Replicação de Máquina de Estados Tolerante a Faltas Bizantinas usando Máquinas Virtuais Gêmeas

Fernando Dettoni[1], **Lau Cheuk Lung**[1,4], **Miguel Correia**[2], **Aldelir Fernando Luiz**[3,4]

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

[2]Instituto Superior Técnico - Universidade Técnica de Lisboa / INESC-ID - Portugal

[3]Câmpus Avançado de Blumenau - Instituto Federal Catarinense - Brasil

[4]Departamento de Automação e Sistemas - Universidade Federal de Santa Catarina - Brasil

{fdettoni, lau.lung}@inf.ufsc.br, miguel.p.correia@ist.utl.pt,

aldelir@das.ufsc.br

Abstract. *We present an architecture and an algorithm for Byzantine fault-tolerant state machine replication. Our algorithm explores the advantages of virtualization to reliably detect and tolerate faulty replicas, allowing the transformation of Byzantine faults into omission faults. Our approach reduces the total number of physical replicas from $3f + 1$ to $2f + 1$. Our approach is based on the concept of twin virtual machines, where there are two virtual machines in each physical host, each one acting as failure detector of its twin.*

Resumo. *Neste artigo é apresentada uma arquitetura e um algoritmo para replicação de máquina de estados tolerante a faltas bizantinas. São exploradas as vantagens fornecidas pela virtualização para detectar e tolerar réplicas faltosas, transformando faltas bizantinas em faltas de omissão. Com esta transformação, nossa abordagem reduz o número total de réplicas físicas necessárias de $3f + 1$ para $2f + 1$. Nossa abordagem é baseada no conceito de máquinas virtuais gêmeas, ou seja, na execução de duas máquinas virtuais em cada máquina física, cada uma funcionando como um detector de falhas de sua gêmea.*

1. Introdução

Cada vez mais, sistemas computacionais são usados em aplicações críticas e por isso necessitam operar corretamente apesar da presença de faltas. Estas faltas causam a parada total, como faltas de *crash*, ou arbitrárias, também chamadas de faltas bizantinas [Lamport et al. 1982]. Para garantir que o sistema funcione corretamente na presença de faltas é necessário o desenvolvimento de técnicas e algoritmos tolerantes a faltas bizantinas. Uma das técnicas mais utilizadas é a *replicação de máquina de estados* (RME) [Schneider 1990], que utiliza máquinas de estados determinísticas para oferecer um serviço replicado. Muitas abordagens tolerantes a faltas bizantinas (BFT) foram desenvolvidas utilizando RME (e.g. [Castro and Liskov 1999, Yin et al. 2003, Kotla et al. 2008]). Dentre estas, o algoritimo PBFT [Castro and Liskov 1999] é frequentemente considerado um pilar, pois foi o primeiro algoritmo com desempenho suficiente para muitas aplicações práticas.

Outra técnica de tolerância a faltas é o uso de *detectores de falhas não-confiáveis* [Chandra and Toueg 1996]. Apesar de originalmente criada para tolerar faltas de *crash*, algumas propostas posteriores foram capazes de estender a ideia para suportar faltas bizantinas [Doudou et al. 1999, Kihlstrom et al. 2003]. Estes detectores de falhas oferecem indicações sobre réplicas que aparentam estar faltosas.

A *virtualização* pode também ser considerada uma técnica de tolerância a faltas bizantinas pois introduz um nível de isolamento entre máquinas virtuais. Diversos trabalhos usam virtualização com o objetivo de proteger alguns componentes de falhas (ou ataques) de outros [Jiang and Wang 2007, Garfinkel and Rosenblum 2003, Laureano et al. 2004]. Tecnologias de virtualização são bastante aceitas pela indústria, e são largamente utilizadas atualmente, por exemplo por serviços de computação em nuvens como Amazon Web Services e Windows Azure.

O PBFT e vários outros algoritmos de replicação de máquinas de estados BFT têm um alto custo de implementação pois possuem uma resiliência de $n \geq 3f + 1$, ou seja, precisam de $n > 3f$ réplicas para tolerar f réplicas faltosas. Para diminuir esse custo, surgiram algumas abordagens que usam um *componente confiável* para limitar o comportamento das réplicas faltosas usando apenas $n \geq 2f + 1$ réplicas [Correia et al. 2004, Chun et al. 2007, Veronese et al. 2013]. Foram propostas também abordagens para executar apenas $f + 1$ réplicas, mantendo outras $2f$ *em espera*, ou seja, sem consumir tempo de CPU mas sendo ativadas em caso de falha [Distler et al. 2011, Wood et al. 2011].

Este artigo explora um outro ponto do espaço de projeto. O artigo apresenta uma nova arquitetura, chamada TwinBFT, para replicação de máquina de estados tolerante a faltas bizantinas eficiente baseada em virtualização. O objetivo é reduzir de $n \geq 3f + 1$ para $n \geq 2f + 1$ o número de máquinas físicas necessárias para tolerar f faltas. Além disso, o algoritmo apresentado reduz o número de passos de comunicação em funcionamento normal de 5 (do PBFT) para 3, sem a participação do cliente no acordo. Até onde sabemos, este é o primeiro algoritmo com este número de passos sem adotar uma abordagem especulativa [Kotla et al. 2008, Veronese et al. 2013], a qual envolve a participação do cliente no acordo.

Nossa proposta consiste na utilização de conjuntos de *máquinas virtuais gêmeas* executando o mesmo serviço da aplicação em cada uma das $n \geq 2f + 1$ máquinas físicas do sistema. Por simplicidade, neste artigo, adotaremos conjuntos de duas máquinas virtuais apenas. Cada máquina virtual executa o mesmo serviço, e cada conjunto de máquinas virtuais atua como uma réplica do esquema de replicação de máquina de estados. A ideia principal da proposta é utilizar cada máquina virtual como um detector de falhas para sua gêmea: ao enviar uma requisição para duas máquinas gêmeas, ambas devem fornecer a mesma resposta, caso contrário, ambas serão consideradas faltosas e sua resposta pode ser ignorada pelas outras réplicas. Assim, cada conjunto de máquinas virtuais gêmeas pode apenas funcionar corretamente ou omitir mensagens. No entanto, essas omissões são toleradas pela replicação de máquina de estados. Um *crash* corresponde também a omissões por tempo indeterminado, logo é também tolerado pelo uso de replicação.

