a difusão atômica e o consenso são problemas equivalentes. O *problema do consenso* [Hadzilacos and Toueg 1994] consiste em fazer com que todos os processos corretos acabem por decidir o mesmo valor, o qual deve ter sido previamente proposto por algum dos processos do sistema. Sendo assim, implementar difusão atômica através de um protocolo de consenso é trivial, pois basta que os processos utilizem o consenso para chegar a um acordo sobre a ordem de entrega das mensagens (operações).

Assumindo que os três requisitos acima sejam satisfeitos, uma implementação de RME deve satisfazer as seguintes propriedades de segurança e terminação:

- Segurança: todas as réplicas corretas executam a mesma sequência de operações.
- Terminação: todas as operações de clientes corretos são executadas.

A propriedade de segurança permite a implementação de consistência forte, conhecida como linearizabilidade (*linearizability*) [Herlihy and Wing 1990]. Já a propriedade de terminação garante que as operações invocadas por clientes corretos são executadas e acabarão terminando. Os protocolos propostos para RME limitam a quantidade de falhas que o sistema pode apresentar para que estas propriedades sejam atendidas, sendo necessário $n \geq 2f + 1$ (no modelo de falhas por parada - *crash*) [Lamport 1998] ou $n \geq 3f + 1$ (no modelo de falhas bizantinas) [Castro and Liskov 2002] réplicas para tolerar até f falhas. Outros requisitos necessários para a execução de uma RME incluem um sistema parcialmente síncrono [Dwork et al. 1988] para terminação [Schneider 1990, Lamport et al. 1982, Castro and Liskov 2002]. A ideia por trás destes modelos é de que o sistema trabalha de forma assíncrona (não respeitando nenhum limite de tempo) a maior parte do tempo. Porém, durante períodos de estabilidade, o tempo para transmissão de mensagens é limitado.

2.4.1. BFT-SMART: Implementação de Replicação Máquina de Estados

O BFT-SMART [Bessani et al. 2014] é uma biblioteca para implementação de aplicações tolerantes a falhas seguindo a abordagem de RME [Schneider 1990]. Inicialmente desenvolvida para tolerar falhas Bizantinas, o sistema pode ser configurado para tolerar apenas *crashes* (como no contexto deste trabalho). Esta biblioteca *open-source* de replicação foi desenvolvida em Java e implementa protocolos para reconfiguração e para gerenciamento de estados (*checkpoints*, atualização e transferência de estados). O leitor interessado pode encontrar uma descrição mais detalhada do BFT-SMART em [Bessani et al. 2014]. Resumidamente, cada réplica realiza as seguintes tarefas:

- **Recebimento de Requisições.** Os clientes enviam suas requisições para as réplicas, que as armazenam em filas diferentes para cada cliente. A autenticidade das requisições é garantida por meio de assinaturas digitais, i.e., os clientes assinam suas requisições (caso configurado). Desta forma, qualquer réplica consegue verificar a autenticidade das requisições e uma proposta para ordenação, que contém a requisição a ser ordenada, somente é aceita por uma réplica correta após a autenticidade desta requisição ser verificada.
- **Ordenamento de Requisições.** Sempre que existirem requisições para serem executadas, um protocolo de difusão atômica é executado onde uma instância de

um protocolo de consenso é inicializada por uma réplica (chamada de líder) para definir uma ordem de entrega para um lote de requisições. No BFT-SMART a ordenação é sequencial, i.e., uma nova instância do consenso só é inicializada após a instância anterior ter terminado. Porém, a ordenação de requisições em lotes visa aumentar o desempenho do sistema.

- **Execução de Requisições.** Quando a ordem de execução de um lote de requisições é definida, este lote é adicionado em uma fila para então ser entregue à aplicação. Após o processamento da cada requisição, uma resposta é enviada ao cliente que solicitou tal requisição.

