UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Paulo Henrique de Morais

TOLERÂNCIA A FALTAS BIZANTINAS USANDO TÉCNICAS DE INTROSPECÇÃO DE MÁQUINAS VIRTUAIS

Florianópolis

2015

Paulo Henrique de Morais

TOLERÂNCIA A FALTAS BIZANTINAS USANDO TÉCNICAS DE INTROSPECÇÃO DE MÁQUINAS VIRTUAIS

Dissertação submetida ao Programa de Pós-Graduação em Ciência da Computação para a obtenção do Grau de Mestre em Ciência da Computação. Orientadora: Prof[a]. Luciana de Oliveira Rech, Dr[a].

Florianópolis

2015

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Morais, Paulo Henrique de
Tolerância a Faltas Bizantinas usando Técnicas de Introspecção de Máquinas Virtuais / Paulo Henrique de Morais ; orientadora, Luciana de Oliveira Rech ; coorientador, Lau Cheuk Lung. - Florianópolis, SC, 2015.
118 p.

Dissertação (mestrado) - Universidade Federal de Santa Catarina, Centro Tecnológico. Programa de Pós-Graduação em Ciência da Computação.

Inclui referências

1. Ciência da Computação. 2. Tolerância a Faltas Bizantinas. 3. Replicação de Máquina de Estados. 4. Virtualização. 5. Introspecção de Máquinas Virtuais. I. Rech, Luciana de Oliveira. II. Lung, Lau Cheuk . III. Universidade Federal de Santa Catarina. Programa de Pós Graduação em Ciência da Computação. IV. Título.

Paulo Henrique de Morais

TOLERÂNCIA A FALTAS BIZANTINAS USANDO TÉCNICAS DE INTROSPECÇÃO DE MÁQUINAS VIRTUAIS

Esta dissertação foi julgada adequada para obtenção do título de mestre e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

Florianópolis, 23 de setembro de 2015.

_____
Prof. Ronaldo dos Santos Mello, Dr.
Coordenador do Programa

Banca Examinadora:

_____
Profª. Luciana de Oliveira Rech, Drª.
Universidade Federal de Santa Catarina
Orientadora

_____
Prof. Carlos Alberto Maziero, Dr.
Universidade Federal do Paraná

_____
Prof. Rafael Rodrigues Obelheiro, Dr.
Universidade do Estado de Santa Catarina

_____
Prof. Frank Augusto Siqueira, Dr.
Universidade Federal de Santa Catarina

Dedico este trabalho à minha família. Aos meus pais Nivaldo e Marli, às minhas irmãs Daniely e Sylvia.

AGRADECIMENTOS

Agradeço aos meus orientadores professora Luciana de Oliveira Rech e professor Lau Cheuk Lung pela direção que me deram para realizar este trabalho. Agradeço aos meus colegas e amigos do LaPeSD os quais sempre estavam dispostos a ajudar quando precisei, não poderia deixar de fazer um agradecimento especial ao colega Fernando Dettoni.

Agradeço a todas as pessoas que participaram direta ou indiretamente na realização deste trabalho de pesquisa.

"Quer você acredite que consiga fazer uma coisa ou não, você está certo."

(Henry Ford)

RESUMO

Atualmente é quase impossível uma pessoa não utilizar direta ou indiretamente um sistema computacional. Ao realizar uma operação bancária ou até mesmo ao fazer compras em uma loja, nós somos auxiliados por sistemas computacionais.

Em contrapartida, surgem novos ataques para comprometer o funcionamento correto dos sistemas utilizados. Várias técnicas são utilizadas para que os sistemas funcionem conforme sua especificação, entre elas, destacam-se sistemas tolerantes a faltas bizantinas/intrusões (BFT) através de replicação de máquina de estados (RME). Nessa abordagem, é proposta uma arquitetura de sistema tolerante a intrusões que garante o seu funcionamento correto, mesmo na presença de réplicas faltosas.

Este trabalho propõe um algoritmo que une replicação de máquina de estados e sistema de detecção de intrusões (IDS) para tolerar faltas bizantinas. A tecnologia de virtualização é utilizada no algoritmo proposto para replicar o serviço e também para isolar o IDS da aplicação monitorada. Dessa forma, é proposto um detector de intrusões como um componente confiável do sistema BFT. As principais contribuições são: (1) propor um modelo unificado, o qual utiliza replicação de máquina de estados e IDS em conjunto, e faz uso dos recursos da tecnologia de virtualização, (2) detector de intrusões como componente confiável do sistema e (3) elaborar um algoritmo BFT baseado no modelo proposto. Através desta abordagem foi possível reduzir o número de réplicas do sistema de $3f + 1$ para $2f + 1$ e reduzir o número de passos do protocolo do algoritmo tradicional BFT de Castro e Liskov de 5 para 3 no caso normal de operação e sem precisar da participação do cliente no protocolo.

Palavras-chave: Tolerância a Faltas Bizantinas; Sistemas Distribuídos; Segurança; Detecção de intrusão; Virtualização.

ABSTRACT

Currently, it is almost impossible for a person not to use a computing system, in a direct or indirect way. When we are using a banking machine, or shopping in a store, we need to use a computing system. On the other hand, there are new attacks to damage the correct working of the systems. There are several techniques to help the systems to work correctly according to their specification; among them, the Byzantine/intrusions fault tolerant systems (BFT) through the state machine replication (SMR) are important ones. In this perspective, it proposes a system architecture tolerant to intrusions that guarantees its proper functioning, even if there are faulty replicas.

This research proposes an algorithm which presents a unified approach by using state machine replication and intrusion detection system in order to tolerate Byzantine faults. The virtualization technology is used on the proposed algorithm to replicate the service and also to isolate the IDS of the monitored application. Therefore, we propose an intrusion detector as a reliable component of the BFT system. The main contributions are: (1) to propose a unified model, which uses state machine replication together with IDS, using the virtualization technology resources; (2) intrusion detector as a reliable component of the system; and (3) to make a BFT algorithm based on the proposed model. This approach made it possible to decrease the number of the system replicas from the $3f + 1$ to $2f + 1$, and to reduce the number of steps of the protocol of the BFT traditional algorithm from Castro and Liskov from 5 to 3 in a normal case of operation without the participation of the client in the protocol.

Keywords: Byzantine fault tolerance; Distributed Systems; Security; Intrusion Detection; Virtualization.

LISTA DE FIGURAS

Figura 1 Serviços replicados entre servidores. 26
Figura 2 Uma memória: máquina de estado (SCHNEIDER, 1990). 26
Figura 3 Classificação de Faltas. 32
Figura 4 Tipo 1 de VMM. 36
Figura 5 Tipo 2 de VMM. 36
Figura 6 Mapeamento de Memória. A visão lógica da perspectiva de (a) um processo, (b) um sistema operacional, (c) um monitor de máquina virtual 37
Figura 7 Arquitetura do VMI IDS 41
Figura 8 Um exemplo de lista de processo (kernel Linux 2.6.x) . . 44
Figura 9 O protocolo PBFT em execução livre de faltas. 48
Figura 10 Arquitetura do sistema, para $f = 1$. 51
Figura 11 Exemplo de um protocolo *atomic multicast* usando o serviço *TO wormhole*. 51
Figura 12 O protocolo Zyzzyva em execução livre de faltas. 53
Figura 13 O protocolo Zyzzyva em execução com falta. 54
Figura 14 O protocolo A2M no caso livre de faltas. Linhas espessas mostram as mensagens que são certificadas usando A2M. 57
Figura 15 O protocolo MinBFT no caso livre de faltas. 59
Figura 16 O protocolo MinZyzzyva. 61
Figura 17 O protocolo TwinBFT no caso livre de faltas. 63
Figura 18 O protocolo ByzID no caso livre de faltas. 65
Figura 19 O protocolo para resolver consenso com detector de faltas não confiável. 68
Figura 20 Interações entre $\mathcal{A}_p$ e $\diamond M_{\mathcal{A}}$. 70
Figura 21 Exemplos de trocas de mensagens: Servidor B é (a)*correct*, (b)*detectably faulty*, (c)*detectably ignorant.* (HAEBERLEN; KOUZNETSOV; DRUSCHEL, 2006) 73
Figura 22 Comunicação entre aplicação, protocolo, e módulo do detector no servidor i. (HAEBERLEN; KOUZNETSOV; DRUSCHEL, 2006) 74
Figura 23 Modelo de Sistema para o VmiBFT. 81
Figura 24 Localizações do detector confiável. 84

Figura 25 Passos do protocolo em execução normal com $f = 1$.... 87
Figura 26 Comunicação interna entre p_i e o detector usando a *post-box*. 89
Figura 27 Latência em operação normal. 98
Figura 28 Histograma da latência do algoritmo vmiBFT (*overhead full*). 98
Figura 29 Sobrecarga dos módulos do algoritmo proposto. 99
Figura 30 Modelo sem virtualização. 99
Figura 31 Latência em operação normal. 100
Figura 32 Histograma da latência do algoritmo vmiBFT (*overhead full*) - cenário 2 100
Figura 33 Sobrecarga da virtualização. 101
Figura 34 Número de trocas de mensagens, para f = 1,2,...,5.... 105

LISTA DE TABELAS

Tabela 1 Classes de detectores de faltas bizantinas 72
Tabela 2 Comparação entre as propriedades de protocolos relacionados BFT 78
Tabela 3 Implementações realizadas utilizando o algoritmo Vmi-BFT 97
Tabela 4 Comparação entre as propriedades de protocolos BFT . 102

LISTA DE ABREVIATURAS E SIGLAS

PBFT	Practical Byzantine Fault Tolerance	21
BFT	Byzantine Fault Tolerance	21
PKI	Public Key Infrastructure	22
SCADA	Supervisory Control and Data Acquisition	22
RME	Replicação de Máquina de Estados	22
VMI	Virtual Machine Introspection	22
VM	Virtual Machine	22
SMA	State Machine Approach	25
MTBF	Mean Time Between Failures	28
VMM	Virtual Machine Monitor	34
VIX	Virtual Introspection for Xen	44
MAC	Message Authentication Code	47
USIG	Unique Sequential Identifier Generator	58
IDS	Intrusion Detection System	64
NIDS	Network Intrusion Detection Systems	83
HIDS	Host Intrusion Detection Systems	83

