Replicação de Máquinas de Estado Usando Detectores de Intrusão e Virtualização

Paulo Henrique de Morais, Lau Cheuk Lung,
Luciana de Oliveira Rech, Fernando Dettoni
Depto de Informática e Estatística - Universidade Federal de Santa Catarina
Florianópolis - SC - Brasil
{paulohenrique, lau.lung, luciana.rech, fdettoni}@inf.ufsc.br

Miguel Correia
INESC-ID
Instituto Superior Técnico
Universidade de Lisboa - Lisboa - Portugal
miguel.p.correia@tecnico.ulisboa.pt

***Resumo*—Este artigo propõe uma arquitetura tolerante a faltas bizantinas através de replicação de máquina de estados. O modelo utiliza a tecnologia de virtualização e um detector de intrusão confiável baseado na técnica de introspecção de máquina virtual. Esta abordagem garante o funcionamento correto do sistema, mesmo na presença de réplicas faltosas. As principais contribuições são um número reduzido de réplicas do sistema de $3f + 1$ para $2f + 1$ e o protocolo funciona com três passos de comunicação em situação livre de falha.**

***Abstract*—This paper proposes an intrusion-tolerant architecture using state machine replication. This architecture uses virtualization technologies and a trusted intrusion detector using virtual machine introspection. This approach ensures the correctness of the system, with in presence of malicious faults. The main contribution of this work is about reducing the number of replicas required to tolerate f faults from $3f + 1$ to $2f + 1$.**

I. Introdução

Com a crescente complexidade em sistemas computacionais e sua, cada vez maior, presença no dia-a-dia da sociedade, torna-se importante que estes sistemas apresentem confiabilidade suficiente, tanto para executar corretamente a especificação desejada quanto para apresentar disponibilidade e não se deixar interferir por fatores externos como agentes maliciosos. Para tanto, pode-se utilizar técnicas de tolerância a faltas que, a partir da utilização de redundância, permitem o contínuo funcionamento de sistemas mesmo na presença de faltas.

Dentre uma imensa variedade de técnicas tolerantes a faltas existentes atualmente, destaca-se o modelo de replicação de máquinas de estados (RME) [1]. Este modelo serve de base para muitas outras técnicas (e.g., [2], [3], [4], [5], [6], [7]) e consiste basicamente da utilização de máquinas de estados determinísticas replicadas executando o mesmo serviço.

A primeira abordagem com fins práticos na literatura foi o PBFT [2]. O PBFT é tolerante a faltas bizantinas/arbitrárias, essas faltas pertencem a uma categoria de faltas que podem ser de qualquer tipo. Na área de tolerância a intrusões os termos intrusão e falta bizantina são usados normalmente como sinônimos [8]. Apesar de prático, o PBFT exige a utilização de $3f+1$ réplicas para que se possa mascarar até f faltas. Muitas abordagens futuras foram motivadas em oferecer alternativas com menor custo. Essa redução normalmente se dá a partir do relaxamento das premissas ou utilização de algum componente inviolável adicional.

O detector confiável de intrusões, baseado nas técnicas de introspecção de máquina virtual (VMI - *Virtual Machine Introspection*), serve para fornecer um indicativo sobre a integridade (não corrupção dos códigos executáveis) do sistema operacional e aplicação de um servidor replicado. Caso uma réplica seja indicada como comprometida, esta é automaticamente descartada pelo protocolo. Com isso. é possível reduzir o número de réplicas necessárias para garantir um processamento correto das requisições de $3f + 1$ para $2f + 1$. Este trabalho apresenta um algoritmo para replicação de máquina de estados tolerante a faltas bizantinas. A proposta utiliza técnicas de virtualização para isolar a execução do serviço replicado do detector de intrusões.

O detector tolerante a faltas bizantinas (BFT) proposto pode agregar diferentes técnicas utilizando VMI para detectar intrusões em uma réplica monitorada. Dessa forma, foi possível desenvolver uma arquitetura BFT que pode ser usada por várias aplicações, sendo que cada aplicação pode utilizá-la para tolerar faltas de acordo com as suas necessidades.

As principais contribuições deste trabalho são: reduzir o número de réplicas do sistema de $3f + 1$ para $2f + 1$ e apresentar um modelo que utiliza um detector confiável, o qual pode unir diferentes técnicas VMI. Até onde sabemos, é a primeira vez que um detector confiável é utilizado em uma abordagem de replicação de máquina de estados tolerante a faltas bizantinas.