A forma de utilização do BFT-SMART, para programação de uma aplicação tolerante a falhas através de RME, é bastante simples. A API básica para clientes e servidores fornece métodos para o cliente invocar o serviço replicado e uma espécie de *callback* para servidores executarem estas requisições/operações. Para acessar o serviço replicado, um cliente do BFT-SMART apenas deve instanciar uma classe *ServiceProxy* fornecendo seu identificador (inteiro) e um arquivo de configuração contendo o endereço (IP e porta) de cada um dos servidores. Após isso, sempre que o cliente desejar enviar alguma requisição para as réplicas (servidores), o mesmo deve invocar o método *invokeOrdered* especificando a requisição. Por outro lado, para implementar o servidor, cada réplica deve estender a interface *Executable* e implementar o método abstrato *executeOrdered* que é invocado quando uma requisição deve ser executada. Além disso, é necessário instanciar uma *ServiceReplica* que representa propriamente a réplica, fornecendo o identificador (inteiro) da réplica que é mapeado para uma porta e endereço IP através de um arquivo de configuração. Uma descrição aprofundada sobre a API para programação de aplicações sobre o BFT-SMART pode ser encontrada em [Bessani et al. 2014].

3. Arquitetura Proposta

A arquitetura proposta neste trabalho tem como objetivo automatizar a implantação de infra-estrutura necessária para promover a utilização de RME, através do acoplamento fraco entre o BFT-SMART e a aplicação de usuário que deseja-se executar. É importante destacar que esta separação da lógica do protocolo e do código da aplicação permite que qualquer aplicação, escrita em qualquer linguagem, sem a necessidade de qualquer biblioteca específica, possa, em princípio, utilizar a solução sem nenhuma modificação. Para a disponibilização deste tipo de serviço, a arquitetura proposta se beneficia dos recursos POD existentes no Kubernetes, que acoplam, de forma transparente, múltiplos contêineres. Vale ainda destacar que neste trabalho consideramos apenas falhas por *crash*[1].

3.1. Visão Geral

A arquitetura proposta (Figura 2) prevê a existência de três componentes principais: *Proxy*, No_SMaRtClient e No_SMaRtServer. Não há restrição quanto ao local de execução desses componentes, i.e., os mesmos podem ser executados em servidores e sub-redes diferentes, sem prejuízo ao funcionamento da arquitetura. O serviço de *Proxy* é executado no nó mestre do Kubernetes e é responsável por receber as requisições externas à arquitetura (vinda de clientes da aplicação) e endereçá-las internamente para os SMaRtClients

[1] Apensar do nome, o BFT-SMaRt pode ser configurado para tolerar apenas falhas por *crash*.

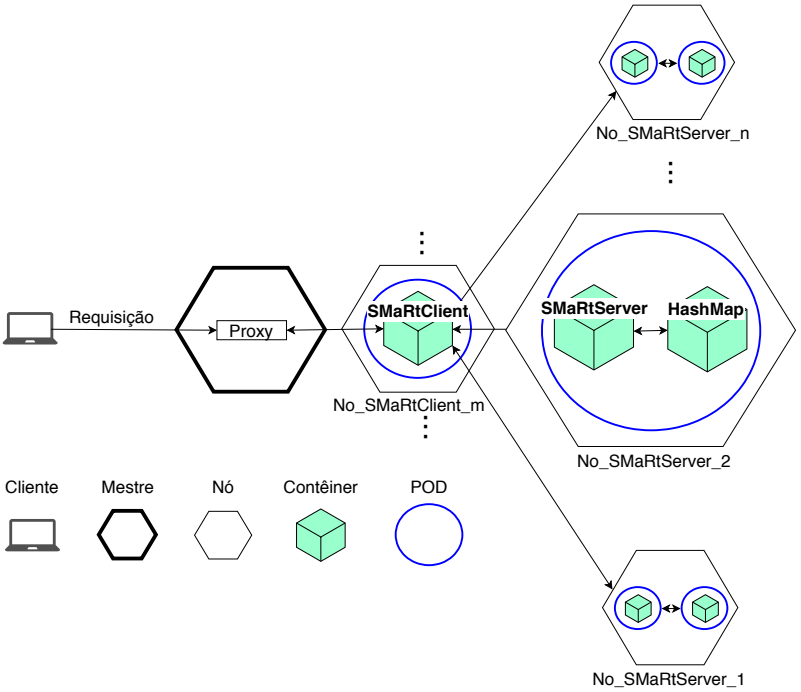


Figura 2. Arquitetura proposta.