A arquitetura e o algoritmo apresentados no artigo não pretendem oferecer a solução ideal para todos os casos. A utilização de máquinas virtuais é adequada para empresas que utilizam este tipo de tecnologia, como as de computação em nuvens. É também indicada quando não estejam disponíveis componentes confiáveis e ou não se possa esperar pela ativação de réplicas em espera em caso de falha. Nesses casos uma solução para replicação de máquinas de estados BFT eficiente – com apenas $2f + 1$ réplicas físicas ($4f + 2$ virtuais) – usando virtualização pode ser adequada.

Na Seção 2, serão mostrados alguns trabalhos relacionados, apresentando o estado da arte. A Seção 3 apresenta uma descrição do nosso modelo de sistema e suposições. Uma explicação detalhada do algoritmo será apresentada na Seção 4. Após isso, a Seção

5 apresenta algumas avaliações da implementação do algoritmo e a Seção 6 resume as conclusões.

2. Trabalhos Relacionados

Muitas propostas surgiram recentemente com o intuito de produzir serviços tolerantes a intrusão baseados em replicação de máquina de estados tolerante a faltas bizantinas (BFT). Sendo a primeira abordagem prática para protocolos BFT, o PBFT [Castro and Liskov 1999] é um dos protocolos mais bem sucedidos. Apesar de ser prático, os custos de implementação do PBFT são relativamente elevados, necessitando pelo menos 4 réplicas ($3f+1$ para $f = 1$) e 5 passos de comunicação. Desta forma, surgiram vários algoritmos derivados do PBFT com dois objetivos: diminuir o número de réplicas e melhorar o desempenho.

Vários trabalhos propuseram alternativas para melhorar a resiliência reduzindo o número de réplicas. Yin *et al.* [Yin et al. 2003] introduziram uma arquitetura separando o serviço em duas camadas, uma responsável pelo acordo, contendo $3f + 1$ réplicas, e outra responsável por executar as requisições com apenas $2f+1$ réplicas. Apesar de ainda serem precisas $3f + 1$ réplicas, as réplicas utilizadas para a execução do serviço tendem a ter um custo bem mais elevados do que as réplicas de acordo.

Correia *et al.* [Correia et al. 2004] apresentaram a primeira solução para executar replicação de máquina de estados BFT com apenas $2f + 1$ réplicas, utilizando um componente confiável distribuído. Mais tarde outro trabalho apresentou o primeiro algoritmo do gênero baseado num componente confiável local que implementa a abstração de *attested append-only memory* [Chun et al. 2007]. Veronese *et al.* [Veronese et al. 2013] propõem dois algoritmos que utilizam um componente confiável que fornece identificadores únicos para cada mensagem. O primeiro deles, MinBFT é capaz de reduzir o número de réplicas necessárias para $2f + 1$ e o número de passos de comunicação para 4. O segundo é um algoritmo especulativo chamado MinZyzzyva que reduz o número de passos ainda mais, para 3, mantendo o número de réplicas em $2f + 1$.

Em outra abordagem, Stumm *et al.* [Stumm et al. 2010] tiram vantagem de técnicas de virtualização para reduzir o número de réplicas necessárias para $2f+1$ desde que a VMM forneça uma forma de comunicação segura entre as réplicas. Esta abordagem, entretanto, requer que todas as réplicas estejam na mesma máquina e, portanto, não tolera faltas de *crash* na máquina física, ao contrário deste artigo.

Outra abordagem baseada na ideia de duas réplicas monitorando uma à outra é apresentada em [Mpoeleng et al. 2003], utilizando a abordagem *signal-on-fail*. Essa abordagem precisa de $4f + 2$ máquinas físicas e requer comunicação síncrona e confiável entre cada par de réplicas, o que é um pressuposto difícil de garantir na prática. Nesta mesma linha, Inayat e Ezhilchevan apresentam um protocolo multicast BFT otimista com ordenação total baseado na abordagem *signal-on-fail*. Os autores demonstram que em uma execução livre de falhas, a abordagem tende a apresentar uma melhor performance do que outras abordagens BFT [Inayat and Ezhilchelvan 2006].

Vários trabalhos apresentaram soluções para melhorar o desempenho do PBFT. Cowling *et al.* [Cowling et al. 2006] apresentaram o HQ, um algoritmo baseado em quóruns e no PBFT que tem muito bom desempenho quando não há concorrência no acesso a dados. Kotla *et al.* [Kotla et al. 2008] apresentaram o Zyzzyva, um algoritmo que reduz o número de passos de comunicação na ausência de faltas. Ao invés de tentar chegar a um

acordo entre as réplicas antes de enviar a resposta, o serviço responde especulativamente ao cliente. O serviço precisa executar novamente a requisição e chegar a um acordo apenas no caso das respostas recebidas pelo cliente divergirem uma das outras. Esta abordagem se aproveita do fato de a maior parte das requisições ser livre de faltas, mas requer a habilidade de se reverter operações executadas previamente.

Como já mencionado na introdução, este artigo explora um outro ponto do espaço de projeto, usando virtualização para implementar replicação de máquina de estados BFT em apenas $2f + 1$ réplicas físicas (e o dobro de máquinas virtuais) e com apenas 3 passos de comunicação em funcionamento normal.

Muitos trabalhos usam virtualização para isolar componentes de software entre si. Dois dos primeiros usam virtualização para proteger um detector de intrusões dos próprios intrusos [Garfinkel and Rosenblum 2003, Laureano et al. 2004], enquanto outro mais recente usa a mesma ideia para proteger o monitor de um *honeypot* [Jiang and Wang 2007]. No entanto, a segurança do hipervisor é essencial para obter isolamento, por isso alguns trabalhos estudaram como melhorar essa segurança. Murray *et al.* [Murray et al. 2008] propuseram *desagregar* o sistema de virtualização como solução para diminuir a *trusted computing base* do sistema. O sistema NoHype vai mais longe retirando o hipervisor do caminho e executando as máquinas virtuais de modo nativo [Szefer et al. 2011]. O sistema HyperSafe usa outra abordagem: protege hipervisor de modo a detectar ataques que modifiquem o fluxo de controle, como *buffer overflows* [Wang and Jiang 2010].