II. Trabalhos Relacionados

Diversas abordagens oferecem tolerância a intrusões a partir da técnica de replicação de máquina de estados (RME). O PBFT foi o primeiro considerado prático com este objetivo [2]. Utiliza uma arquitetura com $3f + 1$ réplicas necessárias para tolerar até f faltas, portanto apesar de prática é considerada uma abordagem bastante custosa. São executados 5 passos para o processamento de uma única requisição.

Para diminuir o número de passos do protocolo BFT, foi proposto o algoritmo Zyzzyva especulativo [4]. Assumindo otimisticamente que todas as réplicas são corretas, é capaz de reduzir o número de passos de comunicação de 5, no PBFT, para 3. Esta redução, entretanto, não pode ser mantida caso haja alguma suspeita de réplicas faltosas. Neste caso, as requisições devem ser executadas novamente ao custo de 5 passos de comunicação.

A utilização de componentes confiáveis para a execução de certas tarefas gera a possibilidade de redução no número de réplicas necessárias de $3f+1$ para $2f+1$. No trabalho BFT-TO é usado um componente chamado *Trusted Timely Computiong Base (TTCB)* que fornece um serviço de ordenação confiável de forma a permitir uma arquitetura com $2f + 1$ réplicas [9]. Outro componente confiável, chamado *Attested Append-Only Memory (A2M)*, fornece a abstração de um log confiável em que é impossível para um host se comportar de maneiras diferentes para diferentes réplicas e, desta forma, também diminuiu o total de réplicas necessárias para $2f + 1$ [5].

Os algoritmos MinBFT e MinZyzzyva foram propostos utilizando-se um componente confiável que provê um identificador único para cada mensagem. Com isso, foi possível reduzir o número de réplicas para $2f + 1$ e o número de passos para 4 no MinBFT. O MinZyzzyva apresenta uma versão especulativa do MinBFT e diminui a quantidade de passos necessários em situações sem faltas para 3 [6].

Utilizando o conceito de máquinas virtuais gêmeas, o TwinBFT reduziu o número de réplicas necessárias para $2f+1$ e a quantidade de passos necessários para 3, sem a utilização de técnicas especulativas [7]. Esta redução, entretanto, se dá pela utilização de pelo menos duas máquinas virtuais em cada réplica executando o mesmo serviço. Neste caso, cada máquina virtual atua como um detector de faltas para as outras, localizadas na mesma réplica.

Trabalhos foram propostos utilizando a tecnologia de virtualização para realizar o monitoramento com o intuito de detectar atividades maliciosas nas máquinas virtuais do sistema. Esses trabalhos se baseiam nos recursos do monitor de máquina virtual (VMM - *Virtual Machine Monitor*) para isolar o detector do sistema monitorado na máquina virtual, garantindo a segurança do detector contra ataques para enganar o programa de detecção/monitoramento.

Na abordagem proposta por Garfinkel e Rosenblum [10] é apresentada uma nova arquitetura utilizando virtualização para construir sistemas de detecção de intrusão (IDSs - *Intrusion Detection System*) que fornece boa visibilidade do estado do host monitorado, e ainda fornece um forte isolamento para o IDS. Em [11] é apresentado um sistema baseado em virtualização chamado *VMscope*. Esse sistema exibe os eventos internos de sistema de *honeypots* baseados em máquina virtual (VM - *Virtual Machine*) externamente aos *honeypots*, através da observação e interpretação de eventos de chamadas de sistema internas da VM na camada do VMM.

Outra proposta, chamada *VSyscall*, utiliza tecnologia de virtualização para habilitar intervenção de chamada de sistema externa ao sistema operacional [12]. Para garantir a exatidão e segurança do sistema *VSyscall*, a interceptação e análise da chamada de sistema são realizadas na camada do VMM, enquanto que as chamadas de sistema são invocadas por programas no SO convidado. No artigo [13] é apresentado um conjunto de ferramentas de introspecção virtual desenvolvida para Xen (VIX tools) para análise forense digital discreto dos dados do sistema volátil em máquinas virtuais.

Xu et al. [14] propõem um *framework* de controle de uso para proteger a integridade do *guest kernel*. Neste *framework* o monitor da máquina virtual é estendido para utilizar políticas de segurança, as quais garantem que as solicitações de acesso dos processos da máquina virtual somente serão permitidas caso não estejam acessando objetos sensíveis do *guest kernel*.