SUMÁRIO

1 INTRODUÇÃO 21
1.1 MOTIVAÇÃO 21
1.2 PROPOSTA DO TRABALHO 22
1.3 OBJETIVOS 23
1.3.1 Objetivo Geral 23
1.3.2 Objetivos Específicos 23
1.4 CONTRIBUIÇÕES 23
1.5 ORGANIZAÇÃO DO TEXTO 24
2 FUNDAMENTAÇÃO TEÓRICA 25
2.1 CONCEITOS BÁSICOS SOBRE REPLICAÇÃO DE MÁQUINA DE ESTADOS E TOLERÂNCIA A FALTAS 25
2.1.1 Máquina de Estados 25
2.1.2 Tolerância a faltas 27
2.1.3 Replicação de Máquina de estados tolerante a faltas 28
2.1.3.1 Acordo 30
2.1.3.2 Ordem e Estabilidade 31
2.1.4 Classificação de Faltas 31
2.1.5 Difusão de Mensagens 33
2.1.5.1 Difusão Confiável 33
2.1.5.2 Difusão com Ordem Total 34
2.2 CONCEITOS BÁSICOS SOBRE VIRTUALIZAÇÃO 34
2.2.1 VMMs e Sistemas de Segurança 37
2.2.2 Introspecção de Máquina Virtual (VMI) 38
2.2.2.1 Espaço Semântico 41
2.2.2.2 Acesso à Memória 41
2.2.2.3 Interceptação de chamada de sistema 42
2.2.2.4 Introspecção Virtual para Xen (VIX) 43
2.3 CONSIDERAÇÕES FINAIS 44
3 TRABALHOS RELACIONADOS 47
3.1 ABORDAGENS BASEADAS NA TÉCNICA RME 47
3.1.1 Practical Byzantine Fault Tolerance - PBFT 47
3.1.2 BFT-TO: Intrusion Tolerance with Less Replicas ... 50
3.1.3 Zyzzyva: Speculative Byzantine Fault Tolerance ... 52
3.1.4 Attested Append-Only Memory: Making Adversaries Stick to their Word - A2M 55
3.1.5 Efficient Byzantine Fault-Tolerance - MinBFT e MinZyzzyva 57

3.1.5.1 MinBFT 58
3.1.5.2 MinZyzzyva 60
3.1.6 Byzantine fault-tolerant state machine replication with twin virtual machines - TwinBFT 61
3.1.7 ByzID: Byzantine Fault Tolerance from Intrusion Detection 64
3.2 ABORDAGENS BASEADAS EM DETECTORES DE FALTAS 66
3.2.1 Unreliable Intrusion Detection in Distributed Computations 66
3.2.2 Muteness Failure Detectors: Specification and Implementation 68
3.2.3 Byzantine Fault Detectors for Solving Consensus . . . 70
3.2.4 The Case for Byzantine Fault Detection 72
3.2.5 Enhanced Fault-Tolerance through Byzantine Failure Detection 75
3.3 CONSIDERAÇÕES FINAIS 77
4 TOLERÂNCIA A FALTAS BIZANTINAS USANDO TÉCNICAS DE INTROSPECÇÃO DE MÁQUINAS VIRTUAIS 79
4.1 MODELO 80
4.2 DETECTOR CONFIÁVEL 82
4.3 *POSTBOX*: MEMÓRIA COMPARTILHADA 84
4.4 PROTOCOLO PROPOSTO 85
4.4.1 Algoritmo VmiBFT 86
4.4.2 Algoritmo: caso normal de operação 86
4.4.3 Protocolo de troca de visão 89
4.5 CORREÇÃO DO ALGORITMO VMIBFT 91
4.6 CONSIDERAÇÕES FINAIS 93
5 RESULTADOS E ANÁLISES 95
5.1 CENÁRIO 1 95
5.2 CENÁRIO 2 99
5.3 COMPARAÇÕES DE NOSSAS PROPOSIÇÕES COM TRABALHOS RELACIONADOS 101
5.4 CONSIDERAÇÕES FINAIS 105
6 CONCLUSÕES E PERSPECTIVAS FUTURAS 107
REFERÊNCIAS 109

1 INTRODUÇÃO

1.1 MOTIVAÇÃO

Com a crescente complexidade dos sistemas computacionais e sua presença cada vez maior 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 sofrer interferência de fatores externos, como agentes maliciosos.

O funcionamento correto de sistemas de acordo com suas especificações é essencial, principalmente para sistemas considerados críticos, onde grandes prejuízos financeiros, ou até mesmo perda de vidas, podem acontecer se houver alguma falha em sua execução. 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) (SCHNEIDER, 1990). Este modelo serve de base para muitas outras técnicas (e.g., (CASTRO; LISKOV, 1999; YIN et al., 2003; KOTLA et al., 2007; CHUN et al., 2007; VERONESE et al., 2013; DETTONI et al., 2013; CORREIA; VERONESE; LUNG, 2010; LUIZ; LUNG; CORREIA, 2011)) e consiste basicamente da utilização de máquinas de estados determinísticas replicadas executando o mesmo serviço.

A primeira abordagem RME com fins práticos descrita na literatura foi o *Practical Byzantine Fault Tolerance* - PBFT (CASTRO; LISKOV, 1999). O PBFT é tolerante a faltas bizantinas/arbitrárias, ou seja, 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 (CORREIA, 2005). Apesar de prático, o PBFT exige a utilização de $3f + 1$ réplicas para que se possa mascarar até f faltas. Muitas abordagens posteriores 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.

Sistemas tolerantes a faltas bizantinas (BFT - *Byzantine Fault Tolerance*) podem ser usados para garantir a segurança de serviços críticos (CORREIA, 2005; FAVARIM et al., 2007; BESSANI; FRAGA;

LUNG, 2006; LUIZ; LUNG; CORREIA, 2014; PRESSER; LUNG; CORREIA, 2015), tais como: infra-estrutura de chaves públicas (PKI - *Public Key Infrastructure*), comércio eletrônico, sistemas de supervisão e aquisição de dados (SCADA - *Supervisory Control and Data Acquisition*), banco de dados distribuídos, sistemas de arquivos distribuídos e serviços de autorização.

1.2 PROPOSTA DO TRABALHO

Um novo algoritmo tolerante a faltas bizantinas, chamado VmiBFT, é apresentado neste trabalho. O algoritmo proposto utiliza a abordagem de replicação de máquina de estados (RME) e faz uso da tecnologia de virtualização. O algoritmo VmiBFT trabalha em conjunto com um detector de intrusões que é um componente confiável do sistema.

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 do sistema operacional e aplicação de um servidor replicado. Através das técnicas de VMI é possível analisar o *software* executado na máquina virtual (VM) de um lugar externo à VM. Caso uma réplica seja indicada como comprometida, esta pode ser 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 $2f + 1$.

O detector apresentado no modelo 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.

O modelo de sistema apresentado utiliza um conjunto de $2f + 1$ máquinas físicas, sendo que cada *host* tem uma única máquina virtual, na qual o serviço é replicado. A proposta do trabalho faz uso dos benefícios fornecidos pela virtualização. Um dos recursos desta tecnologia é o isolamento entre máquina virtual e o *host*. Desse modo, apenas a máquina virtual é exposta para que os clientes solicitem seus pedidos e as aplicações executadas no sistema *host*, como exemplo o sistema de detecção de intrusões, têm sua segurança garantida por meio do isolamento fornecido pelo monitor de máquinas virtuais.

1.3 OBJETIVOS

1.3.1 Objetivo Geral

Elaborar um algoritmo de replicação de máquina de estados tolerante a faltas bizantinas que utiliza detecção de intrusão e virtualização para reduzir o número de réplicas necessárias do sistema de $3f+1$ para $2f+1$, e com menos passos de comunicação.

1.3.2 Objetivos Específicos

Para alcançar o objetivo geral, foram determinados os seguintes objetivos específicos:

- Levantamento bibliográfico para conhecer o estado da arte na área de tolerância a faltas bizantinas através de replicação de máquina de estados.
- Propor um modelo de sistema, o qual será utilizado pelo algoritmo tolerante a faltas bizantinas.
- Propor um detector como componente confiável do sistema através da tecnologia de virtualização.
- Avaliar o algoritmo proposto através de experimentos práticos.

1.4 CONTRIBUIÇÕES

As principais contribuições deste trabalho são:

1. Propor um modelo unificado, o qual utiliza replicação de máquina de estados e IDS em conjunto, e faz uso dos recursos da tecnologia de virtualização.
2. Apresentar um detector de intrusões como componente confiável do sistema, o qual utiliza as técnicas de introspecção de máquina virtual para verificar a integridade da réplica monitorada.
3. Elaborar um algoritmo BFT baseado no modelo proposto.
4. Através da utilização do IDS confiável e da tecnologia de virtualização foi possível a redução do número de réplicas do sistema

de $3f+1$ para $2f+1$ e reduzir do número de passos do algoritmo tradicional BFT de Castro e Liskov de 5 para 3 no caso normal de operação.

1.5 ORGANIZAÇÃO DO TEXTO

Neste capítulo foram descritas as motivações para o desenvolvimento da proposta e os objetivos desta dissertação. O restante deste documento está organizado da seguinte maneira:

- **Capítulo 2 - Fundamentação teórica:** Os conceitos sobre a abordagem replicação de máquina de estados (RME), tolerância a faltas, tecnologia de virtualização e outros assuntos relevantes são explicados no decorrer deste capítulo, para melhor compreensão do trabalho proposto.
- **Capítulo 3 - Trabalhos relacionados:** descreve os trabalhos correlacionados à proposta desenvolvida, apresentando uma revisão da literatura e o estado da arte no qual a proposta está inserida.
- **Capítulo 4 - Tolerância a Faltas Bizantinas usando Técnicas de Introspecção de Máquinas Virtuais:** apresenta um novo algoritmo tolerante a faltas bizantinas através da técnica de Replicação de Máquina de Estados (RME), chamado VmiBFT. Neste capítulo também são descritos o detector confiável de intrusões baseado nas técnicas de introspecção de máquina virtual (VMI), o qual interage com o algoritmo proposto e o modelo de sistema que o algoritmo VmiBFT deve utilizar.
- **Capítulo 5 - Resultados e Análises:** apresenta os resultados obtidos de uma avaliação experimental que foi realizada para validação da proposta. Análises são feitas com os dados coletados em dois cenários. O capítulo também mostra uma comparação qualitativa entre o algoritmo proposto e outros algoritmos que são relacionados a este trabalho.
- **Capítulo 6 - Conclusões e Perspectivas Futuras:** retoma os principais pontos que foram vistos ao longo desta dissertação e apresenta alguns trabalhos que podem ser realizados futuramente.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 CONCEITOS BÁSICOS SOBRE REPLICAÇÃO DE MÁQUINA DE ESTADOS E TOLERÂNCIA A FALTAS

2.1.1 Máquina de Estados

A abordagem de máquina de estado (SMA - *state machine approach*) foi primeiramente descrita em (LAMPORT, 1978b) para ambientes nos quais não podem ocorrer falhas. A abordagem (SCHNEIDER, 1982) foi generalizada para lidar com falhas por *crash*; uma classe de falhas que trata falhas por *crash* e falhas bizantinas é apresentada em (LAMPORT, 1978a), e (LAMPORT, 1984) trata falhas bizantinas.