3. Modelo de Sistema

Uma representação da arquitetura do sistema é mostrada na Figura 1. O sistema é composto por um conjunto de *n* máquinas físicas (ou *hosts*) *H* = $\{h_1, h_2, \ldots, h_n\}$ sendo que $n \geq 2f + 1$ e f é o número máximo de máquinas físicas faltosas. Cada *host* da Figura 1 contém um gerenciador de máquinas virtuais (VMM ou hipervisor) com duas máquinas virtuais (vm_i, vm'_i),chamadas gêmeas, executando emcada uma, uma réplica ou processo, respectivamente p_i e p'_i. Ambos os processos $\{p_i, p'_i\}$ executam o mesmo serviço e se comunicam entre si para validar cada mensagem de saída antes de enviar para outros processos.

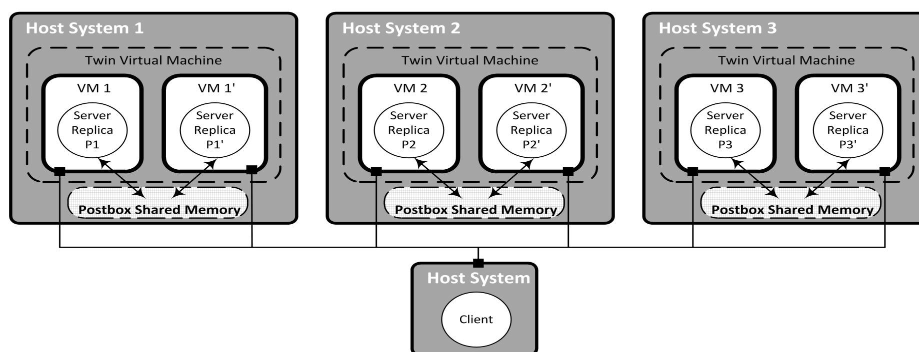


Figura 1. TwinBFT - Arquitetura com Máquinas Virtuais Gêmeas.

Assumimos que até f máquinas virtuais podem falhar de forma bizantina, ou arbitrária, mas apenas uma em cada máquina física. Quando uma máquina virtual falha arbitrariamente, o mecanismo de validação transforma essa falha numa omissão. Assim, assumimos também que até f máquinas físicas podem falhar por paragem ou omitindo mensagens, acidentalmente ou devido à falha de uma das suas máquinas virtuais. Para

substanciar o limite de f falhas é necessário recorrer a *diversidade*, ou seja, implementações diferentes dos processos e máquinas físicas [Bessani et al. 2009, Garcia et al. 2011, Gashi et al. 2007]. Essa diversidade reduz a chance de máquinas virtuais de um mesmo *host* sofrerem intrusão simultaneamente. Assumimos também que a virtualização fornece isolamento entre as máquinas virtuais e do próprio VMM / hipervisor. Como já explicado no final da Seção 2, técnicas como desagregação ou mecanismos como o HyperSafe e o NoHype podem ser usadas para tornar esta premissa mais realista.

Nenhuma suposição é feita sobre o tempo necessário para o sistema computar uma mensagem. A comunicação entre diferentes VMs dentro de um mesmo *host* é feita por um espaço de memória compartilhada, chamada *postbox*. Os processos nas VMs, em diferentes hosts, se comunicam pela rede, apenas por troca de mensagens. Esta rede pode falhar ao entregar mensagens, entregar fora de ordem, atrasar, ou duplicar mensagens.

Cada *host* pode assumir dois diferentes papéis: (1) *host* primário, sendo responsável por definir a ordem em que as requisições dos clientes serão executadas; e (2) *host* backup, que executa as requisições seguindo a ordem proposta pelo primário. Dentro de um *host* primário, um processo pode assumir dois diferentes papéis: (1) líder, que é responsável por atribuir um número de sequência para cada requisição recebida; e (2) seguidor, que executa as requisições seguindo a ordem definida pelo líder. Todos os processos em *hosts* backups são considerados seguidores. O *host* primário h_j é definido por $j = v \bmod |S|$, sendo *v* a visão atual, conforme definido na próxima seção. O processo líder primário em um *host* é, por definição, p_j.

Utilizamos técnicas criptográficas para autenticar mensagens e garantir sua autenticidade. Cada par de processos compartilha entre si uma chave secreta usada para gerar um vetor de MACs (*Message Authentication Code*) [Tsudik 1992] com um MAC válido para cada processo.

Por simplicidade, assumimos que cada máquina física comporta apenas duas máquinas virtuais (VMs), onde, para uma dada entrada, as respostas fornecidas por ambas têm que ser a mesma para que a máquina física não seja considerada faltosa. No entanto, o modelo pode facilmente ser estendido para mais VMs, segundo a condição $nVM \geq 2fVM + 1$, onde fVM é o número máximo de máquinas virtuais faltosas em uma máquina hospedeira. Neste caso, a máquina hospedeira não será considerada faltosa se a maioria das VMs ($fVM + 1$) fornecerem a mesma resposta.

4. Algoritmo TwinBFT

O algoritmo implementa uma replicação de máquina de estados no modelo de sistema que acabamos de apresentar. As réplicas se alternam seus papéis por uma sucessão de configurações chamadas de visão. Em cada visão, temos uma réplica primária que é responsável por definir a ordem das mensagens e encaminhar as requisições para todas as réplicas. Como mostrado por Schneider [Schneider 1990], a máquina de estados deve ser determinística e todas as réplicas precisam iniciar em um mesmo estado, caso contrário, a propriedade *safety* não pode ser garantida.