Este trabalho utiliza a tecnologia de virtualização para implementar replicação de máquina de estados tolerante a faltas bizantinas. Muitos benefícios são obtidos através desta abordagem, tais como: a redução de número de réplicas para $2f + 1$ e a redução nos passos de comunicação necessários. Além disso, o sistema BFT pode incorporar novos tipos de detecção referente à introspecção de máquina virtual para detectar uma réplica faltosa/maliciosa.

III. Virtualização e Introspecção de Máquina Virtual

A virtualização oferece muitos benefícios incluindo a capacidade de emular *hardware* ou expor várias interfaces de hardware virtuais em um único *host* físico. A camada de *software* que expõe essa interface de *hardware* é referida como um monitor de máquina virtual (VMM) ou hipervisor e o *software* que é executado na máquina virtual é chamado de convidado [15].

Aplicações de segurança podem ser suportadas através de virtualização usando mecanismos de introspecção de máquina virtual (VMI). Essa técnica foi definida por Garfinkel e Rosenblum [10] como uma abordagem para examinar, monitorar e manipular o estado de uma máquina virtual de um local externo com o propósito de analisar o software executado na VM.

A virtualização fornece três propriedades que possibilitam VMI [10]. (1) **Isolamento**: o *software* que está sendo executado em uma máquina virtual não pode acessar ou modificar o *software* que está sendo executado no VMM ou em uma VM separada. Este isolamento garante que mesmo se um atacante tem o controle total sobre o *host* monitorado, ele não pode interferir no sistema VMI. (2) **Verificação**: o VMM possui acesso a todos os estados de uma máquina virtual: estado da CPU (por exemplo, registradores), memória, e todos os dispositivos I/O, tais como o conteúdo de dispositivos de armazenamento e estado do registrador dos controladores I/O. Ser capaz de examinar diretamente a máquina virtual faz com que atividades maliciosas sejam difíceis de serem escondidas de um sistema VMI desde que não há estado no sistema monitorado que o sistema VMI não possa ver. (3) **Interferência**: o VMM precisa interferir sobre certas operações da máquina virtual (por exemplo, execução de instruções privilegiadas). Um sistema VMI pode utilizar esta funcionalidade para o seu próprio propósito. Por exemplo, com apenas uma modificação mínima no VMM, um sistema VMI pode ser notificado se o código executado na VM tenta modificar um determinado registrador.

Para que uma aplicação VMI possa realizar qualquer tipo de introspecção do estado do sistema operacional (SO) convidado, o componente de introspecção deve ter algum conhecimento semântico sobre o estado do convidado. Isto é, a aplicação VMI deve entender o que representa o estado binário que será analisado. Sem um conhecimento adicional, é difícil determinar quais instruções ou estrutura de dados o componente de introspecção está verificando quando é dado simplesmente uma informação binária. Esta dissociação entre

os dados binários e seu significado é chamado de espaço semântico [16].

Para preencher o espaço semântico dois métodos são usados [15]. (1) **Informações recebidas a priori**: estas informações fornecem ao componente de introspecção o conhecimento necessário para interpretar o estado binário do convidado e executar introspecção. Por exemplo, o VMM pode usar uma tabela de símbolos do *kernel* do SO convidado fornecida anteriormente para determinar a posição das principais estruturas de dados. (2) **Derivado**: este método utiliza o *hardware* virtual para derivar informações de um SO simplesmente pelo conhecimento da arquitetura de hardware no qual ele executa. O VMM pode extrair informações de um SO convidado por monitorar os registradores da CPU.

Aplicações VMI podem monitorar e/ou interferir no estado de uma VM. Um mecanismo de segurança usando VMI para monitorar um sistema pode somente detectar e relatar problemas, enquanto que VMI utilizada para interferir pode responder para uma ameaça detectada. Este último, por exemplo, pode encerrar os processos, a própria VM, ou reduzir os recursos disponíveis da VM para inviabilizar os atacantes de acessá-los [17].

IV. Modelo de Sistema

Na figura 1 é apresentado o modelo proposto. Nesse modelo as réplicas são representadas pelos processos executados nas máquinas virtuais (servidores) que efetuam o serviço solicitado pelo cliente. O número total de máquinas físicas ou *hosts* $H = \{h_1, h_2, ..., h_n\}$ é igual a $n >= 2f + 1$ e f é o número de servidores que podem falhar. Cada máquina física tem uma correspondente máquina virtual, na qual executa um processo ou réplica do sistema e um detector de intrusão para verificar a integridade da máquina virtual.