Lamport e Schneider (LAMPORT, 1978b; SCHNEIDER, 1990) apresentaram os conceitos sobre máquina de estados. A abordagem de máquina de estado é um método geral para implementar serviços tolerantes a faltas em sistemas distribuídos.

Uma máquina de estado consiste de variáveis de estados, que definem seu estado, e comandos, os quais modificam seu estado. Cada comando é implementado por um programa determinístico; a execução do comando é atômica com relação aos outros comandos, ou seja, não interfere na execução de outros comandos, e modifica as variáveis de estados e/ou produz alguma saída.

Um cliente da máquina de estado faz um pedido para executar um comando. O pedido indica uma máquina de estado, o comando para ser executado, e contém as informações exigidas pelo comando. A saída do processamento do pedido pode ser para um atuador (por exemplo, em um sistema de controle de processo), para algum dispositivo periférico (por exemplo, um disco ou terminal), ou para clientes aguardando respostas de pedidos anteriores. A figura 1 mostra um exemplo onde clientes fazem as solicitações de pedidos e aguardam suas respostas.

Um exemplo de uma máquina de estado é mostrado na figura 2. A memória de máquina de estado implementa um mapeamento com variação no tempo de localizações para valores. Um comando *read* permite um cliente determinar o valor atual associado com uma localização, e um comando *write* associa um novo valor com uma localização.

Comandos podem ser implementados da seguinte maneira:

- Usando uma coleção de procedimentos que compartilham dados e são invocados por uma chamada de subrotina, como em um

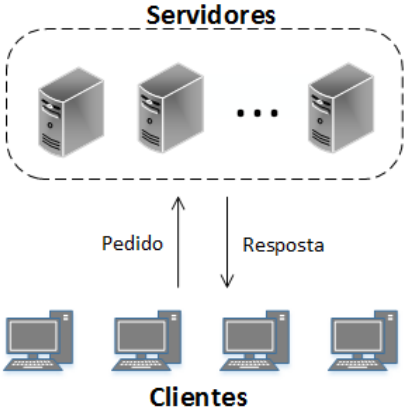

Figura 1: Serviços replicados entre servidores.

```
memory : state_machine
         var store : array [0..n] of word

    read : command(loc : 0..n)
           send store[loc] to client
           end read;

   write : command(loc : 0..n , value : word)
           store[loc] := value
           end write
         end memory
```

Figura 2: Uma memória: máquina de estado (SCHNEIDER, 1990).

monitor.

- Utilizando um único processo que aguarda mensagens contendo pedidos e executa as ações especificadas, como em um servidor.
- Usando um conjunto de controladores de interrupções, caso em que um pedido é feito, causando uma interrupção, como em um *kernel* do sistema operacional.

Pedidos são executados por uma máquina de estado um de cada vez, em uma ordem que é consistente com causalidade potencial, ou seja, o resultado produzido com relação ao pedido executado. Portanto, clientes de uma máquina de estado podem fazer as seguintes hipóteses sobre a ordem em que pedidos são executados:

1. $O1$: Pedidos enviados por único cliente para uma determinada máquina de estado sm são executados pela sm na ordem que foram enviados.

2. $O2$: Se o pedido r feito para uma máquina de estado sm pelo cliente c poderia causar um pedido r' para sm pelo cliente c', então sm executa r antes de r'.

É importante observar que devido a atrasos de comunicações na rede, $O1$ e $O2$ não implicam que uma máquina de estado executará pedidos na ordem estabelecida ou recebida.

A característica que define uma máquina de estado é a especificação de uma computação determinística que lê um fluxo de pedidos e os executa, algumas vezes produzindo saída (SCHNEIDER, 1990):

- Caracterização semântica de uma máquina de estado: saídas de uma máquina de estado são completamente determinadas pela sequência de pedidos executados, independentemente do tempo e de qualquer outra atividade em um sistema.

2.1.2 Tolerância a faltas

Um componente é considerado faltoso quando seu comportamento não é mais consistente com sua especificação (SCHNEIDER, 1990). Duas classes representativas de comportamento faltoso são:

- **Faltas bizantinas**: o componente pode exibir comportamentos maliciosos e arbitrários, talvez envolvendo combinação com outros componentes faltosos (LAMPORT; SHOSTAK; PEASE, 1982).

- **Faltas por parada (*crash*)**: como resposta para uma falta, o componente muda para um estado que permite outros componentes detectarem que uma falta ocorreu. (SCHNEIDER, 1984).

Faltas bizantinas podem ser mais disruptivas e há indícios de que tais faltas ocorrem na prática. Admitir faltas bizantinas é a hipótese mais fraca possível que poderia ser feita sobre os efeitos de uma falta. Dado que um projeto baseado em hipóteses sobre o comportamento de componentes faltosos corre o risco de falhar se essas hipóteses não são satisfeitas, é prudente que sistemas *life-critical* tolerem faltas bizantinas. Entretanto, para a maioria das aplicações, é suficiente assumir faltas por *crash*.

Um sistema que consiste de um conjunto de componentes distintos é tolerante a f faltas se ele satisfaz sua especificação desejada quando não mais do que f desses componentes tornam-se faltosos durante algum intervalo de interesse. Tradicionalmente, tolerância a faltas é especificada em termos de tempo médio entre faltas (MTBF - *mean-time-between-failures*), probabilidade de faltas em um dado intervalo, e outras medidas estatísticas (SIEWIOREK; SWARZ, 1982).

Embora seja claro que tais caracterizações são importantes pelos usuários de um sistema, há vantagens em descrever tolerância a faltas de um sistema em termos do número máximo de faltas de componentes que podem ser toleradas durante algum intervalo de interesse. Durante a execução dos pedidos o número de componentes faltosos não pode exceder o número suportado pelo sistema para garantir os resultados corretos dos pedidos enviados pelos usuários.

Declarar que um sistema é tolerante a faltas torna explícitas as hipóteses necessárias para a operação correta, mas isso não acontece quando se trata de MTBF e outras medidas estatísticas. Além disso, tolerância a faltas não é relacionada com a confiabilidade de componentes que compõem o sistema e logo é uma medida de tolerância a faltas suportada pela arquitetura do sistema a qual define o número máximo de componentes faltosos suportados pelo sistema, ao contrário, quando tolerância a faltas é obtida simplesmente utilizando componentes confiáveis os quais têm seu comportamento correto assumido.

MTBF e outras medidas de confiabilidade estatística de um sistema tolerante a f faltas podem ser derivadas de medidas de confiabilidade estatística pelos componentes utilizados na construção desse sistema, em particular, a probabilidade de que haverá f ou mais faltas durante o intervalo operacional de interesse. Dessa forma, f é tipicamente determinado com base em medidas estatísticas da confiabilidade de componentes.

2.1.3 Replicação de Máquina de estados tolerante a faltas

Uma versão tolerante a f faltas de uma máquina de estado pode ser implementada replicando essa máquina de estado e executando uma réplica em cada um dos processos em um sistema distribuído. Desde que cada réplica seja executada por um processo correto, inicie no mesmo estado inicial e execute os mesmos pedidos na mesma ordem, então cada réplica realizará a mesma coisa e produzirá a mesma saída (SCHNEIDER, 1990).

Assumindo que cada falta pode afetar no máximo um processo, ou seja, uma réplica de máquina de estado, então combinando a saída das réplicas de máquina de estado desse conjunto, pode-se obter a resposta correta para a máquina de estado tolerante a f faltas considerando as respostas fornecidas pelos processos corretos.

Quando processos lidam com faltas bizantinas, um conjunto que implementa uma máquina de estado tolerante a f faltas deve ter pelo menos $2f + 1$ réplicas, e a saída do conjunto é a saída produzida pela maioria das réplicas. Com $2f + 1$ réplicas, a maioria das saídas permanece correta mesmo após f faltas ocorrerem. Se processos lidam com apenas faltas por *crash*, então um conjunto contendo $f + 1$ réplicas é suficiente, e a saída do conjunto pode ser a saída produzida por qualquer réplica desse conjunto. Isso acontece porque somente saídas corretas são produzidas por processos de faltas por *crash*, e após f faltas uma réplica correta permanecerá entre as $f + 1$ réplicas.

Para implementar uma máquina de estado tolerante a f faltas é essencial garantir:

- **Coordenação de réplicas**: todas as réplicas recebem e executam a mesma sequência de pedidos. Isso pode ser decomposto em duas condições referentes à difusão dos pedidos para réplicas em um conjunto:

    1. **Acordo**: toda réplica correta de máquina de estados recebe todos os pedidos.
    2. **Ordem**: toda réplica correta de máquina de estados executa os pedidos recebidos na mesma ordem.

O acordo trata do comportamento de um cliente ao interagir com réplicas de máquina de estado, e a ordem trata o comportamento de uma réplica de máquina de estado em relação aos pedidos de vários clientes. Portanto, ainda que a coordenação de réplicas possa ser particionada em outras formas, a partição acordo-ordem é uma escolha natural porque ela corresponde à separação existente entre cliente e réplicas de máquina de estado.

O acordo pode ser relaxado para pedidos somente-leitura quando processos de faltas por *crash* são assumidos. Nessa situação, um pedido r que não modifica variáveis de estado após sua execução precisa apenas ser enviado para uma única réplica correta de máquina de estado. Isso pode ser feito porque a resposta dessa réplica é, por definição, garantida estar correta e, como r não altera variáveis de estado, o estado da réplica que executa r permanecerá idêntico ao estado das réplicas que não o executam.