(que estão executando em nós chamados No_SMaRtClient). O processo de distribuição de requisições é feito de forma balanceada entre todos os SMaRtClients instanciados. Ao receber uma requisição, o SMaRtClient selecionado irá encapsular esta requisição e distribuir, de acordo com o BFT-SMART, a todos os SMaRtServers. Cada POD instanciado em um No_SMaRtServer é formado por dois contêineres: um executando uma instancia do SMaRtServer e outro executando uma instancia da aplicação virtualizada.

Após a execução dos protocolos da RME pelo BFT-SMART (i.e., no momento de executar uma requisição), cada SMaRtServer encaminha as requisições para a sua respectiva aplicação (o outro contêiner formando o mesmo POD). As respostas, por sua vez, seguem o caminho inverso: a aplicação responde ao SMaRtServer, que encaminha ao SMaRtClient, que por sua vez irá receber a resposta de todos os SMaRtServers e irá decidir qual mensagem é a correta através dos protocolos do BFT-SMART, para então encaminhá-la ao cliente via *proxy*[2]. Vale destacar ainda que para manter um grau de transparência elevado para as aplicações, os SMaRtServers abrem uma conexão com os respectivos contêineres da aplicação para cada requisição a ser executada. Como não se sabe a natureza dos dados transmitidos, os SMaRtServers não sabem quando a resposta completa foi recebida, a menos que os contêineres com a aplicação fechem a conexão TCP após o término do envio da resposta.

3.2. Lidando com falhas nos No_SMaRtClients

No contexto deste trabalho estamos considerando apenas falhas por parada (*crash*) na nossa arquitetura. O BFT-SMART é responsável por mascarar as falhas que acontecem nos servidores (No_SMaRtServers), mas os No_SMaRtClients podem falhar durante a

[2]É importante observar que o Kubernetes possui soluções para lidar com falhas nos *proxies*.

execução e tanto não encaminhar as requisições aos servidores quanto a resposta ao cliente. As soluções seguintes solucionam estes problemas.