Nesta seção é apresentada nossa proposta de transformação das faltas bizantinas em faltas de omissão. Neste sentido, será apresentada uma adaptação do protocolo PBFT [Castro and Liskov 1999] para o modelo proposto. Em cada visão, apenas um réplica p_j é primária (líder), e é responsável por definir a ordem das mensagens e encaminhar as requisições para as réplicas do serviço. Se uma mensagem enviada por um *host* qualquer

for assinada por ambas as VMs deste *host*, ou seja, ambas resposta são iguais, assumimos esta mensagem como correta, já que apenas uma VM pode falhar ao mesmo tempo em um mesmo *host*.

4.1. Propriedades

Sendo uma Replicação de Máquina de Estados, é necessário assegurar as seguintes propriedades para garantir a corretude do serviço:

- **Ordem Total** (*safety*): cada requisição é executada sequencialmente na mesma ordem em cada réplica, i.e. apesar da replicação, as operações são executadas da mesma forma como seriam em um sistema centralizado;
- **Terminação** (*liveness*): qualquer requisição iniciada pelo cliente é terminada em algum momento, mesmo em caso de falha.

O algoritmo proposto fornece tanto *safety* quanto *liveness*, assumindo que não mais do que $f = \lfloor \frac{n-1}{2} \rfloor$ *hosts* são faltosos e, ao menos um processo *p* seja correto em cada *host* faltoso. Para garantir que as réplicas executarão as requisições na mesma ordem, todas as réplicas seguem a ordem definida pelo líder e esta ordem pode ser considerada correta desde que assinada por ambos os processos na réplica primária. Os algoritmos asseguram a corretude (ou *safety*) apesar do tempo levado para o processamento das requisições, porém uma certa sincronia é necessária para garantir as propriedades de progresso (i.e. *liveness*).

Como todas as réplicas seguem a ordem definida pelo líder primário, não é necessário um algoritmo de consenso pois pode-se confiar na ordem definida pela réplica primária, desde que as suposições prévias não sejam violadas. Isto ocorre porque quando o primário define a ordem, esta ordem apenas será seguida se ambos os processos no *host* primário tiverem acordo.

4.2. Detalhamento dos Algoritmos

Nesta seção o algoritmo será apresentado em detalhes. Primeiro, na Figura 2 é mostrado um diagrama com uma visão geral dos passos do protocolo. A configuração mostrada assume $f = 1$, sendo necessários três *hosts*, cada um com duas VMs. Cada par de VMs em um mesmo *host* se comunica através de um canal confiável FIFO chamado *postbox*. A *postbox* pode ser mais rápida do que a comunicação via rede, usando um espaço de memória compartilhado entre as VMs, fornecido pela VMM.

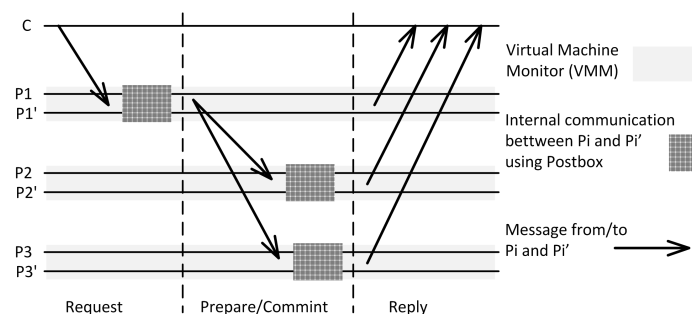


Figura 2. Passos do algoritmo em execução normal com $f = 1$.

O algoritmo funciona basicamente como segue:

1. Cliente envia a requisição para ambos os processos na réplica primária;

2. O líder primário p_i define um número de sequência e insere uma mensagem "ORDER"na *postbox*;
3. O seguidor primário p'_i lê a mensagem da *postbox*, pega o número de sequência e insere na postbox a mensagem "ORDER" contendo a requisição original e o número de sequência recebido;
4. Ambos os processos assinam a mensagem "ORDER" lida da *postbox* e enviam para todas as réplicas;
5. Assim que cada processo nas réplicas recebe a mensagem "ORDER", executa a operação e insere na *postbox* uma mensagem "REPLY" assinada;
6. Quando um processo lê da *postbox* uma mensagem "REPLY", compara com a mensagem gerada localmente e, se todos os argumentos forem idênticos, adiciona sua assinatura e envia a resposta para o cliente;
7. Se o cliente receber ao menos $f + 1$ respostas corretamente assinadas de diferentes réplicas, aceita a resposta.

Como um exemplo de cliente, o Algoritmo 1 mostra uma execução normal em que o cliente envia uma requisição (linha 3) para o serviço e aguarda até receber ao menos $f + 1$ respostas válidas de diferentes réplicas (linhas 4-6). A requisição tem o formato $\langle$REQUEST, c, seq, op$\rangle_{\sigma_c}$ sendo que *c* é o identificador do cliente, *seq* é o número de sequência em relação ao cliente, e *op* é a operação a ser executada. Se o cliente não receber $f+1$ respostas em um determinado tempo, reenvia a requisição para todas as réplicas (linha 8).

Algoritmo 1 Algoritmo executado pelo cliente

```
1: procedure REQUEST                                        ▷ Envia uma nova requisição
2:     Δc ← default timeout
3:     multi_send(⟨REQUEST, c, seq, op⟩σc)   ▷ Envia a requisição para ambos os processos na réplica primária
4:     repeat
5:         buffer ← buffer ∪ recv()
6:     until f + 1 respostas correspondentes ∃ buffer /* Timer em uma thread separada */
7:     if Δc expired then
8:         multi_send(⟨REQUEST, c, seq, op⟩σc)                ▷ Envia a mensagem para todas as réplicas
9:     end if
10: end procedure
```

4.3. Operação normal do protocolo

O Algoritmo 2, executado em cada um dos processos possui duas tarefas concorrentes. A Tarefa 1 é responsável por ler as mensagens recebidas através da rede. A Tarefa 2 é responsável por ler as mensagens recebidas a partir da *postbox*, inseridas por sua gêmea. O estado de cada processo é composto pelo estado do serviço, um *buffer* de mensagens e a visão atual. Este estado é compartilhado entre as tarefas.