Ordem pode ser relaxada para pedidos nos quais a ordem de execução não faz diferença para manter o estado consistente entre as réplicas, ou seja, se a sequência de saídas e o estado final da máquina de estado ao executar r seguido de r' são os mesmos quando executar r' seguido de r. Nesse cenário, não é necessário implementar ordem, pois diferentes réplicas da máquina de estado produzirão as mesmas saídas mesmo se executarem pedidos em ordens diferentes.

2.1.3.1 Acordo

O requisito de acordo pode ser satisfeito utilizando qualquer protocolo que permita a um processo designado, chamado emissor, difundir um valor para outros processos, com as seguintes condições (SCHNEIDER, 1990):

1. Todos os processos corretos concordam com o mesmo valor.
2. Se o emissor é correto, então todos os processos corretos utilizam seu valor para realizar o acordo.

Protocolos para estabelecer as condições 1 e 2 receberam considerável atenção na literatura e são identificados como:

- Protocolos de acordo bizantino.
- Protocolos de difusão confiável.
- Protocolos de acordo.

Protocolos que podem tolerar faltas de processos bizantinos são apresentados em (STRONG; DOLEV, 1983; CASTRO; LISKOV, 1999; CORREIA; NEVES; VERISSIMO, 2004; KOTLA et al., 2007; CHUN et al., 2007; VERONESE et al., 2013; DETTONI et al., 2013; DUAN et al., 2014; CORREIA; VERONESE; LUNG, 2010). Em (SCHNEIDER; GRIES; SCHLICHTING, 1984) é mostrado um protocolo, significativamente mais barato, que pode tolerar faltas de processos apenas por *crash*.

Se pedidos são difundidos para todas as réplicas de máquina de estado usando um protocolo que satisfaz as condições 1 e 2, então o requisito de acordo é satisfeito. O cliente pode ser o emissor ou o cliente pode enviar seu pedido para uma única réplica de máquina de estado e deixar que essa réplica seja o emissor. Quando o cliente não é

o emissor, ele deve garantir que seu pedido não seja perdido ou alterado pelo emissor antes de ser difundido às réplicas de máquina de estado. Uma forma de monitorar a alteração de pedido é ter o cliente incluído entre os processos que recebem o pedido do emissor.

2.1.3.2 Ordem e Estabilidade

O requisito de ordem pode ser satisfeito atribuindo identificadores únicos para os pedidos e tendo as réplicas de máquina de estado executando os pedidos de acordo com uma relação de ordem total baseada nos identificadores únicos que lhes foram atribuídos. Um pedido é definido como sendo estável em uma máquina de estado sm_i quando não existe pedido de um cliente correto com identificador único menor para ser entregue posteriormente para a réplica de máquina de estado sm_i (SCHNEIDER, 1990):

- **Implementação de ordem**: uma réplica executa o pedido estável com menor identificador único.

Refinamento adicional de implementação de ordem requer selecionar um método para atribuir identificadores únicos para os pedidos e elaborar um teste de estabilidade para esse método de atribuição. Qualquer método para atribuir identificadores únicos é restrito pelas hipóteses $O1$ e $O2$ da seção 2.1.1, as quais implicam que se o pedido r_i poderia causar o pedido r_j a ser feito, então $uid(r_i) < uid(r_j)$ mantém, onde $uid(r)$ é o identificador único atribuído para um pedido r.

2.1.4 Classificação de Faltas

A classificação de faltas divide faltas de processos dentro de vários grupos com a propriedade de interesse que uma classe mais forte é um subconjunto de uma classe mais fraca. As classes das mais fortes para mais fracas são: faltas *fail-stop*, faltas de parada, faltas de omissão, faltas por temporização, faltas de valor, e faltas bizantinas (BARBORAK; DAHBURA; MALEK, 1993). A figura 3 mostra as classes de faltas com as relações de subconjunto e superconjunto entre elas.

Uma descrição de cada classe de faltas é feita a seguir:

- **Falta *fail-stop***: a falta que ocorre quando um processo interrompe sua operação e avisa outros processos dessa falta (SCHLICHTING; SCHNEIDER, 1983).

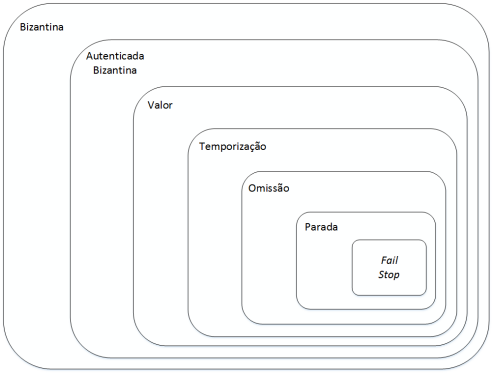

Figura 3: Classificação de Faltas.

- **Falta de parada**: a falta que ocorre quando um processo perde seu estado interno ou interrompe sua operação. Por exemplo, um processo que teve o conteúdo de sua instrução *pipeline* corrompida ou aconteceu um desligamento da energia sofreu uma falta por *crash*.

- **Falta de omissão**: a falta que ocorre quando um processo falha em cumprir um *deadline* ou começar uma tarefa (CRISTIAN et al., 1985). Em particular, uma falta de omissão por envio ocorre quando um processo falha em enviar uma mensagem requerida em um tempo, e uma falta de omissão por receber ocorre quando um processo falha em receber uma mensagem requerida ou comporta-se como se a mensagem não tivesse sido entregue.

- **Falta por temporização**: a falta que ocorre quando um processo completa uma tarefa antes ou depois de seu tempo especificado ou nunca a completa (CRISTIAN et al., 1985). Isso é alguma vezes chamado como falta de desempenho.

- **Falta de valor**: a falta que ocorre quando um processo falha em produzir o resultado correto em resposta a uma entrada correta (LARANJEIRA; MALEK; JENEVEIN, 1991).

- **Falta autenticada bizantina**: uma falta maliciosa ou arbitrária, por exemplo, quando um processo envia mensagens diferentes durante um *broadcast* para seus vizinhos, mas as mensagens alteradas autenticadas podem ser percebidas pelos processos que as

recebem tendo conhecimento da mensagem original (LAMPORT; SHOSTAK; PEASE, 1982).

- **Falta bizantina**: Toda falta possível no modelo de sistema (LAMPORT; SHOSTAK; PEASE, 1982). Essa classe de faltas pode ser considerada como o conjunto universal de faltas.

A classe de faltas de valor é um superconjunto das classes: parada, omissão e temporização e um subconjunto da classe de faltas bizantinas. A primeira característica é verdadeira porque um erro de cálculo pode acontecer no tempo ou espaço. Visto que a falta é consistente para todos observadores externos, embora, a classe de valor é mais restrita do que faltas bizantinas (LARANJEIRA; MALEK; JENEVEIN, 1991).

As classes de faltas mais básicas: parada, omissão e temporização, são problemas que ocorrem no domínio do tempo e são problemas detectáveis no domínio do tempo.

2.1.5 Difusão de Mensagens

Na literatura podem ser encontrados vários tipos de difusão de mensagens, como descritos em (HADZILACOS; TOUEG, 1994). Neste trabalho são enfatizados: a difusão confiável (*reliable multicast*) e a difusão com ordem total (*atomic multicast*).

2.1.5.1 Difusão Confiável

A difusão confiável é definida em termos de duas primitivas: *broadcast* e *deliver*. Quando um processo p invoca *broadcast* com uma mensagem m como parâmetro, é dito que p difunde m. É assumido que m pertence a um conjunto $\mathcal{M}$ de mensagens possíveis. Quando um processo q retorna da execução de *deliver*, é dito que q entrega m.

Como cada processo pode difundir várias mensagens, é importante ser capaz de determinar a identidade do emissor de uma mensagem e distinguir as diferentes mensagens disseminadas por um emissor particular. Deste modo, é assumido que cada mensagem m inclui os seguintes campos: a identidade de seu emissor, indicada por *sender(m)*, e um número de sequência, denotado por *seq#(m)*. Se *sender(m)*=p e *seq#(m)*=i, então m é a i-ésima mensagem difundida por p. Esses campos tornam cada mensagem única.

A difusão confiável satisfaz as seguintes propriedades:

- **Validade**: Se um processo correto difunde uma mensagem m, então algum processo correto terminará por entregar m.
- **Acordo**: Se um processo correto entrega uma mensagem m, então todos processos corretos irão entregar m.
- **Integridade**: Para qualquer mensagem m, todo processo correto entrega m no máximo uma vez, e somente se m foi previamente difundido pelo *sender(m)*.

Validade com acordo garantem que uma mensagem difundida por um processo correto é entregue por todos processos corretos.

2.1.5.2 Difusão com Ordem Total

A difusão com ordem total requer que todos processos corretos entreguem todas mensagens na mesma ordem. Esta ordem total na entrega de mensagens garante que todos processos corretos têm a mesma "visão" do sistema, por isso eles podem agir consistentemente sem qualquer comunicação adicional. A difusão confiável não trata ordenação, uma difusão com ordem total é uma difusão confiável que satisfaz a seguinte condição:

- **Ordem Total**: Se processos corretos p e q entregam mensagens m e m', então p entrega m antes de m' se e somente se q entrega m antes de m'.

Os requisitos de acordo e ordem total da difusão com ordem total implicam que processos corretos acabarão por entregar a mesma sequência de mensagens.

2.2 CONCEITOS BÁSICOS SOBRE VIRTUALIZAÇÃO

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 chamada de monitor de máquina virtual (VMM - *Virtual Machine Monitor*) ou hipervisor, e o *software* que é executado na máquina virtual (VM - *Virtual Machine*) é chamado de convidado (PFOH, 2013).

As duas técnicas de virtualização tradicionais são (SMITH; NAIR, 2005; NANCE; BISHOP; HAY, 2008):

- **Tipo 1**: em um sistema tipo 1 mostrado na figura 4, o VMM executa diretamente no hardware físico, eliminando uma camada de abstração e frequentemente melhorando a eficiência como um resultado. Exemplos de sistemas tipo 1 incluem: VMware ESX (VMWARE, 2015a), Xen (XEN, 2015) e Microsoft Hyper-V (MICROSOFT, 2015a).