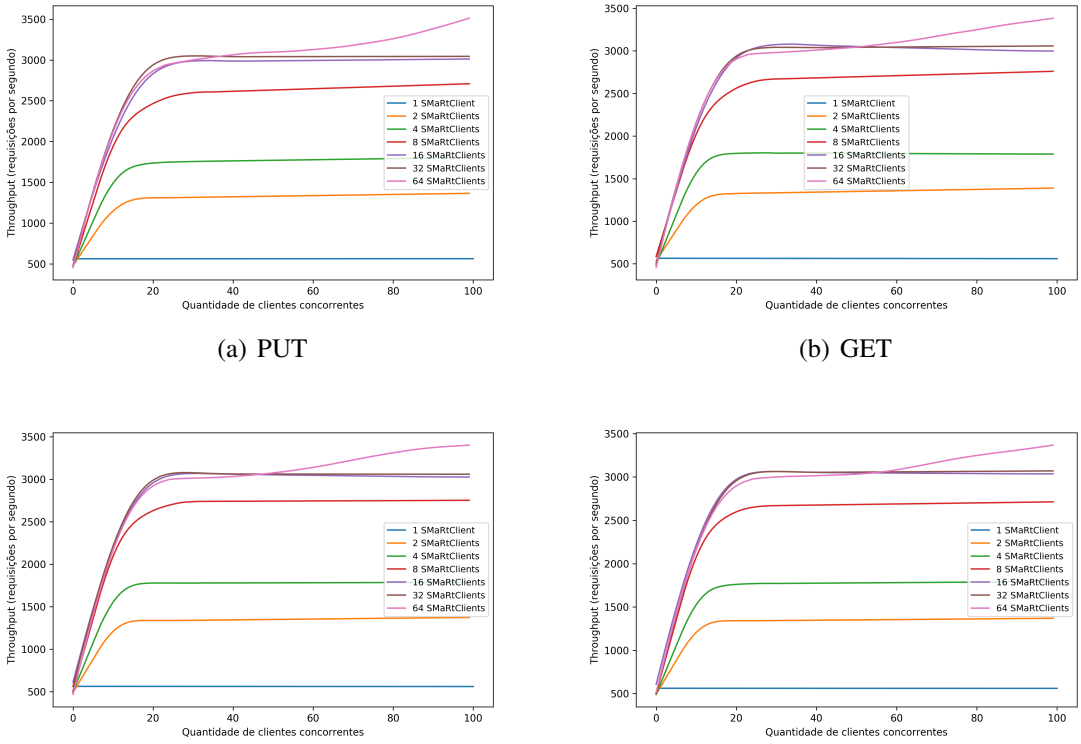
- Requisições incompletas: para resolver este problema, um *timeout* é associado a cada envio de requisição no cliente. Caso acontecer o *timeout* de uma requisição e a resposta ainda não foi obtida, o cliente solicita a execução desta requisição novamente e renova o *timeout*, até obter uma resposta. Em caso de reenvio de uma requisição, o Kubernetes removerá o componente faltoso e escolherá (através do balanceador de carga) outro NO_SMaRtClient para execução da requisição. A solução completa deste problema envolve a combinação do mecanismo descrito acima com o de eliminação de requisições duplicadas descrito a seguir.
- Requisições duplicadas: Com a execução de reenvios de requisições é possível que os servidores (SMaRtServers) recebam uma mesma requisição mais de uma vez para ser executada. Nestes casos, apenas uma requisição deve ser executada e as outras precisam ser eliminadas para manter a consistência do sistema. Para que isto seja possível, o cliente deve identificar cada requisição de forma única, através de sua identidade e de um número de sequencia, assim os servidores poderão verificar se tal requisição pode ser executada. Para este fim, cada servidor mantém um *buffer* onde são armazenadas as respostas correspondentes a última requisição de cada cliente. Assim, uma requisição é executada por um servidor apenas se ela tem um número de sequencia uma unidade maior que a invocação cuja resposta está no *buffer*. Caso a requisição recebida pelos servidores tenha o mesmo identificador da última executada para este cliente, a resposta armazenada no *buffer* é enviada ao SMaRtClient solicitante. Em qualquer outro caso a requisição é descartada. Deste modo, um cliente não pode solicitar a execução de uma requisição sem que a anterior esteja completamente atendida[3].

Tolerando SMaRtClients Maliciosos. Para tolerar SMaRtClients maliciosos, que podem modificar tanto a requisição quanto a resposta (ou ainda forjar requisições), mecanismos criptográficos (assinaturas digitais) precisam ser empregados tanto no cliente (externos ao Kubernetes que estão acessando o sistema) quanto nos servidores, a fim de garantir a autenticidade e a integridade das requisições e das respostas.

4. Experimentos

Visando analisar o desempenho da arquitetura proposta, bem como o comportamento de uma RME em execução integrada com o Kubernetes, realizamos uma série de experimentos em dois cenários distintos. No primeiro cenário, buscou-se avaliar o impacto quando há alteração no número de SMaRtClients, visto que os mesmos podem ser o gargalo do sistema pois são responsáveis pelo encaminhamento das requisições vindas dos clientes externos. Já o segundo cenário avalia o sistema quando há variação no número de nós do tipo No_SMaRtServers (que contém os servidores). A quantidade destes nós impacta no grau de tolerância a falhas do sistema: como consideramos apenas falhas por *crash*, temos que $n \geq 2f + 1$. Entretanto, aumentando a quantidade destes nós também leva a um aumento na quantidade de mensagens trocadas no sistema, o que impacta negativamente o seu desempenho.

[3]Esta limitação pode ser relaxada para k requisições se o servidor armazenar as últimas k respostas a cada cliente.



(a) PUT (b) GET

(c) REMOVE (d) SIZE

Figura 3. *Throughput* medido em função da variação do número de clientes e do número de SMaRtClients, com número de SMaRtServers igual a 3.