Quando qualquer um dos processos $\{p_i, p'_i\}$ no *host* primário recebe a requisição do cliente, p_i gera um novo número de sequência *n* e cria uma mensagem $\langle\langle$ORDER, p_i, v, n, dm$\rangle_{\sigma_{p_i}}$, m$\rangle$, sendo *v* o número da visão atual, e *dm* o resumo da mensagem *m* (linhas 4-6). Assim que p'_i lê a mensagem "ORDER" inserida na postbox por p_i e possui sua respectiva mensagem "REQUEST"no buffer, pega o número de sequência proposto por p_i, cria uma mensagem "ORDER", assina e insere na *postbox* (linha 23). Quando cada um dos processos lê a mensagem "ORDER" da *postbox*, verifica se todos os parâmetros correspondem aos processados localmente e, em caso positivo, adiciona sua própria assinatura à mensagem de sua gêmea e envia para todas as réplicas (linha 25).

Algoritmo 2 Algoritmo em operação normal

```
/* Tarefa 1: rede */
1: loop
2:   msg ← receive()
3:   if received (REQUEST) then
4:     if é o líder primário then
5:       n ← n + 1
6:       postbox.append(⟨⟨ORDER, p_i, v, n, dm⟩_σp_i , msg⟩)
7:     else if é o seguidor primário then
8:       buffer ← buffer ∪ msg
9:     else
10:      envia mensagem para primários
11:      inicia Δ_p
12:    end if
13:  else if received (ORDER) then
14:    termina Δ_p
15:    postbox.append(⟨REPLY, p_i, v, seq, c, res⟩ >_σp_i )
16:  end if
17: end loop
    /* Tarefa 2: postbox */
18: loop
19:   msg ← postbox.read()
20:   if received (ORDER) then
21:     if todos os parâmetros correspondem aos computados localmente then
22:       if is the primary follower then
23:         postbox.append(⟨⟨ORDER, p_i, v, m.n, dm⟩_σp_i , msg⟩)
24:       end if
25:       multicast(⟨⟨⟨ORDER, p'_i, v, n, dm⟩_σp'_i ⟩σp_i , msg⟩)
26:     end if
27:   else if received (REPLY) then
28:     if todos os parâmetros correspondem aos computados localmente then
29:       reply_to_client(⟨⟨REPLY, p'_i, v, seq, c, res⟩_σp'_i ⟩_σp_i )
30:     end if
31:   end if
32: end loop
```

Para cada mensagem “ORDER” recebida, as réplicas consideram válida caso as seguintes condições estejam cumpridas:

- A mensagem é corretamente assinada, i.e., se recebida pela rede deve estar assinada por ambas máquinas gêmeas no remetente, e se recebida pela *postbox* deve estar assinada pela sua própria gêmea;
- A visão na mensagem é a visão atual;
- Não aceitou outra mensagem “ORDER” com o mesmo número de sequência para uma requisição diferente;
- A número de sequência está entre um valor mínimo *h* e máximo *H* de possíveis números de sequência (na prática, se esta verificação for feita pelo seguidor primário quando lê a mensagem “ORDER"da *postbox* as réplicas *backup* nunca receberão um número de sequência fora de *h* e *H*).

Ao receber uma mensagem “ORDER” de ambos os processos em uma máquina física, cada um dos processos $\{p_i, p'_i\}$ verifica se a mensagem é válida e, em caso positivo, executa operação e cria a mensagem $\langle$REPLY, p_i, v, seq, c, res$\rangle_{\sigma_{p_i}}$, sendo *res* o resultado da operação executada, e insere na *postbox* (linha 15). Quando um processo lê a mensagem “REPLY” da *postbox*, compara os seus parâmetros e, se forem idênticos aos processados localmente, assina a mensagem e envia ao cliente (linha 29).

Quando o cliente recebe uma mensagem “REPLY”, aceita como válida se as seguintes condições forem verdadeiras:

- Está assinada por ambos os processos $\{p_i, p'_i\}$ no *host* remetente.
- Ainda não aceitou uma mensagem válida remetida pela gêmea do *host* remetente.

O cliente aguarda até ter recebido ao menos $f + 1$ mensagens válidas das réplicas para aceitar o resultado. Se estas mensagens não forem recebidas em um determinado tempo, envia a requisição para todas as réplicas, podendo ocorrer uma troca de visão por suspeita do primário.

4.4. Maior Número de Máquinas Gêmeas

Conforme citado anteriormente, é possível que mais do que duas máquinas virtuais componham o grupo de máquinas gêmeas em cada *host*. Desta forma, o algoritmo confere uma maior resistência permitindo que uma ou mais máquinas virtuais se comportem de forma bizantina e ainda assim o *host* seja considerado correto, desde que exista uma maioria de respostas idênticas. Esta maioria é indicada pela quantidade de assinaturas presentes em cada mensagem.

Desta forma, ao criar uma nova mensagem, cada processo publica sua proposta de mensagem assinada na *postbox* (linhas 6 e 15). Ao ler as mensagens da postbox (linhas 21 e 28), é necessário verificar também se, juntamente com sua assinatura, a mensagem estará assinada por uma maioria das máquinas virtuais do *host*. Em caso positivo, assina e envia para o destinatário. Caso contrário, apenas insere novamente na *postbox* a mensagem com sua assinatura até que esteja assinada por uma quantidade suficiente de máquinas virtuais no *host*.

4.5. Coleta de Lixo

Para prevenir que ocorra um estouro na memória, é necessário um mecanismo que descarte as mensagens antigas armazenadas no *buffer*, isto é, que efetue uma coleta de lixo (i.e. *garbage collection*). Para isso, o algoritmo gera periodicamente um *checkpoint*, após algum número constante de requisições recebidas. Ao chegar em um *checkpoint*, cada processo cria uma mensagem $\langle$CHECKPOINT, p_i, v, n, d$\rangle_{\sigma_{p_i}}$, sendo que *n* é o número da última requisição processada e *d* é um sumário do estado atual de p_i, e insere na *postbox*.