O detector é composto por três módulos. (1) **Módulo VMI** contém as implementações de detecção responsáveis por responder se a réplica está faltosa/maliciosa. Esse módulo analisa as informações do servidor monitorado e define se o estado do servidor está correto ou corrompido. (2) **Módulo MP**: O mecanismo de política (MP) é utilizado para definir as regras específicas de segurança de acordo com o objetivo do detector. O módulo MP serve para indicar as informações que o detector deve considerar para realizar a análise do estado do servidor monitorado, por exemplo, em uma verificação de integridade de sistema de arquivos, quais seriam os arquivos monitorados, arquivos somente leitura e os arquivos de leitura/escrita. (3) **Módulo BD**: A base de dados (BD) pode ser usada para vários propósitos, como registrar os resultados da verificação do módulo do detector de intrusão para serem analisados posteriormente ou armazenar *hashes* de arquivos que podem ser utilizados para verificar integridade de arquivos e/ou garantir execução segura de código.

Os papéis assumidos pelas réplicas do sistema BFT podem ser: (1) réplica líder, sendo responsável por definir uma ordem para a mensagem recebida do cliente; (2) réplica *backup* (seguidora), que executa a mensagem na ordem proposta pelo líder. A réplica líder também executa a própria mensagem ordenada. Se a réplica líder é faltosa, uma das réplicas *backups* corretas será escolhida para ser o novo líder.

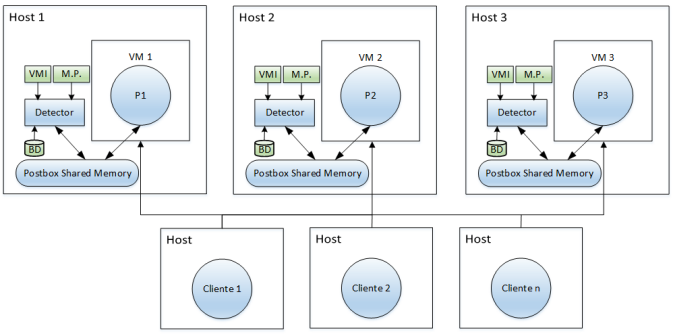

Figura 1. Modelo de Sistema.

Para realizar a comunicação entre o detector e a máquina virtual, o sistema *host* compartilhará localmente um espaço de memória para sua correspondente máquina virtual, chamada *postbox*, similar a outras propostas na literatura [7], [18]. A comunicação entre as réplicas em diferentes hosts é feita através da rede, apenas por troca de mensagens.

O detector de intrusão é um componente confiável implementado no sistema *host*. Essa premissa se baseia no isolamento que o VMM fornece entre o detector e a máquina virtual e através do isolamento da introspecção de máquina virtual realizada pelo detector. Para impossibilitar que um atacante tenha acesso ao código executado pelo detector, o sistema *host* é isolado também em relação a rede WAN (*Wide Area Network*). Isto pode ser feito através de técnicas de *firewall* e/ou desabilitando/removendo os *drivers* de rede do sistema operacional do *host*, bloqueando o acesso para o endereço de rede do *host* e mantendo o endereço IP da máquina virtual acessível. Dessa forma, o atacante nem mesmo tem ciência da existência do *host* [18].

O detector, diretamente, é apenas tolerante a faltas de *crash*, ao passo que as réplicas são tolerantes a faltas bizantinas relacionadas ao tipos de detecção implementada. O detector realiza a verificação de integridade da sua correspondente máquina virtual. Diversas formas de verificar a integridade de um servidor/sistema monitorado foram propostas na literatura, tais como, examinar a integridade do sistema de arquivos em disco ([19], [20]), verificar a integridade dos códigos executáveis na memória principal ([10], [21]), interceptação de chamadas de sistema [12].

V. Algoritmo VmiBFT

O algoritmo proposto utiliza o modelo RME [1] para tolerar faltas bizantinas. Dessa forma cada réplica é uma máquina de estados. Para que cada réplica tenha a mesma sequência de estados, as seguintes propriedades devem ser satisfeitas [8]:

- **Estado inicial**: todos os servidores começam no mesmo estado.
- **Acordo**: todos os servidores executam as mesmas operações.
- **Ordem total**: todos os servidores executam as operações na mesma ordem.
- **Determinismo**: A mesma operação executada no mesmo estado inicial gera o mesmo estado final.