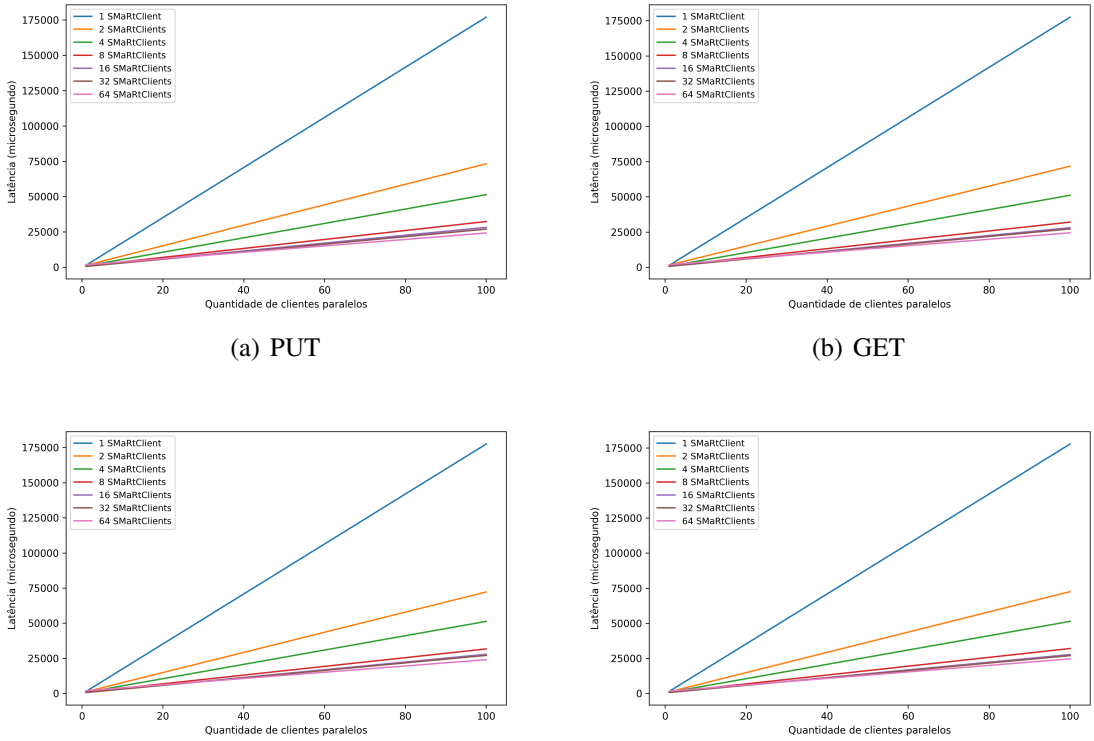
Aplicação. Para a avaliação da arquitetura proposta, implementamos uma aplicação de *hashmap*. As requisições implementadas foram as seguintes: PUT (inserir um par chave-valor), GET (dado uma chave, ler o valor associado), REMOVE (dado uma chave, remover o valor associado) e SIZE (retorna o número de entradas no *hashmap*).

Configurações. Para a validação da arquitetura proposta, foi utilizado um ambiente formado por três servidores homogêneos, em uma rede isolada. Os servidores apresentaram a seguinte configuração: dois processadores Intel Xeon E5620, 6 GB RAM (1067 Mhz), Ethernet 100 Mbps, sistema operacional Ubuntu Server LTS 16.04.3, Kubernetes 1.8.4 e BFT-SMART v1.1-beta. Um dos servidores foi usado para hospedar os No_SMaRtServers, um para os No_SMaRtClients e o último para os clientes do sistema.

Métricas. As métricas utilizadas na avaliação da arquitetura foram *throughput* e latência. Cada cliente executou $100k$ requisições e calculou a média da latência descartando os 5% dos valores com maior desvio. A cada $5k$ requisições executadas, os servidores calcularam o *throughput* e os valores reportados nesta seção referem-se à media destes valores. Para a execução do primeiro cenário, o número de SMaRtServers foi fixado em 3 e variou-se o número de SMaRtClients, assumindo os seguintes valores: 1, 2, 4, 8, 16, 32 e 64. Para o segundo cenário, o número de SMaRtClients foi fixado em 16, enquanto o número de SMaRtServers assumiu os seguintes valores: 3, 5, 7, 9 e 11.