- **Tipo 2**: em um sistema tipo 2, o VMM usa um SO como uma interface para o hardware físico (figura 5). Sistemas do tipo 2 incluem: VirtualBox (VIRTUALBOX, 2015), VMware Workstation (VMWARE, 2015b), o emulador de processo de código aberto QEMU (QEMU, 2015), KVM (KVM, 2015), Parallels (PARALLELS, 2015), e Virtual PC/Server (MICROSOFT, 2015b). Sistemas tipo 2 dependem do SO na camada inferior para fornecer interação com o hardware e *drivers* de dispositivos, e assim, muitas vezes têm uma variedade maior de componentes para interagir.

  As distinções entre as duas técnicas de virtualização tradicionais são úteis para compreender o espaço semântico (*semantic gap*) ao observar e interpretar o estado/evento da VM na camada do VMM (JIANG; WANG, 2007).

  A equipe da VMware criou uma técnica chamada tradução binária, porém essa técnica está defasada depois da fabricação de processadores mais novos (*Core 2 Duo* em diante) os quais fornecem um suporte melhor à virtualização. A chave para implementar um VMM de tradução binária é capturar a execução de instruções sensíveis e tratá-las, desde que estas instruções possam interferir no estado do VMM ou SO *host* na camada inferior (ROBIN; IRVINE, 2000). O VMM examinará todas as instruções antes da execução. Quando uma instrução sensível é identificada, ela será forçada para ser capturada no VMM. *INT n* e *iRET* são instruções sensíveis interceptadas no VMM. Quando um processo no SO convidado invoca uma chamada de sistema, a chamada de sistema primeiro é capturada no VMM, onde é inserido um gancho no tratador de interrupções do VMM para interceptar chamadas de sistema; quando terminada, o VMM passará para o tratador de chamada de sistema do SO e permite ao SO tratar a chamada de sistema; quando o SO termina, uma instrução *iRET* será executada, a qual também será capturada pelo VMM; por

último, o VMM retorna o controle para o processo do usuário (LI et al., 2010).

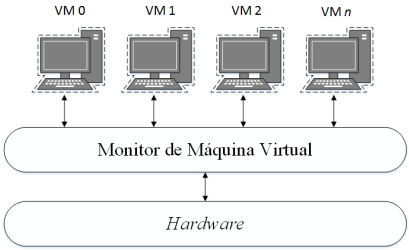

Figura 4: Tipo 1 de VMM.

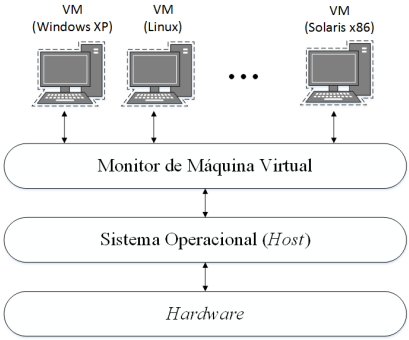

Figura 5: Tipo 2 de VMM.

Para mostrar como virtualização funciona, será analisada uma sequência de eventos que ocorrem quando um processo tenta acessar um endereço de memória em seu espaço de endereço virtual. Sob uma perspectiva de processo, a solicitação resulta no acesso direto para o endereço de memória (Figura 6a). Entretanto, como a figura 6b mostra, ao passo que a camada de SO tem um papel ativo em fornecer acesso de local de memória, na verdade é abstraído do processo, porque ele acessa a tabela de páginas para mapear o endereço de memória lógica para o endereço de memória física (NANCE; BISHOP; HAY, 2008).

Quando a mesma solicitação vem de uma VM, a virtualização acrescenta uma complexidade adicional (6c). Para isolar as VMs que podem executar em um único sistema, o VMM fornece uma camada

de abstração entre cada gerenciamento de memória do SO da VM e o hardware físico subjacente. O VMM então traduz o número de *frame* da página solicitada da VM para um número de *frame* de página para o hardware físico, e dessa forma dá acesso a VM para aquela página (NANCE; BISHOP; HAY, 2008).

Por causa da participação ativa do VMM neste processo e seus privilégios elevados, o VMM pode também acessar diretamente páginas de memória atribuídas para cada VM, sem a VM solicitar a página. O VMM pode fazer aquelas páginas acessíveis para outras VMs no sistema, as quais facilitam o processo de introspecção de máquina virtual (VMI - *Virtual Machine Introspection*) (NANCE; BISHOP; HAY, 2008).

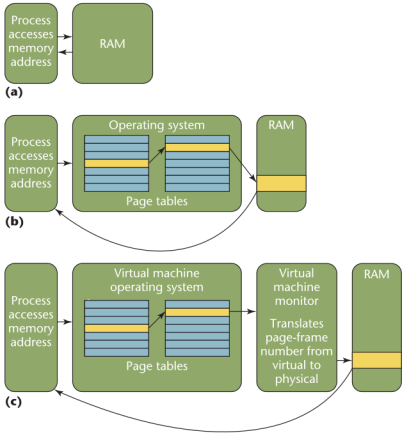

Figura 6: Mapeamento de Memória. A visão lógica da perspectiva de (a) um processo, (b) um sistema operacional, (c) um monitor de máquina virtual

(NANCE; BISHOP; HAY, 2008)

2.2.1 VMMs e Sistemas de Segurança

Aplicações de segurança podem ser suportadas através de virtualização, existem dois tipos de esquemas que combinam VMMs e sistemas de segurança:

- O primeiro executa um sistema de segurança (por exemplo, sistema de detecção de *malware* e *sandboxes*) em um sistema ope-

racional sendo executado em uma VM (SO convidado). Este esquema tem várias vantagens. Primeiro, ele requer nenhuma ou pouca modificação para o código de um VMM ou um SO convidado. Segundo, informação a nível de SO é facilmente disponível para programas executados em um SO convidado. Entretanto, este esquema tem uma desvantagem. Alguns ataques estão cientes de sistemas de segurança. Se o ataque evita de ser detectado pelos sistemas de segurança e obtém com sucesso o privilégio de administrador do SO convidado, o ataque pode parar os sistemas de segurança (ONOUE; OYAMA; YONEZAWA, 2008).

- O outro tipo de esquema é executar um sistema de segurança externo ao SO convidado monitorado. Isto soluciona o problema de ataques cientes de sistemas de segurança. Mesmo se um atacante obtém o privilégio de administrador do SO convidado, o atacante não pode parar o VMM ou o sistema de segurança executado externamente. Por outro lado, este esquema tem uma limitação que sistemas de segurança obtêm somente informações de eventos de baixo nível para interações entre os SOs convidados e o hardware virtual. Não é simples para sistemas de segurança entender e controlar o comportamento de alto nível do SO convidado. Se informação sobre comportamento de alto nível é recuperada de informação sobre eventos de baixo nível, esta limitação é superada (ONOUE; OYAMA; YONEZAWA, 2008).

2.2.2 Introspecção de Máquina Virtual (VMI)

VMI foi definada em (GARFINKEL; ROSENBLUM, 2003; LAUREANO; MAZIERO; JAMHOUR, 2004) como uma abordagem para analisar o software executado em uma VM a partir de um lugar externo à máquina virtual.

Similar à explicação no parágrafo anterior, para (HAY; NANCE, 2008) introspecção virtual é o processo pelo qual o estado de uma máquina virtual (VM) é observado, através do Monitor de Máquina Virtual (VMM), ou de alguma outra máquina virtual que não aquela a ser analisada.

VMI pode ser usada para analisar o estado do sistema dinamicamente, como por exemplo, quais processos estão sendo executados ou quais conexões de rede estão ativas (NANCE; BISHOP; HAY, 2008).

Sistemas VMI podem estar em uma de duas categorias: aqueles que apenas monitoram o comportamento dos processos e aqueles que

interferem no funcionamento dos processos (NANCE; BISHOP; HAY, 2008).

Por exemplo, (GARFINKEL; ROSENBLUM, 2003) propuseram um sistema de detecção de intrusão baseado em *host* que monitora VMs para coletar informações e detectar ataques. Quando o sistema encontra um ataque, ele apenas relata-o, em vez de interferir. O trabalho de (LITTY; LIE, 2006a) utilizou VMI para detectar *malware*. O sistema proposto compara *hashes* conhecidos de páginas de instruções com *hashes* de páginas de memória em tempo de execução. Se não houver correspondência, a página de instrução é considerada corrompida e marcada como não executável (NANCE; BISHOP; HAY, 2008).

A distinção entre monitorar e interferir se assemelha à diferença de segurança entre detecção e resposta. Um mecanismo de segurança usando VMI para monitorar um sistema pode somente detectar e relatar problemas, enquanto que VMI utilizada para interferir pode responder a uma ameaça detectada. Algumas das ações utilizadas para responder ao ataque detectado podem ser: encerrar o processo, encerrar a VM, ou reduzir os recursos disponíveis da VM para inviabilizar os atacantes de acessá-los (NANCE; BISHOP; HAY, 2008).

Aplicações VMI podem ser implementadas em pelo menos dois locais do sistema. A primeira opção é embutir a aplicação VMI no próprio VMM. Isto requer modificar o código do VMM, e pode tornar a aplicação VMI muito dependente da versão do VMM (NANCE; BISHOP; HAY, 2008).

A segunda opção é colocar a aplicação VMI externa ao VMM. Esta é a opção escolhida no trabalho de (NANCE; BISHOP; HAY, 2008) usando Xen (XEN, 2015), o qual coloca a aplicação VMI no domínio privilegiado Dom0. Isto torna as ferramentas menos propensas a mudanças à medida que ocorram alterações no VMM, devido ao fato delas interagirem através de uma API estável. Entretanto, essa opção pode reduzir a capacidade da aplicação de executar processamento *inline* (isto é, reagir às solicitações da VM alvo em tempo real).

Os recursos fundamentais que um VMM deve fornecer a fim de suportar a arquitetura de sistema VMI são:

- **Isolamento**: *Software* que está sendo executado em uma máquina virtual não pode acessar ou modificar o *software* que está sendo executado em um *host* físico ou em uma VM separada. Isolamento garante que mesmo se um atacante tem o controle total sobre o *host* monitorado, ele não pode interferir no sistema VMI (GARFINKEL; ROSENBLUM, 2003).