Para que todas as réplicas iniciem no mesmo estado, é considerada uma configuração em que as máquinas virtuais

estão executando o algoritmo BFT pela primeira vez. O acordo é realizado pelo líder, o qual define uma ordem para cada mensagem recebida. Para garantir a ordem total, todas as réplicas seguem a ordem definida pelo líder se a mensagem assinada pelo detector informa que o líder está correto. O determinismo é alcançado, considerando que cada réplica execute uma sequência de instruções equivalentes de cada operação.

O algoritmo proposto se alterna em uma sequência de configurações, chamadas de visão. Em cada visão, somente uma réplica é líder e as outras são réplicas *backups*. Quando as réplicas *backups* suspeitam que o líder é malicioso, uma troca de visão é realizada para que se defina uma nova réplica correta para assumir como líder.

A. Algoritmo: caso normal de operação

Os passos do protocolo são apresentados na Figura 2. O número de passos é igual a 3 em execução normal. Considerando um sistema em que $f = 1$, o número de máquinas físicas é igual a 3 ($2f + 1$).

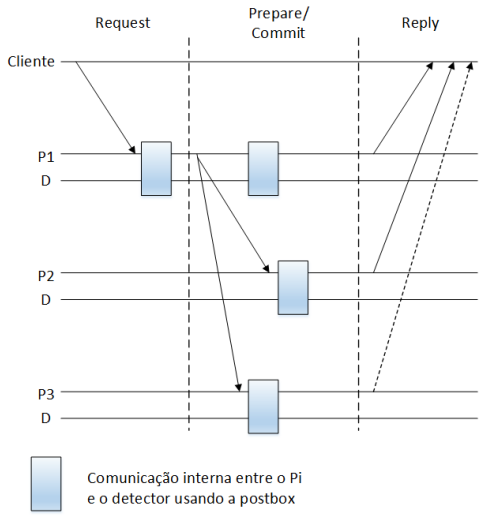

Figura 2. Passos do protocolo em execução normal com $f = 1$.

O algoritmo 1 é enumerado a seguir:

1) Cliente envia a requisição para a réplica primária (linha 1).
2) O líder define uma ordem para a requisição recebida ser executada e insere na *postbox* uma mensagem ORDER identificada por p_i contendo a requisição original msg, o número de sequência proposto n, a visão atual v e dm o resumo da mensagem msg. Em seguida aguarda o detector assinar a requisição ordenada (linhas 3-9).
3) O líder envia para todas as réplicas *backups* a mensagem ORDER assinada pelo detector de intrusão (linha 10).
4) Assim que cada processo, inclusive o processo na réplica primária, recebe a mensagem ORDER, a operação requisitada pelo cliente é executada na ordem definida e a resposta REPLY (em que seq é o número de sequência do cliente identificado por c e res é a resposta da requisição solicitada) é inserida na *postbox* (linhas 13-16).
5) Cada processo aguarda o seu correspondente detector assinar a resposta (linhas 17-19).
6) Cada processo envia a resposta para o cliente (linha 20) e ao receber ao menos $f+1$ respostas válidas[1] de diferentes réplicas, aceita a resposta.

Algorithm 1: Algoritmo BFT em caso normal de operação

```
1  while true do
2      msg ← receive();
3      if received(REQUEST) then
4          if é o líder then
5              n ← n + 1;
6              postbox.append(<< ORDER, p_i, v, n, dm> , msg>);
7              repeat
8                  signedOrder ← getSignedOrder();
9              until receber a mensagem ordenada assinada pelo
                   detector;
10             multicast(<<ORDER, p_i, v, n, dm,
                   replicaCompremetida>k_DetPrivate, msg>);
11         end
12     end
13     else if received(ORDER) then
14         if a order assinada pelo detector informa que o líder está
               correto then
15             reply ← execute(signedOrder);
16             postbox.append(<REPLY, p_i, v, seq, c, res>);
17             repeat
18                 signedReply ← getSignedReply();
19             until receber a resposta assinada pelo detector;
20             reply_to_client(<REPLY, p_i, v, seq, c, res,
                   replicaComprometida>_kDetPrivate);
21         end
22     end
23 end
```

As mensagens assinadas pelo detector são geradas através de assinatura digital [22]. Cada réplica tem o conhecimento da chave pública do seu próprio detector e também da chave pública correspondente do detector de outras réplicas. Os clientes também possuem as chaves públicas de cada réplica do sistema BFT, e de seus detectores. As instruções executadas pelo detector no algoritmo 2 são:

1) O detector de intrusão lê a mensagem da *postbox* e verifica a integridade da máquina virtual (linhas 2-4).
2) Se a mensagem é ORDER, então concatena com a mensagem recebida uma assinatura realizada com sua chave privada informando se a máquina virtual está ou não comprometida (linhas 5-7).
3) Se a mensagem é REPLY, então concatena com a mensagem recebida uma assinatura realizada com sua chave privada informando se a máquina virtual está ou não comprometida (linhas 8-10).