Resultados e Análises. A Figura 3 apresenta os valores de *throughput* quando há variação

na quantidade de SMaRtClients. Dado o custo computacional aproximado na execução das operações PUT, GET, REMOVE e SIZE, os gráficos apresentam o mesmo comportamento e pouca variação de valores entre si. É possível verificar que o aumento no número de SMaRtClients tem um impacto no *thoughput*, que aumenta quando a quantidade destes nós aumenta até 16. A partir deste ponto, o aumento no número de SMaRtClients não se traduz no aumento do *throughput*, visto que esta camada não representa mais o gargalo do sistema. A quantidade de clientes paralelos é outro ponto a ser observado, pois o aumento em seu número reflete no aumento de carga submetida ao sistema. A saturação é obtida, em todas as operações, quando o número de clientes chega a aproximadamente 20.



(a) PUT (b) GET

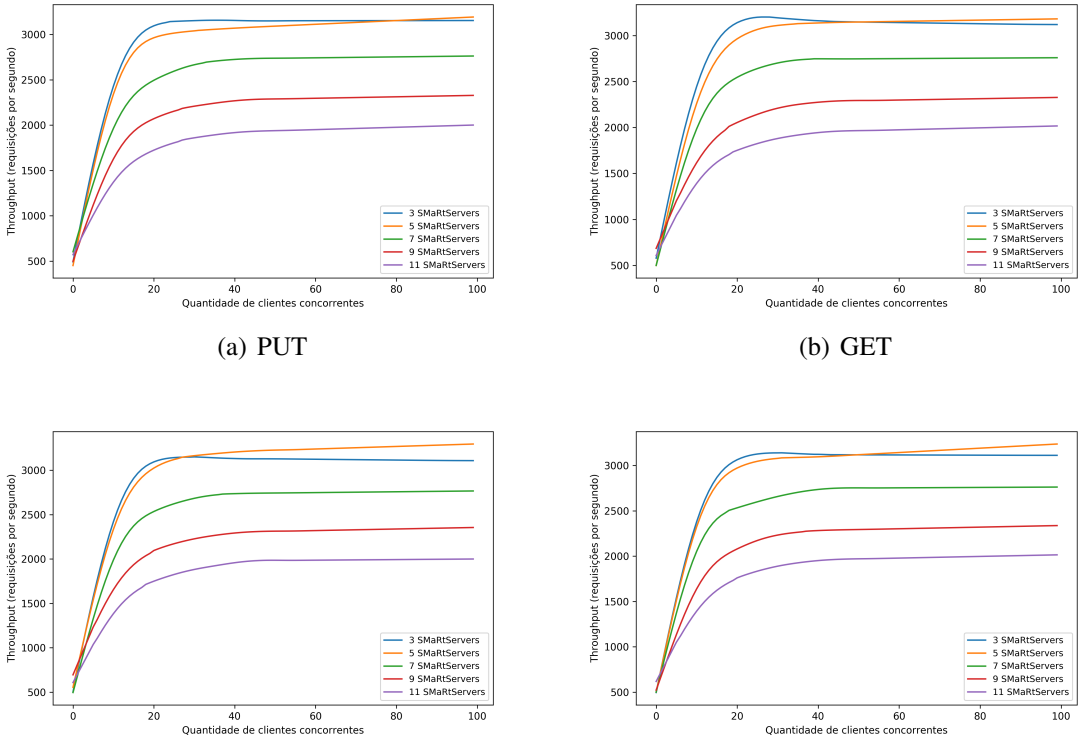
(c) REMOVE (d) SIZE

Figura 4. Latência medido em função da variação do número de clientes e do número de SMaRtClients, com número de SMaRtServers igual a 3.