- **Verificação**: O VMM tem acesso a todos os estados de uma máquina virtual: estado da CPU (por exemplo, registradores), toda memória, e todos os dispositivos E/S como o conteúdo de dispositivos de armazenamento e estado do registrador dos controladores E/S. 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 haja estado no sistema monitorado que o sistema VMI não possa ver (GARFINKEL; ROSENBLUM, 2003).

- **Interferência**: VMMs precisam 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 (GARFINKEL; ROSENBLUM, 2003).

VMMs fornecem isolamento por padrão, entretanto, fornecer verificação e interferência para um sistema VMI requer alguma modificação do VMM (GARFINKEL; ROSENBLUM, 2003).

Um exemplo de sistema VMI para detecção de intrusão (VMI IDS) é apresentado em (GARFINKEL; ROSENBLUM, 2003). O VMI IDS é responsável por implementar políticas de detecção de intrusão analisando o estado e eventos da máquina virtual através da interface do VMM. O VMI IDS é dividido em duas partes, a *OS interface library* e o *policy engine*, como mostrado na figura 7. A tarefa da *OS interface library* é fornecer uma visão a nível de SO do estado da máquina virtual a fim de facilitar a implementação e desenvolvimento da política. O trabalho do *policy engine* é simplesmente executar políticas IDS que estão usando a *OS interface library* e a interface do VMM.

A *OS interface library* interpreta o estado de máquina de baixo-nível do VMM em termos de estruturas do SO em alto nível, através do uso do conhecimento sobre a implementação do SO convidado para interpretar o estado da VM, o qual é exportado pelo VMM. A *OS interface library* deve ser combinada com o SO convidado, e diferentes SOs convidados terão diferentes bibliotecas de interface do SO. O *policy engine* é fornecido com uma interface para fazer consultas em alto nível sobre o SO do *host* monitorado. Este componente interpreta o estado do sistema e eventos da interface do VMM e da *OS interface library*, e decide se o sistema foi comprometido, O *policy engine* é responsável por responder de uma forma adequada. (GARFINKEL; ROSENBLUM,

2003)

No VMI IDS em (GARFINKEL; ROSENBLUM, 2003) foram adicionados ganchos ao VMM para permitir verificação de memória, registradores, e estado do dispositivo. Também foram adicionados ganchos para permitir interferência em determinados eventos, como interrupções e atualizações de dispositivo e estado de memória.

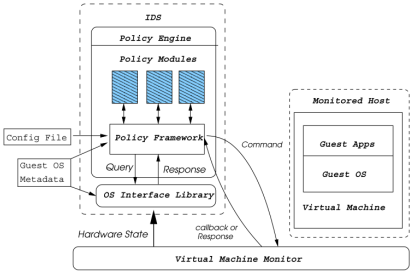

Figura 7: Arquitetura do VMI IDS
(GARFINKEL; ROSENBLUM, 2003)

2.2.2.1 Espaço Semântico

O conceito de espaço semântico (*semantic gap*) é frequentemente abordado em relação à introspecção virtual. O espaço semântico se refere à diferença entre a apresentação dos dados da memória volátil pelo sistema operacional (por exemplo, para processos do espaço do usuário na VM alvo) e o formato bruto pelo qual se pode acessar memória usando técnicas de introspecção virtual. Para as aplicações de introspecção virtual que acessam os dados brutos da memória volátil, isto representa uma desvantagem, em vez de uma limitação fundamental, e o necessário a fazer por essas aplicações é executar a mesma tradução dos dados brutos para informação em alto nível que o sistema operacional já realiza (HAY; NANCE, 2008).

2.2.2.2 Acesso à Memória

Para uma aplicação VMI acessar a memória associada a um processo particular, ela deve identificar o processo individual na VM. Uma aplicação VMI deve fazer isso reconstruindo a lista de processos, então

processar os dados contidos na lista para calcular o local da tabela de páginas de cada processo. A partir daí, a aplicação pode derivar as entradas da tabela de página individual. Neste ponto, a aplicação VMI pode reconstruir a memória associada a cada processo. Determinado isto e a informação da tabela de cada processo, a aplicação VMI pode determinar exatamente o que cada processo estava fazendo. Esta reconstrução deveria ser feita com a VM pausada para que o estado da VM não mude durante a reconstrução (NANCE; BISHOP; HAY, 2008).

Acessar memória (como uma estrutura de dados) em uma típica aplicação é uma tarefa totalmente automatizada e bastante trivial: a aplicação solicita um endereço de memória no espaço de endereço do processo, e o SO traduz de forma transparente o endereço em um frame de página. Entretando, para programadores escreverem aplicações de introspecção virtual o processo é mais complexo. Ao invés de ter o SO mapeando endereços lógicos para endereços físicos, o programa de introspecção deve manualmente traduzir as tabelas de páginas para converter os endereços lógicos em que uma VM acredita ser um endereço físico, mas que é, na verdade, simplesmente outro endereço lógico para o SO subjacente. Isto fornece um frame de página somente no contexto da VM, a qual acredita que o frame tem RAM física contígua. Para acessar a página real de memória física dos dados requeridos, o programa de introspecção deve executar mais uma tradução manual entre o frame de página da VM e os frames de página do *host* físico subjacente (NANCE; BISHOP; HAY, 2008).

Neste ponto, a aplicação VMI tem apenas acesso disponível a uma página de memória que contém os dados solicitados. A aplicação VMI ainda deve transformar os dados brutos em informações úteis para o usuário. E apesar da aplicação VMI estar observando a estrutura de dados, a estrutura é definida na declaração do contexto da VM. Por exemplo, tanto o SO da VM alvo quanto o SO no qual a aplicação VMI é executada, podem inclusive ter SO's diferentes (NANCE; BISHOP; HAY, 2008).

2.2.2.3 Interceptação de chamada de sistema

Para interceptar eventos de chamadas de sistema de uma VM, traduções são realizadas das instruções de chamadas de sistema (por exemplo, interrupção *$0x80* ou *sysenter/sysexit*) que estão sendo invocadas pelos processos internos (JIANG; WANG, 2007).

Além disso, as semânticas implicitamente associadas com estas

instruções de chamadas de sistema são usadas para suas interpretações. Especificamente, sobre a interceptação de um evento de chamada de sistema, o código de interpretação correspondente será executado para entender e coletar as informações associadas de contexto para resolver o espaço semântico (JIANG; WANG, 2007).

Por exemplo, na interceptação de um evento *sys_ execve*, é preciso descobrir o novo processo que será lançado. A resposta está nos argumentos ou no contexto desta chamada de sistema. Especificamente, para a chamada de sistema *sys_ execve* na arquitetura x86 de 32 *bits*, o registrador EBX contém um endereço de memória que aponta para a *string* do nome do arquivo do processo; o registrador ECX tem o endereço de memória de um *array* de *strings* com todos os argumentos de linha de comando (ou seja, argv[]); e o registrador EDX contém o endereço de memória de outro *array* de *strings* com todas as configurações de ambiente (isto é, envp[]). Finalmente é preciso ressaltar que todo endereço de memória mencionado anteriormente é um endereço virtual, o qual é especificado para um processo interno e seria diferente para outros processos. Dessa maneira, sua interpretação exige o percurso da tabela de páginas daquele processo particular executado na VM (JIANG; WANG, 2007).

Na maioria dos sistemas operacionais modernos, processos do usuário que desejam executar operações privilegiadas devem usar a interface de chamada de sistema para acessar serviços do *kernel*. Quando uma chamada de sistema é invocada por um processo a nível de usuário, o processo primeiro coloca os argumentos nos registradores, e então os argumentos são capturados no *kernel* usando uma instrução de interrupção (instrução *int 0x80* para Linux e *INT 2e* para Windows). Plataformas tais como: Linux 2.6, Windows XP e posteriores, normalmente utilizam instrução *sysenter* para serviços de chamada de sistema (LI et al., 2010).

2.2.2.4 Introspecção Virtual para Xen (VIX)

Em Xen, VMs são chamadas de domínios, e o domínio especial de administração é chamado de Dom0 (Figura 4). O VMM concede ao sistema Dom0 acesso a uma biblioteca de controle, que permite ao administrador do sistema criar, destruir, iniciar, pausar, parar, e alocar recursos para VMs utilizando a VM Dom0 (NANCE; BISHOP; HAY, 2008).

Além destas funções administrativas comuns, o sistema Dom0

pode também solicitar páginas de memória alocadas a VMs não privilegiadas para estarem disponíveis ao sistema Dom0. Isto permite a uma aplicação VMI executando no Dom0 examinar a memória de qualquer outra VM no sistema. Essa funcionalidade deve estar disponível apenas para o sistema privilegiado Dom0 para servir de função administrativa (NANCE; BISHOP; HAY, 2008).

A introspecção virtual para Xen (VIX - *Virtual Introspection for Xen*) pausa operação da VM alvo, mapeia parte de sua memória no sistema Dom0, adquire e interpreta os dados relevantes das páginas de memória, e então volta à operação da VM alvo. Como um exemplo, todos os processos correntes em um sistema Linux têm uma estrutura de dados *task_ struct* que armazena ou vincula informações como ID do processo, nome do processo, mapa de memória, e tempo de execução. Uma forma que VIX pode referenciar essas estruturas de dados é percorrer a lista de processos: a lista de *task_ struct* mantida pelo SO (Figura 8) (NANCE; BISHOP; HAY, 2008).

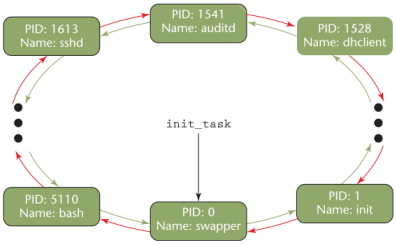

Figura 8: Um exemplo de lista de processo (kernel Linux 2.6.x) (NANCE; BISHOP; HAY, 2008)

Para tornar páginas de memória de uma VM disponíveis para outras partes do sistema é preciso na maioria das vezes fazer uma modificação no VMM. No caso do Xen, existe uma API chamada *libvix* que fornece essa funcionalidade, e dessa forma não precisa ser feita a modificação no VMM (HAY; NANCE, 2008).

2.3 CONSIDERAÇÕES FINAIS

Neste capítulo foram apresentados conceitos básicos os quais são encontrados na literatura e serviram como base para o desenvolvimento desta dissertação. Foi apresentada uma visão geral acerca dos fundamentos teóricos sobre replicação de máquina de estados, tolerância a

faltas, tecnologia de virtualização e introspecção de máquina virtual. Por fim, cabe ressaltar que grande parte dos conceitos discutidos ao longo do capítulo serão úteis para uma melhor compreensão do trabalho proposto.