Cada máquina gêmea lerá a mensagem da *postbox* e assim que atingir o mesmo *checkpoint*, confirma se o estado recebido é o mesmo estado local e, em caso positivo, adiciona sua própria assinatura à mensagem "CHECKPOINT" e envia para as outras réplicas. Quando um processo recebe $f + 1$ mensagens "CHECKPOINT" corretamente assinadas de diferentes réplicas, para o mesmo número de visão *v*, o mesmo número de sequência *n* e o mesmo estado *d*, aceita como último checkpoint válido e remove do buffer todas as mensagens com número de sequência menor do que *n*.

4.6. Protocolo de Troca de Visão

A principal função do protocolo de troca de visão é manter o serviço progredindo mesmo na presença de um primário faltoso. Se o primário é faltoso, as réplicas backup nunca receberão uma mensagem "ORDER" válida e, portanto, devem definir um novo primário. Se um cliente não receber respostas suficientes em um tempo hábil, envia a requisição para todos os processos do sistema. Se uma réplica backup recebe uma requisição diretamente do cliente, verifica se já processou esta requisição e, em caso positivo, reenvia a resposta ao cliente. Caso contrário, encaminha a requisição aos processos da réplica primária e inicia um contador Δ_p (linhas 6-11).

Ao receber a mensagem "ORDER" correspondente do primário, o contador Δ_p é cancelado (linha 14) e o algoritmo continua normalmente. Se nenhuma mensagem "ORDER for recebida até o contador expirar, o processo inicia o protocolo de troca de visão,

apresentado no Algoritmo 3, inserindo na *postbox* a mensagem ⟨VIEW-CHANGE, p_i, v+1, n, C, P⟩$_{\sigma_{p_i}}$, sendo *n* o número de sequência do último checkpoint válido, *C* um conjunto composto por $f+1$ mensagens "CHECKPOINT" garantindo o último *checkpoint*, e *P* um conjunto com todas as requisições processadas após o último *checkpoint* (linha 2). Se sua gêmea concordar com a troca de visão, a partir da mensagem "VIEW-CHANGE" lida da *postbox*, adiciona sua própria assinatura e envia a mensagem para todos os processos (linha 5).

Algoritmo 3 Algoritmo do protocolo de troca de visão

```
/* Tarefa 1: rede */
 1: loop
 2:   msg ← receive()
 3:   if received (VIEW-CHANGE) then
 4:     buffer ← buffer ∪ msg
 5:     if buffer contém ao menos um VIEW-CHANGE com n = msg.n ∧ d = msg.d then
 6:       envia mensagem aos primários
 7:       if i = msg.v mod |S| then
 8:         postbox.append(⟨NEW-VIEW, p_i, msg.v, V, P⟩σ_{p_i})
 9:       end if
10:     end if
11:   else if received (NEW-VIEW) then
12:     buffer ← buffer ∪ msg
13:     for all req in msg.P do
14:       garante que req foi processada e armazenada no log.
15:     end for
16:   end if
17: end loop
/* Tarefa 2: postbox */
 1: loop
 2:   msg ← postbox.read()
 3:   if received (VIEW-CHANGE) then
 4:     if all parameters corresponds the ones locally computed then
 5:       multi_send(⟨⟨VIEW-CHANGE, p'_i, v+1, n, C, P⟩σ_{p'_i}⟩σ_{p_i})
 6:     end if
 7:   else if received (NEW-VIEW) then
 8:     if todos os parâmetros correspondem aos computados localmente then
 9:       multi_send(⟨⟨NEW-VIEW, p'_i, v+1, V, P⟩σ_{p'_i}⟩σ_{p_i})
10:     end if
11:   end if
12: end loop
/* Tarefa 3: timeout */
 1: procedure TIMEOUT_EXPIRE                         ▷ Ao expirar o timeout Δ_p
 2:   postbox.append(⟨VIEW-CHANGE, p_i, v+1, n, C, P⟨σ_{p_i})
 3: end procedure
```

Ao receber $f+1$ mensagens "VIEW-CHANGE" válidas de diferentes *hosts*, ambos os processos $\{p_i, p'_i\}$ no *host* h_i verificam se h_i é primário. Se sim, *p* aceita a troca de visão criando e inserindo na *postbox* a mensagem ⟨NEW-VIEW, p_i, v+1, V, P⟩$_{\sigma_{p_i}}$, sendo *V* um conjunto contendo as mensagens "VIEW-CHANGE" que originaram a troca de visão, e *P* um conjunto contendo as mensagens "ORDER" enviadas após o último *checkpoint* válido (linhas 5-8). Quando um processo *p* lê da *postbox* uma mensagem "NEW-VIEW", verifica se seus parâmetros são idênticos aos processados localmente e, caso sejam, adiciona sua assinatura e envia a mensagem para todas as réplicas (linha 9).

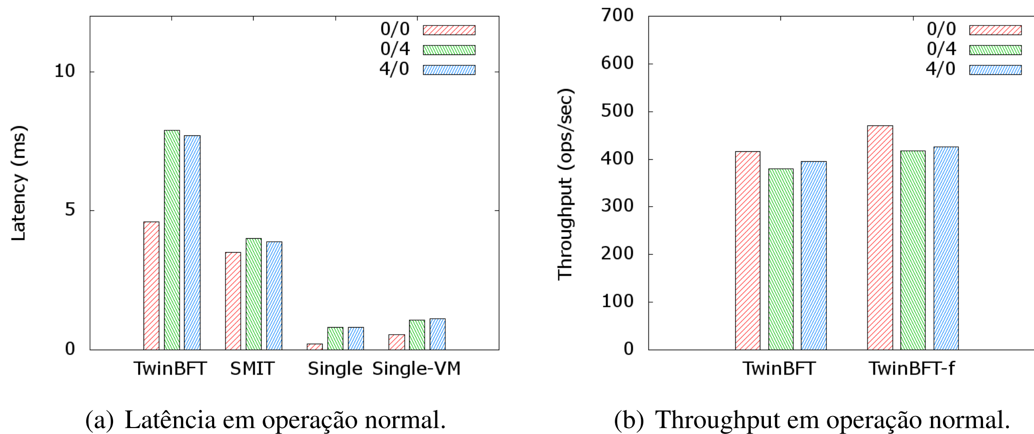
Quando qualquer processo *p* recebe uma mensagem "NEW-VIEW" do novo primário, verifica se: (1) a mensagem está devidamente assinada, (2) contém um conjunto *V* com $f+1$ mensagens "VIEW-CHANGE" válidas. Se as condições forem satisfeitas, reexecuta todas as requisições contidas em *P* para a nova visão (linhas 11-14).