A Figura 4 apresenta os valores de latência para o primeiro cenário. O aumento no número de SMaRtClients tem um impacto positivo na redução da latência. Assim como para o *throughput*, esta melhora é obtida até um certo ponto, no caso para até 8 SMaRtClients. Com os resultados deste primeiro cenário, é possível observar que quando o número de SMaRtClients é 16, atingimos os melhores valores de *throughput* e latência para a arquitetura proposta, i.e., esta camada deixa de ser o gargalo do sistema.

A Figura 5 apresenta os valores de *throughput* para o segundo cenário, onde fixamos o número de SMaRtClients em 16 de forma a não serem mais gargalo no sistema. Conforme esperado, o aumento no número de SMaRtServers acarreta na diminuição do *throughput*. Este comportamento está relacionado com a natureza dos protocolos de RME: um número maior de SMaRtServers implica em um maior número de mensagens

trocadas. Porém, vale destacar que com mais servidores também mais falhas são toleradas. Por fim, a Figura 6 apresenta os valores de latência. É possível observar que o aumento no número de SMaRtServers acarreta também no aumento da latência, apresentando um desempenho semelhante ao observado para o *throughput*.



(a) PUT (b) GET

(c) REMOVE (d) SIZE

Figura 5. *Throughput* medido em função da variação do número de clientes e do número de SMaRtServers, com número de SMaRtClients igual a 16.

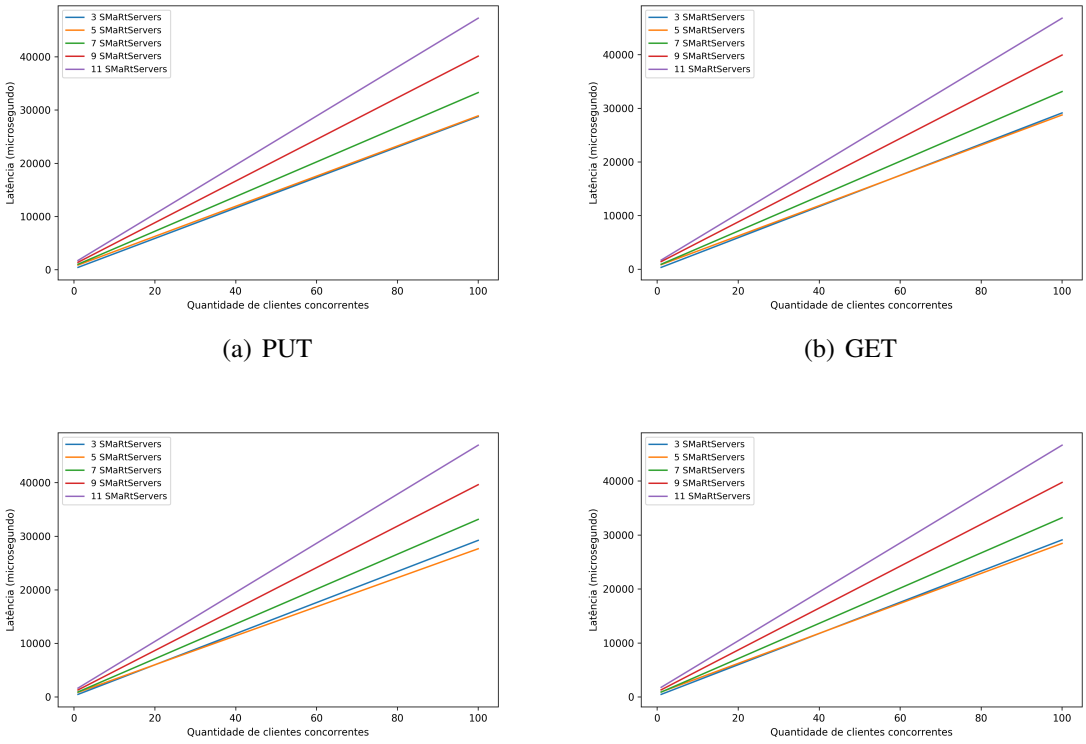
5. Conclusões

Este trabalho apresentou nossos esforços no sentido de introduzir suporte a tolerância a falhas em ambientes virtualizados, através da integração de uma biblioteca para RME no orquestrador de contêineres Kubernetes. A arquitetura resultante fornece um elevado grau de transparência tanto para desenvolvedores de aplicações quanto para os usuários finais das mesmas. Esta transparência é desejada para evitar que desenvolvedores precisem entender e se preocupar com os aspectos relacionados com a replicação, além de tornar viável a utilização desta arquitetura para introduzir tolerância a falhas em sistemas legados.