3 TRABALHOS RELACIONADOS

Este capítulo apresenta os principais trabalhos relacionados, os quais serviram como uma base para propor uma nova abordagem de replicação de máquina de estados para tolerar faltas bizantinas utilizando detecção de intrusões, a qual é intitulada como VmiBFT.

3.1 ABORDAGENS BASEADAS NA TÉCNICA RME

3.1.1 Practical Byzantine Fault Tolerance - PBFT

Trabalhos de tolerância a faltas bizantinas anteriores ao PBFT (CASTRO; LISKOV, 1999) apresentaram técnicas projetadas para demonstrar viabilidade teórica (MALKHI; REITER, 1997; GARAY; MOSES, 1998) que são muito ineficazes para serem usadas na prática ou supuseram sincronia (REITER, 1995; KIHLSTROM; MOSER; MELLIAR-SMITH, 1998), ou seja, se basearam em limites conhecidos de atrasos de mensagens e processamento de pedidos.

Castro e Liskov (CASTRO; LISKOV, 1999) propuseram o primeiro algoritmo prático de replicação de máquina de estado que tolera faltas bizantinas para ser usado em sistemas distribuídos assíncronos. Eles utilizaram um esquema eficiente de autenticação que consiste em usar códigos de autenticação de mensagem (MACs - *Message Authentication Codes*) durante operação normal e criptografia de chave pública apenas quando houver faltas.

O modelo de sistema assume que no máximo f réplicas podem falhar, e um número total de n réplicas, onde $n \geq 3f+1$. Por exemplo, considerando um sistema onde $f = 1$ réplica faltosa para que o sistema continue funcionando conforme sua especificação, o número total de réplicas n é: $n = 3f + 1 = 3 \times 1 + 1 = 4$ réplicas.

A figura 9 mostra a operação do algoritmo PBFT no caso livre de faltas. Os passos do algoritmo são enumerados a seguir:

1. O cliente envia um pedido para o servidor primário.

2. O servidor primário, ao receber o pedido do cliente, lhe atribui um número de sequência e envia através de *multicast* uma mensagem PRE-PREPARE para todas as réplicas *backups*.

3. A réplica *backup*, ao aceitar a mensagem PRE-PREPARE recebida

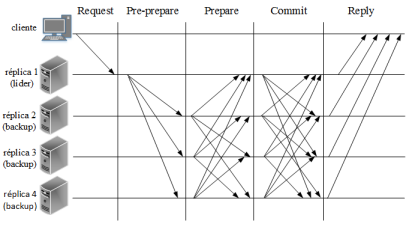

Figura 9: O protocolo PBFT em execução livre de faltas.

do líder, realiza um *multicast* às outras réplicas de uma mensagem PREPARE correspondente à sua mensagem PRE-PREPARE recebida anteriormente.

4. Ao receber $2f$ mensagens PREPARE de diferentes réplicas *backup*, cada réplica, incluindo o servidor primário, realiza um *multicast* de uma mensagem COMMIT às outras réplicas.

5. Cada réplica, após receber $2f+1$ mensagens COMMIT de diferentes réplicas (contando com sua própria mensagem COMMIT), executa a operação solicitada no pedido e envia uma mensagem REPLY para o cliente.

6. O cliente, ao receber $f+1$ mensagens REPLY válidas de diferentes réplicas, aceita o resultado recebido.

O cliente c, para ter seu pedido executado, envia à réplica primária uma mensagem $\langle \text{REQUEST}, o, t, c \rangle$, onde o é a operação a ser executada. Um *timestamp* t é usado para garantir que cada pedido seja executado uma única vez.

O servidor primário, ao receber o pedido do cliente, atribui um número de sequência n para essa solicitação e envia, por *multicast*, uma mensagem $\langle \langle \text{PRE-PREPARE}, v, n, d \rangle, m \rangle$ para as réplicas *backups*. (Essa fase é conhecida como *pre-prepare*.) Os atributos v, m, d da mensagem correspondem à visão a que pertence a mensagem, a mensagem do pedido do cliente e o resumo criptográfico de m, respectivamente.

Após a fase *pre-prepare*, inicia-se a fase *prepare*. Cada réplica *backup*, ao receber a mensagem PRE-PREPARE, realiza algumas verificações para se certificar que a mensagem é válida, por exemplo, verificando se o resumo da mensagem enviada é igual ao valor passado d e se a visão atual informada está correta. Se a mensagem for válida, a réplica a aceita e transmite, por *multicast*, para todas as réplicas do

sistema uma mensagem $\langle \text{PREPARE}, v, n, d, i \rangle$, onde i é a identificação do servidor. Caso contrário, nada é realizado.

As fases *pre-prepare* e *prepare* são usadas para garantir a ordem total dos pedidos enviados na mesma visão, mesmo quando o primário que lhes propõe a ordem é faltoso.

Cada réplica, para iniciar a próxima fase, conhecida como *commit*, precisa ter registrado em seu *log*: o pedido m, um *pre-prepare* para m na visão v com o número de sequência n, e $2f$ *prepares* de diferentes réplicas *backups* que combinam com a mensagem *pre-prepare*, ou seja, que tenham a mesma visão, número de sequência e resumo criptográfico.

Quando a condição mencionada anteriormente for verdadeira, então a réplica envia, através de *multicast*, uma mensagem $\langle \text{COMMIT}, v, n, D(m), i \rangle$ às outras réplicas do sistema. Cada réplica, ao aceitar $2f+1$ mensagens COMMIT de diferentes servidores (incluindo sua própria mensagem), executa a operação solicitada pelo cliente, seguindo a ordem que lhe foi atribuída pela réplica primária. Em seguida, a réplica envia diretamente a resposta para o cliente.

As fases *prepare* e *commit* são usadas para assegurar que os pedidos validados (*committed*) tenham ordem total em visões diferentes.

O cliente, ao receber $f+1$ mensagens REPLY de diferentes réplicas que tenham assinaturas válidas, *timestamp* t e resultado r iguais, aceita a resposta.

Se o cliente não receber as respostas dentro de um limite de tempo determinado pelo seu temporizador, ele envia, por *broadcast*, o pedido para todas as réplicas. Se o pedido já foi processado anteriormente, as réplicas simplesmente reenviam a resposta. Caso contrário, se a réplica não é a primária, ela retransmite o pedido para a primária. Se a réplica primária não envia o pedido para o grupo, em algum momento essa réplica será suspeitada como faltosa por uma quantidade suficiente de réplicas para que haja uma troca de visão.

O protocolo de troca de visão é utilizado para garantir continuidade (*liveness*) dos pedidos recebidos pelo sistema na presença de uma réplica primária faltosa. Este protocolo realiza uma troca de réplica primária: uma réplica *backup* correta assume o papel de primária na nova visão $v + 1$ no lugar da primária faltosa.

Cada réplica registra em seu *log* as mensagens que são utilizadas na execução do protocolo, porém para que essas mensagens não ocupem um grande espaço em disco com o passar do tempo, é realizado o *checkpoint* para descartar as mensagens.

O protocolo de *checkpoint* é executado periodicamente, sempre

que um pedido com um número de sequência divisível por alguma constante (por exemplo, 100) é executado. O *checkpoint* é dito como estável após as réplicas provarem que seu estado está correto. As réplicas descartam do seu *log* todas as mensagens *pre-prepare*, *prepare* e *commit* com número de sequência n menor ou igual ao estado que teve comprovada sua corretude.

3.1.2 BFT-TO: Intrusion Tolerance with Less Replicas

No trabalho BFT-TO (CORREIA; NEVES; VERISSIMO, 2004; CORREIA; NEVES; VERÍSSIMO, 2013) foi proposto um algoritmo tolerante a intrusões baseado na abordagem de máquina de estado (SMA - *State Machine Approach*). O modelo proposto, através da utilização de um componente confiável distribuído chamado *TO wormhole* (*Trusted Ordering Wormhole*), diminuiu o número total de réplicas n do sistema replicado BFT de $3f+1$ para $2f+1$, para tolerar no máximo f réplicas faltosas.

A arquitetura do sistema é mostrada na figura 10. O serviço de replicação de máquina de estado (SMR -*State Machine Replication*) é executado por um conjunto de servidores $S = \{s_1, s_2, \ldots, s_n\}$. O serviço pode ser chamado por um conjunto de clientes $C = \{c_1, c_2, \ldots, c_m\}$. Os servidores e clientes são conectados por uma rede, chamada de rede *payload*. Cada servidor precisa de um *TO wormhole* local. O *TO wormhole* foi implementado como um módulo do *kernel*, mas outras soluções poderiam ser utilizadas para garantir a segurança desse componente confiável, por exemplo, executando o *TO wormhole* em um módulo de *hardware* como uma placa PC/104. O canal de controle do *TO wormhole* utiliza uma rede própria para realizar a comunicação (cada servidor tem duas placas de rede).

As características principais do *TO wormhole* são:

- O componente é seguro, ou seja, resistente a qualquer tipo de ataque e pode apenas falhar por *crash*.
- Tempo-real, capaz de executar certas operações com um atraso limitado.

A comunicação entre o servidor e o *TO wormhole* é realizada através de uma API (*Application Programming Interface*) que disponibiliza as seguintes funções:

- *TOW_ register()*.

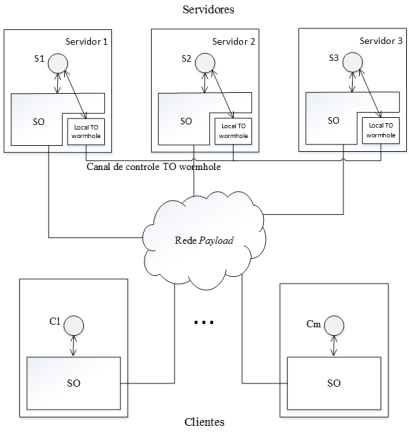

Figura 10: Arquitetura do sistema, para $f = 1$.