5. Implementação e Resultados

Para avaliar a performance da abordagem, foi escolhido um método experimental. O algoritmo foi implementado em Java, de acordo com as especificações da versão 1.6. Os canais de comunicação foram feitos pela utilização de *channels* do Java NIO, usando TCP com MACs (Códigos de Autenticação de Mensagem).

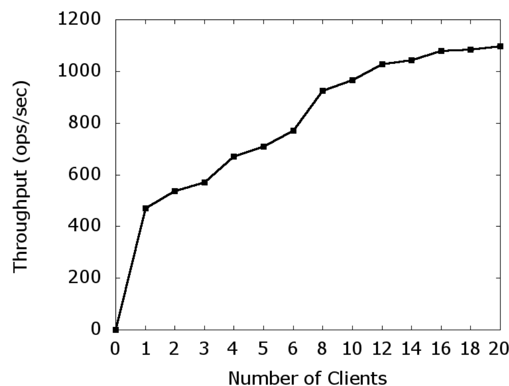
Executamos nossos experimentos em três servidores Intel®Core™i7 3.8Ghz com Debian 7.0 "wheezy" (Kernel 3.2.0 x86-64) e VMM Xen Hypervisor 4.1.3. Cada máquina virtual foi configurada com 2GB de memória e duas CPUs virtuais, equipadas com SUN's JDK 1.6.0_29.

Como medida de avaliação, foi adotado latência e *throughput*, por serem largamente utilizadas neste tipo de avaliação e porque permitem uma verificação simplificada da eficiência do sistema [Jain 1991]. Os resultados foram obtidos através de *microbenchmarks* em diferentes condições de carga. A latência foi obtida a partir de algumas requisições com um único cliente enviando uma requisição por vez, e o *throughput* medindo quantas requisições o sistema consegue responder em uma unidade de tempo. O sistema foi avaliado a partir de *microbenchmarks* para que o custo fosse avaliado sem a influência do serviço. Para estes *microbenchmarks*, foi utilizado um serviço *stateless* com uma operação nula, variando o tamanho das requisições e respostas entre 0KB e 4KB.



(a) Latência em operação normal.

(b) Throughput em operação normal.



(c) Throughput com múltiplos clientes.

Figura 3. Desempenho verificado para o *TwinBFT* em operação normal.

A fim de avaliar o desempenho da solução proposta, o algoritmo foi executado em condições, em que foram enviadas 10.000 requisições de um único cliente, em três

diferentes cargas: 0/0, 0/4 e 4/0. Eles representam, respectivamente, uma requisição e resposta nula, uma requisição nula e uma resposta de 4KB, e uma requisição de 4KB e resposta nula. Todos os tempos foram medidos pelo cliente, a partir da leitura de seu relógio local antes do envio da requisição e após o recebimento de uma resposta válida.

Na Figura 3(a), são mostradas as diferentes latências em cada carga. Para obter a latência, 10.000 requisições foram enviadas e executadas sequencialmente, de forma individual, sendo a latência o tempo médio destas requisições. A abordagem é comparada com o algoritmo SMIT [Stumm et al. 2010], que contém certas semelhanças pela utilização de máquinas virtuais e memória compartilhada, porém não suporta faltas de *crash*. *Single* se refere a uma implementação do serviço centralizado, sem máquinas virtuais enquanto *Single-VM* representa uma execução centralizada dentro do isolamento de uma máquina virtual. É esperado que a abordagem proposta apresente uma avaliação pior do que versões centralizadas pois estas não são tolerantes a faltas.

O *throughput*, como mostrado na Figura 3(b) foi calculado baseado no tempo total para execução das 10.000 requisições enviadas simultaneamente para o serviço. No primeiro grupo, são mostradas as medidas para o serviço em seu caso normal, sem falhas. Em TwinBFT-f mostramos o *throughput* do algoritmo em caso de faltas, sendo que em 1% das requisições a réplica primária se mostra faltosa, gerando uma troca de visão para manter a consistência. Para uma comparação do desempenho em diferentes cargas do algoritmo, a Figura 3(c) apresenta o *throughput* quando temos mais de um cliente fazendo requisições simultaneamente. O número de requisições por segundo se eleva com o aumento de clientes simultâneos até se estabilizar em torno de 1000 operações por segundo quando passa a ficar mais limitado pela capacidade das réplicas do que pela capacidade dos clientes.

Na Tabela 1 é mostrada uma comparação entre a abordagem proposta e outros algoritmos BFT na literatura. Todos os número consideram apenas execuções sem faltas. Os benefícios no uso de máquinas virtuais gêmeas são visíveis no número de réplicas e passos de comunicação. Enquanto nossa abordagem tem o menor número de réplicas, juntamente com [Correia et al. 2004, Chun et al. 2007, Veronese et al. 2013], ela tem o mesmo número de passos de comunicação de algoritmos especulativos [Kotla et al. 2008, Veronese et al. 2013], mesmo em caso de faltas. Algoritmos especulativos, entretanto requerem um número maior de passos em casos com faltas, além de envolverem o cliente no protocolo, perdendo transparência.

Tabela 1. Comparação entre as propriedades de protocolos BFT.

Protocolos Avaliados	*Propriedades e Características*				
	Número de réplicas	**Número de processos**	**Número de máquinas físicas**	**Passos de comunicação**	**Especulativo/ Otimista**
PBFT [Castro and Liskov 1999]	$3f+1$	$3f+1$	$3f+1$	5	não
Zyzzyva [Kotla et al. 2008]	$3f+1$	$3f+1$	$3f+1$	3 / 5[1]	sim
BFT-TO [Correia et al. 2004]	$2f+1$	$2f+1$	$2f+1$	5	não
A2M-PBFT-EA [Chun et al. 2007]	$2f+1$	$2f+1$	$2f+1$	5	não
MinBFT [Veronese et al. 2013]	$2f+1$	$2f+1$	$2f+1$	4	não
MinZyzzyva [Veronese et al. 2013]	$2f+1$	$2f+1$	$2f+1$	3 / 5[1]	sim
TwinBFT	$2f+1$	$4f+2$	$2f+1$	3	não

Como a abordagem proposta utiliza duas máquinas virtuais em cada réplica, possui um número maior de processos, apesar do número de máquinas físicas ser o mesmo de [Correia et al. 2004, Chun et al. 2007, Veronese et al. 2013].