Como trabalhos futuros pretendemos investigar o comportamento de outras aplicações, com o intuito de entender melhor as peculiaridades da arquitetura proposta e identificar otimizações possíveis de serem aplicadas na mesma.

Referências

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



(a) PUT (b) GET

(c) REMOVE (d) SIZE

Figura 6. Latência medida em função da variação do número de clientes e do número de SMaRtServers, com número de SMaRtClients igual a 16.

Bessani, A., Sousa, J., and Alchieri, E. (2014). State machine replication for the masses with BFT-SMaRt. In *Proceedings of the International Conference on Dependable Systems and Networks*.

Burns, B., Grant, B., Oppeheimer, D., Brewer, E., and Wilkes, J. (2016). Lessons learned from three container-management systems over a decade: Borg, omega, and kubernetes. Queue Magazine, ACM.

Castro, M. and Liskov, B. (2002). Practical Byzantine fault-tolerance and proactive recovery. *ACM Transactions Computer Systems*, 20(4):398–461.

Dettoni, F., Lung, L. C., Correia, M., and Luiz, A. F. (2013). Replicação de máquina de estados tolerante a faltas bizantinas usando máquinas virtuais gêmeas. In *Anais do 31º Simpósio Brasileiro de Redes de Computadores - SBRC 2013*.

Docker (2018). What is docker. `https://www.docker.com/what-docker`. Acessado em Março de 2018.

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

Garfinkel, T. and Rosenblum, M. (2003). A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the Network and Distributed Systems Security Symposium*.

Goldberg, R. P. (1973). Architecture of virtual machines. In *Proceedings of the Workshop on Virtual Computer Systems*.

Goldberg, R. P. and Mager, P. S. (1979). Virtual machine technology: A bridge from large mainframes to networks of small computers. In *Compcon Fall 79. Proceedings*.

Hadzilacos, V. and Toueg, S. (1994). A modular approach to the specification and implementation of fault-tolerant broadcasts. Technical report, Department of Computer Science, Cornell.

Herlihy, M. and Wing, J. M. (1990). Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programing Languages and Systems*, 12(3):463–492.

Howard, H., Schwarzkopf, M., Madhavapeddy, A., and Crowcroft, J. (2015). Raft refloated: do we have consensus? *ACM SIGOPS Operating Systems Review*, 49(1):12–21.

Jiang, X. and Wang, X. (2007). "out-of-the-box"monitoring of vm-based high-interaction honeypots. In *Proceedings of the 10th International Conference on Recent Advances in Intrusion Detection*.

Lamport, L., Shostak, R., and Pease, M. (1982). The Byzantine generals problem. *ACM Transactions on Programing Languages and Systems*, 4(3):382–401.

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

Laureano, M., Maziero, C., and Jamhour, E. (2004). Intrusion detection in virtual machine environments. In *Euromicro Conference, 2004. Proceedings. 30th*.

Laureano, M. A. P. and Maziero, C. A. (2008). Virtualização: Conceitos e aplicações em segurança. In *VIII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais (Minicurso)*.

Merkel, D. (2014). Docker: lightweight linux containers for consistent development and deployment. *ACM Linux Journal*, 2014(239):1–8.

Netto, H., 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 XXXIV Simpósio Brasileiro de Redes de Computadores - SBRC 2016*.

Oliveira, C., Lung, L. C., Netto, H., and Rech, L. (2016). Evaluating raft in docker on kubernetes. In Świątek J. and J., T., editors, *International Conference on Systems Science (ICSS)*, volume 539 of *Advances in Intelligent Systems and Computing*, pages 123–130. Springer.

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

Schwarzkopf, M., Konwinski, A., Abd-El-Malek, M., and Wilkes, J. (2013). Omega: flexible, scalable schedulers for large compute clusters. European Conference on Computer Systems, ACM.