Algorithm 2: Algoritmo executado pelo Detector

```
1  while true do
2      msg ← postbox.read();
3      // próxima linha retorna um valor booleano
4      replicaComprometida ← verifiesIntegrityOfVM();
5      if received(ORDER) then
6          postbox.append(<<ORDER, p_i, v, n, dm,
               replicaComprometida>_kPrivate, msg>);
7      end
8      else if received(REPLY) then
9          postbox.append(<REPLY, p_i, v, seq, c, res,
               replicaComprometida>_kPrivate)
10     end
11 end
```

[1]Respostas assinadas corretamente pelo detector.

B. Protocolo de troca de visão

O protocolo de troca de visão é executado quando $f+1$ réplicas *backups* suspeitam da réplica líder como faltosa. A troca de visão se inicia quando a réplica backup recebe uma requisição diretamente do cliente e verifica que esta requisição não foi repassada previamente pelo líder. Escolher um novo líder é essencial para garantir continuidade em relação aos pedidos que o sistema BFT recebe dos clientes.

O algoritmo de troca de visão 3 funciona da seguinte maneira:

1) Cada réplica ao receber $f+1$ mensagens VIEW_CHANGE válidas, verifica se é o novo líder e em caso positivo insere uma mensagem NEW_VIEW na sua *postbox*, sendo que msg.v é o número da nova visão (linhas 2-11).
2) O novo líder aguarda o detector assinar a mensagem NEW_VIEW e depois envia a mensagem NEW_VIEW assinada para as réplicas *backups* (linhas 12-15).
3) As réplicas *backups* ao receber a mensagem válida NEW_VIEW do novo líder aceitam a réplica como novo líder (linhas 20-24).

Algorithm 3: Algoritmo troca de visão

```
1  while true do
2      msg ← receive();
3      if received(VIEW_CHANGE) then
4          if mensagem é válida then
5              // C = conjunto de mensagens
                  VIEW_CHANGE
6              C ← C ∪ msg;
7              // |C| = número de mensagens VIEW_CHANGE
8              if (|C| ≥ f + 1) then
9                  // |R| = número de réplicas do
                      sistema
10                 if i = msg.v mod |R| then
11                     postbox.append(<NEW_VIEW, p_i, msg.v>);
12                     repeat
13                         signedNewView ←
                           getSignedNewView();
14                     until receber a mensagem de nova visão
                       assinada pelo detector;
15                     multicast(<NEW_VIEW, p_i, msg.v,
                       replicaComprometida>k_kDetPrivate);
16                 end
17             end
18         end
19     end
20     else if received(NEW_VIEW) then
21         if novo líder está correto then
22             v ← msg.v;
23         end
24     end
25 end
```

VI. Implementação e Resultados

Essa seção apresenta resultados de uma avaliação experimental da arquitetura proposta. A linguagem de programação Java foi utilizada para fazer a implementação do protótipo. O detector de intrusão utilizado é uma aplicação executada em cima do SO da máquina física. Este componente verifica a integridade de arquivos em disco de um programa binário. A implementação considerou os arquivos compilados do algoritmo BFT para verificar sua integridade, porém outros arquivos poderiam ser adicionados caso fosse necessário e outras formas de verificação de integridade poderiam ser incluídas pelo detector, como aquelas mostradas na seção IV. Para realizar e verificar a integridade dos arquivos, o detector baseou-se numa base de dados contendo os *hashes* dos arquivos monitorados. A comunicação entre o detector e a máquina virtual é feita utilizando um espaço de memória compartilhado (arquivos) entre esses processos pela VMM.

A máquina física utilizava um processador Intel®Core™i5 @ 2.50 GHz x 4 com Ubuntu 12.04 LTS e 4 GB de memória. Cada máquina virtual rodava sobre o SO Ubuntu 14.04 LTS e utilizando o VMware®Player 5.0.1 como VMM, uma CPU virtual e 1 GB de memória. As máquinas virtuais estavam conectadas em uma rede Ethernet 10/100.