[1]Necessita de dois passos adicionais para confirmar a requisição no caso de suspeita de falta.

6. Conclusões

A partir da exploração de algumas técnicas de virtualização, foi possível a proposta de um algoritmo BFT alternativo. Foi mostrado que é possível implementar um algoritmo RME confiável com $2f + 1$ réplicas físicas em um ambiente assíncrono. Apesar de necessitar de um canal de comunicação confiável para a comunicação entre as máquinas virtuais, acreditamos que a virtualização é vastamente disponível atualmente e pode fornecer um isolamento entre as réplicas e o mundo exterior. Além do mais, foi possível reduzir também o número de passos de comunicação, reduzindo o custo desta comunicação.

Referências

Bessani, A., Daidone, A., Gashi, I., Obelheiro, R. R., Sousa, P., and Stankovic, V. (2009). Enhancing fault / intrusion tolerance through design and configuration diversity. In *Proceedings of the 3rd Workshop on Recent Advances on Intrusion-Tolerant Systems*.

Castro, M. and Liskov, B. (1999). Practical Byzantine fault tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, pages 173–186.

Chandra, T. D. and Toueg, S. (1996). Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267.

Chun, B.-G., Maniatis, P., Shenker, S., and Kubiatowicz, J. (2007). Attested append-only memory: making adversaries stick to their word. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, pages 189–204.

Correia, M., Neves, N. F., and Verissimo, P. (2004). How to tolerate half less one Byzantine nodes in practical distributed systems. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems*, pages 174–183.

Cowling, J., Myers, D., Liskov, B., Rodrigues, R., and Shrira, L. (2006). HQ-Replication: A hybrid quorum protocol for Byzantine fault tolerance. In *Proceedings of 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 177–190.

Distler, T., Popov, I., Schröder-Preikschat, W., Reiser, H. P., and Kapitza, R. (2011). SPARE: Replicas on hold. In *Proceedings of the 18th Network and Distributed System Security Symposium*, pages 407–420.

Doudou, A., Garbinato, B., Guerraoui, R., and Schiper, A. (1999). Muteness failure detectors: Specification and implementation. In *Proceedings of the Third European Dependable Computing Conference on Dependable Computing*, pages 71–87. Springer-Verlag.

Garcia, M., Bessani, A., Gashi, I., Neves, N., and Obelheiro, R. (2011). OS diversity for intrusion tolerance: Myth or reality? In *Proceedings of the IEEE/IFIP 41st International Conference on Dependable Systems and Networks*, pages 383 –394.

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*.

Gashi, I., Popov, P. T., and Strigini, L. (2007). Fault tolerance via diversity for off-the-shelf products: A study with SQL database servers. *IEEE Transactions on Dependable and Secure Computing*, 4(4):280–294.

Inayat, Q. and Ezhilchelvan, P. (2006). A performance study on the signal-on-fail approach to imposing total order in the streets of byzantium. In *In Proceedings of the 36th International Conference on Dependable Systems and Networks*, pages 578–587.

Jain, R. K. (1991). *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley & Sons.

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

Kihlstrom, K. P., Moser, L. E., and Melliar-Smith, P. M. . (2003). Byzantine fault detectors for solving consensus. *The Computer Journal*, 46.

Kotla, R., Clement, A., Wong, E., Alvisi, L., and Dahlin, M. (2008). Zyzzyva: speculative Byzantine fault tolerance. *Commun. ACM*, 51:86–95.

Lamport, L., Shostak, R., and Pease, M. (1982). The Byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401.

Laureano, M., Maziero, C., and Jamhour, E. (2004). Intrusion detection in virtual machine environments. In *Proceedings of the 30th Euromicro Conference*, pages 520–525.

Mpoeleng, D., Ezhilchelvan, P., and Speirs, N. (2003). From crash tolerance to authenticated Byzantine tolerance: A structured approach, the cost and benefits. In *Proceedings of the IEEE/IFIP 33rd International Conference on Dependable Systems and Networks*, pages 227–236.

Murray, D. G., Milos, G., and Hand, S. (2008). Improving Xen security through disaggregation. In *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 151–160.

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

Stumm, V., Lung, L. C., Correia, M., da Silva Fraga, J., and Lau, J. (2010). Intrusion tolerant services through virtualization: A shared memory approach. In *Proceedings of the 24th IEEE International Conference on Advanced Information Networking and Applications*, pages 768 –774.

Szefer, J., Keller, E., Lee, R. B., and Rexford, J. (2011). Eliminating the hypervisor attack surface for a more secure cloud. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, pages 401–412.

Tsudik, G. (1992). Message authentication with one-way hash functions. *SIGCOMM Comput. Commun. Rev.*, 22(5):29–38.

Veronese, G. S., Correia, M., Bessani, A. N., C., L., and Verissimo, P. (2013). Efficient Byzantine fault tolerance. *IEEE Transactions on Computers*, 62(1):16–30.

Wang, Z. and Jiang, X. (2010). HyperSafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Proceedings of the IEEE Security and Privacy Symposium*, pages 380–395.

Wood, T., Singh, R., Venkataramani, A., Shenoy, P., and Cecchet, E. (2011). ZZ and the art of practical BFT execution. In *Proceedings of the 6th ACM SIGOPS/EuroSys European Systems Conference*, pages 123–138.

Yin, J., Martin, J.-P., Venkataramani, A., Alvisi, L., and Dahlin, M. (2003). Separating agreement from execution for Byzantine fault tolerant services. *SIGOPS Oper. Syst. Rev.*, 37:253–267.