Para realizar a avaliação, foi calculada a latência de diferentes cargas com tamanho entre 0/4 KB sendo que o primeiro valor é o tamanho da requisição e o segundo a resposta, como mostra a figura 3. As cargas foram geradas através de mensagens (pedido/resposta) com tamanho de 0/4 KB. Foi utilizado um serviço *stateless* com uma operação nula para que a latência fosse avaliada sem a influência do serviço. A latência de cada carga foi obtida pela média de 10.000 requisições enviadas por um cliente. O desempenho da solução proposta mostra que mesmo com o *overhead* causado pelo uso da virtualização, o algoritmo vmiBFT é viável para ser usado.

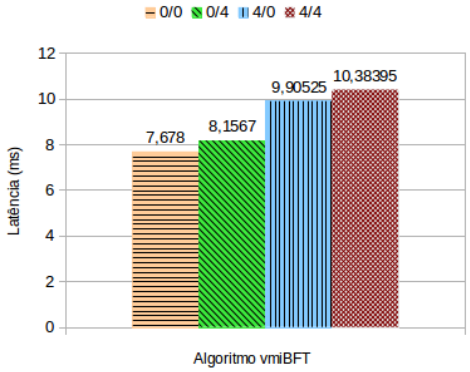

Figura 3. Latência em operação normal.

Também foi calculado, com base na informação da coluna número de mensagens da tabela I, o número de trocas de mensagens necessárias para executar uma operação no caso normal, conforme é apresentado na figura 4. Os algoritmos vmiBFT e BFT-TO possuem o menor número de mensagens. E como pode ser analisado, para cada valor de f sucessivo, o número de trocas de mensagens adicionais é sempre 4. Enquanto que no PBFT em cada configuração consecutiva o número de mensagens acrescentadas é cada vez maior.

Uma comparação é apresentada na tabela I entre o algoritmo BFT proposto e outros algoritmos BFT de trabalhos relacionados. Alguns trabalhos utilizaram um componente confiável para reduzir o número de réplicas do sistema, como mostra a última coluna. A proposta além de contribuir para que utilize o menor número de réplicas, mensagens e passos de comunicação, utiliza também o menor número de processos comparado com os trabalhos anteriores relacionados a replicação de máquina de estados.

VII. Conclusões

Este trabalho apresentou um modelo e um algoritmo de tolerância a faltas bizantinas através de replicação de máquina

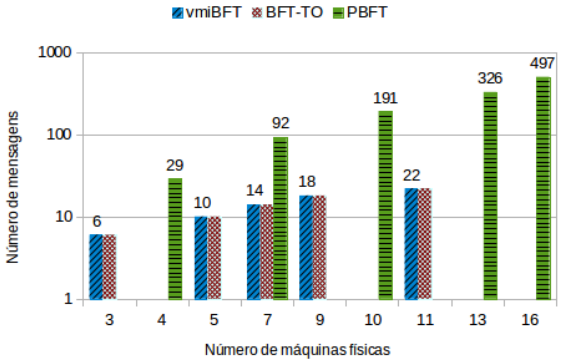

Figura 4. Número de trocas de mensagens, para f = 1,2,...,5.

Tabela I. COMPARAÇÃO ENTRE AS PROPRIEDADES DE PROTOCOLOS BFT

Protocolos Avaliados	Núm. de rép./máq. físicas	Núm. de processos	Passos de comunicação	Núm. de mensagens	Comp. confiável
PBFT [2]	$3f+1$	$3f+1$	5	$2n^2-n+1$	não
Zyzzyva [4]	$3f+1$	$3f+1$	3 / 5 [1]	$(2n)/(4n)$	não
BFT-T0 [9]	$2f+1$	$2f+1$	5	$2n$	sim
A2M-PBFT-EA [5]	$2f+1$	$2f+1$	5	$2n^2-n+1$	sim
MinBFT [6]	$2f+1$	$2f+1$	4	n^2+2n-1	sim
MinZyzzyva [6]	$2f+1$	$2f+1$	3 / 5	$(3n-1)/(5n-1)$ [1]	sim
TwinBFT [7]	$2f+1$	$4f+2$	3	$5n-2$	não
VmiBFT	$2f+1$	$2f+1$	3	$2n$	sim

[1] Caso de suspeita de falta.

de estados. A proposta faz uso da virtualização e de um componente confiável identificado como detector no modelo BFT. Deste modo, foi possível reduzir para $2f+1$ o número de réplicas necessárias para tolerar f réplicas faltosas. Além disso, o número de passos de comunicação é menor comparado com o PBFT [2].

O detector baseado na técnica de introspecção de máquina virtual é fundamental no modelo proposto, pois VMI garante o isolamento do detector e permite monitorar/analisar uma aplicação externamente à máquina virtual que executa o sistema monitorado. Dessa forma, o detector se torna um componente inviolável e essencial para a corretude do sistema BFT como um todo.

Para validar o modelo BFT, foi proposta uma abordagem experimental utilizando a verificação de integridade de arquivos em disco como um tipo de detecção para ser usada na arquitetura. Porém, outros tipos de detecção baseada em VMI podem ser usadas em conjunto pelo detector, construindo dessa forma um sistema BFT mais robusto.

Agradecimentos: Paulo Morais e Miguel Correia são bolsistas CAPES/Brasil.

REFERÊNCIAS

[1] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: a tutorial," *ACM Comput. Surv.*, vol. 22, no. 4, pp. 299–319, Dec. 1990.

[2] M. Castro and B. Liskov, "Practical Byzantine fault tolerance," in *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, 1999, pp. 173–186.

[3] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin, "Separating agreement from execution for Byzantine fault tolerant services," *SIGOPS Oper. Syst. Rev.*, vol. 37, pp. 253–267, October 2003.

[4] R. Kotla, A. Clement, E. Wong, L. Alvisi, and M. Dahlin, "Zyzzyva: speculative Byzantine fault tolerance," *Commun. ACM*, vol. 51, pp. 86–95, 2008.

[5] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz, "Attested append-only memory: making adversaries stick to their word," in *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, October 2007, pp. 189–204.

[6] G. S. Veronese, M. Correia, A. N. Bessani, L. C., and P. Verissimo, "Efficient Byzantine fault tolerance," *IEEE Transactions on Computers*, vol. 62, no. 1, pp. 16–30, 2013.

[7] F. Dettoni, L. C. Lung, M. Correia, and A. Luiz, "Byzantine fault-tolerant state machine replication with twin virtual machines," in *Computers and Communications (ISCC), 2013 IEEE Symposium on*, July 2013, pp. 000 398–000 403.

[8] M. Correia, "Serviços distribuídos tolerantes a intrusões: resultados recentes e problemas abertos," in *SBSeg 2005, V Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais, Livro Texto dos Minicursos*, L. P. Gaspary and F. Siqueira, Eds. Sociedade Brasileira de Computação, 2005, pp. 113–162.

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

[10] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *In Proc. Network and Distributed Systems Security Symposium*, 2003, pp. 191–206.

[11] X. Jiang and X. Wang, "Out-of-the-box monitoring of vm-based high-interaction honeypots," in *Proceedings of the 10th International Conference on Recent Advances in Intrusion Detection*, ser. RAID'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 198–218.

[12] B. Li, J. Li, T. Wo, C. Hu, and L. Zhong, "A vmm-based system call interposition framework for program monitoring," in *Proceedings of the 2010 IEEE 16th International Conference on Parallel and Distributed Systems*, ser. ICPADS '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 706–711.

[13] B. Hay and K. Nance, "Forensics examination of volatile system data using virtual introspection," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 3, pp. 74–82, Apr. 2008.

[14] M. Xu, X. Jiang, R. Sandhu, and X. Zhang, "Towards a vmm-based usage control framework for os kernel integrity protection," in *Proceedings of the 12th ACM Symposium on Access Control Models and Technologies*, ser. SACMAT '07. New York, NY, USA: ACM, 2007, pp. 71–80.

[15] J. Pfoh, "Leveraging derivative virtual machine introspection methods for security applications," Ph.D. dissertation, Technische Universität München, 2013, doctoral Thesis.

[16] P. Chen and B. Noble, "When virtual is better than real [operating system relocation to virtual machines]," in *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on*, May 2001, pp. 133–138.

[17] K. Nance, M. Bishop, and B. Hay, "Virtual machine introspection: Observation or interference?" *IEEE Security and Privacy*, vol. 6, no. 5, pp. 32–37, Sep. 2008.

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

[19] G. H. Kim and E. H. Spafford, "The design and implementation of tripwire: A file system integrity checker," in *Proceedings of the 2Nd ACM Conference on Computer and Communications Security*, ser. CCS '94. New York, NY, USA: ACM, 1994, pp. 18–29.

[20] A. Hay, D. Cid, and R. Bray, *OSSEC Host-Based Intrusion Detection Guide*. Syngress Publishing, 2008.

[21] L. Litty and D. Lie, "Manitou: A layer-below approach to fighting malware," in *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*. ACM, 2006.

[22] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, no. 2, pp. 120–126, Feb. 1978.