- *TOW_sent(id, members, msg_id, msg_hash).*
- *TOW_received(id, members, msg_id, msg_hash, sender_id).*
- *TOW_decide(sender_id, msg_id, order_n, hash, has_msg).*

O serviço de ordem é o núcleo da solução do trabalho proposto. Esse serviço, fornecido pelo componente confiável *TO wormhole*, atribui um número de ordem para as mensagens recebidas dos clientes através da rede *payload* e, dessa forma, auxilia o serviço SMR na implementação de um protocolo *multicast* com ordem total, conforme mostra a figura 11.

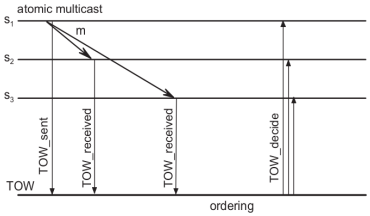

Figura 11: Exemplo de um protocolo *atomic multicast* usando o serviço *TO wormhole*.
(CORREIA; NEVES; VERÍSSIMO, 2013)

Para cada mensagem m difundida atomicamente, o *TO wormhole* indica duas coisas: (1) quando m pode ser entregue; e (2) a ordem na qual m tem que ser entregue.

Quando um servidor difunde uma mensagem, ele notifica esse evento para o *TO wormhole* chamando a função *TOW_sent*. Os parâmetros da função significam: (1) *id* é o identificador do servidor do ponto de vista do componente confiável (o *id* é obtido por chamar a função *TOW_register*), (2) O parâmetro *members* é um conjunto com os *ids* dos servidores registrados que implementam o algoritmo BFT-TO, (3) *msg_id* é um número da mensagem que deve ser único para o servidor que enviá-lo, (4) *msg_hash* é o resumo criptográfico da mensagem.

Um servidor, ao receber uma mensagem, notifica esse evento para seu componente *TO wormhole* chamando a função *TOW_received*. Os parâmentros são os mesmos da função *TOW_sent* mais o *id* do emissor: *sender_id*. As funções *TOW_received* e *TOW_sent* são interconectadas pelo canal de controle do *TO wormhole*.

Após ser notificado por um dos eventos, o *TO wormhole* de cada servidor fornece o próximo número de ordem para a mensagem e notifica os outros servidores chamando a função *TOW_decide*. Os parâmetros dessa função são: (1) *sender_id* é o identificador do servidor que enviou a mensagem, (2) *msg_id* é o identificador da mensagem, (3) *order_n* é o número de ordem da mensagem, (4) *hash* é o resumo criptográfico da mensagem e (5) *has_msg* é um conjunto com os *ids* dos servidores que têm a mensagem. A função *TOW_decide* dará o mesmo número de ordem para todos servidores, porque o *TO wormhole* é um componente seguro do sistema.

Quando $f + 1$ servidores notificam o evento para seu componente confiável, o *TO wormhole* pode ter certeza que pelo menos um servidor correto tem a mensagem, porque no máximo f servidores podem falhar. Se um servidor correto tem a mensagem, então o servidor vai enviá-la para todos os outros.

3.1.3 Zyzzyva: Speculative Byzantine Fault Tolerance

O trabalho de (KOTLA et al., 2007) propõe um protocolo especulativo, chamado Zyzzyva, que precisa de 3 passos para executar o pedido do cliente no caso livre de faltas. Em Zyzzyva, as réplicas do sistema BFT especulativamente executam os pedidos sem realizar um protocolo de acordo, no qual todas as réplicas corretas têm a garantia

que a ordem proposta é igual para elas. Consequentemente, os estados das réplicas corretas podem divergir, e réplicas podem enviar respostas diferentes para os clientes. Nesse caso, quando houver divergência nas respostas, é necessária a execução de dois passos adicionais, para que as réplicas corretas convirjam sobre a resposta enviada para o cliente.

O serviço BFT que o sistema proposto implementa é executado por n réplicas, onde $n = 3f + 1$ e no máximo f servidores podem falhar para manter a especificação correta do sistema. A execução é organizada dentro de uma sequência de visões. Em cada visão, uma única réplica é a primária que propõe uma ordem para a mensagem enviada do cliente, enquanto as outras são as réplicas *backups*. Esse modelo assume que réplicas ou clientes podem comportar-se de forma arbitrária.

O protocolo Zyzzyva no caso livre de faltas pode ser visto na figura 12. Esse protocolo funciona da seguinte maneira:

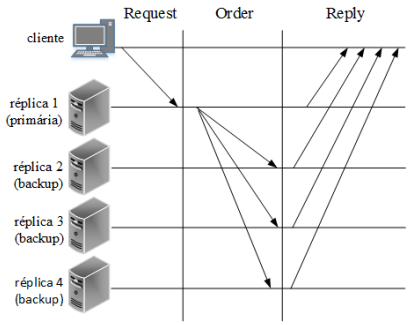

Figura 12: O protocolo Zyzzyva em execução livre de faltas.

1. O cliente envia o pedido para o servidor primário.
2. A réplica primária, ao receber o pedido, atribui a ele um número de sequência e envia uma mensagem ORDER-REQ para todas as réplicas *backups*.
3. Cada réplica executa especulativamente o pedido ordenado e envia uma mensagem SPEC-RESPONSE para o cliente. O cliente ao receber $3f + 1$ respostas iguais completa a requisição.

A operação o do cliente c que será executada pelas réplicas corretas do sistema BFT é enviada para o servidor primário através da mensagem $m = \langle \text{REQUEST}, o, t, c \rangle$, onde t é o *timestamp* da operação.

Quando a réplica primária recebe a mensagem do cliente, ela atribui um número de sequência n na visão v para o pedido e envia para as réplicas *backups* uma mensagem $\langle\langle \text{ORDER-REQ}, v, n, h_n, d, ND\rangle, m\rangle$, os parâmetros h_n, d e ND são: resumo criptográfico histórico de n, resumo criptográfico da mensagem m e um conjunto de valores não determinísticos necessários para execução, respectivamente.

A réplica i, ao receber a mensagem $\langle\langle \text{ORDER-REQ}, v, n, h_n, d, ND\rangle, m\rangle$ da primária, aceita esse pedido ordenado se d é um resumo criptográfico correto da mensagem m, $n = max_n + 1$, onde max_n é o maior número de sequência no histórico de i, e se h_n é um *hash* correto. No caso em que o pedido ordenado é aceito, a réplica i adiciona-o no seu histórico, executa o pedido e envia a resposta r para o cliente c através da mensagem $\langle\langle \text{SPEC-RESPONSE}, v, n, h_n, H(r), c, t\rangle, i, r, OR\rangle$, onde $H(r)$ é o resumo criptográfico da resposta r e $OR = \langle \text{ORDER-REQ}, v, n, h_n, d, ND\rangle$.

O cliente, ao receber $3f+1$ mensagens SPEC-RESPONSE idênticas de diferentes réplicas, aceita a resposta e completa o pedido.

Na execução quando pelo menos uma das réplicas é faltosa, o protocolo precisa de dois passos adicionais, conforme mostra a figura 13.

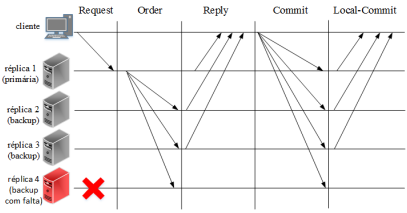

Figura 13: O protocolo Zyzzyva em execução com falta.

Nesse exemplo, o cliente recebe 3 respostas $(2f + 1)$ das réplicas (passo 3). Para não ter que ficar esperando por um tempo indeterminado as $3f + 1$ respostas, o cliente define um tempo de *timeout* quando envia um pedido. Quando esse tempo termina, se o cliente c recebeu entre $2f + 1$ e $3f$ mensagens SPEC-RESPONSE das réplicas, então ele envia uma mensagem $\langle \text{COMMIT}, c, CC\rangle$ para todas as réplicas (passo 4), onde CC é um certificado *commit* que contém uma lista de $2f + 1$ réplicas que enviaram a mensagem SPEC-RESPONSE para c e sua correspondente assinatura.

Cada réplica i correta, ao receber uma mensagem válida *commit* que fornece informação sobre o pedido a ser executado com um número

de sequência específico e histórico na visão atual, atualiza seu estado em relação ao certificado *commit* e envia para o cliente c uma mensagem $\langle \textit{local-commit}, v, d, h, i, c \rangle$.

Zyzzyva propôs mais dois protocolos: VIEW CHANGE e CHECKPOINT. O primeiro protocolo é executado para escolher uma nova réplica primária dentro de uma nova visão v, no caso em que o servidor primário é faltoso ou o sistema está lento. O segundo, é executado dentro de um intervalo pré-definido de requisições: CP_INTERVAL. O protocolo CHECKPOINT limita a quantidade de mensagens armazenadas pelas réplicas do sistema BFT, as quais constituem seu estado.

3.1.4 Attested Append-Only Memory: Making Adversaries Stick to their Word - A2M

O trabalho de (CHUN et al., 2007) propõe uma abstração de *log* confiável, chamada *Attested Append-Only Memory* (A2M), que impede uma réplica faltosa de enganar seus clientes enviando diferentes respostas para diferentes clientes de um mesmo pedido; por exemplo, um servidor faltoso ao tratar uma escrita concorrente recebida do cliente A e B para a mesma variável compartilhada poderia responder para que um cliente veja a escrita do cliente A como dominante, enquanto outro cliente vê a escrita do cliente B, em vez disso.

Os protocolos BFT que utilizam A2M melhoram tolerância a faltas arbitrárias por utilizar um número menor de réplicas para tolerar no máximo f faltas e continuar garantindo as propriedades *safety* e *liveness* do sistema BFT. O número total de réplicas que executam o pedido solicitado dos clientes é $2f + 1$, onde f é o número de servidores que podem falhar. O código A2M precisa ser executado em uma base de computação confiável para que o protocolo BFT assuma que uma réplica aparentemente correta possa dar somente uma única resposta para cada pedido.

Um módulo A2M prepara uma réplica para armazenar um conjunto de *logs* confiáveis ordenados e incontestáveis. Cada log tem um identificador único q e consiste de uma sequência de valores, sendo que cada valor é anotado com um número de sequência específico do *log* que é incrementado a partir de 0 e um resumo criptográfico incremental de toda entrada do *log*. Os valores são armazenados em cada *log* dentro de compartimentos (*slots*) que são representados pela posição *"low"* $\mathcal{L} \geq 0$ e posição *"high"* $\mathcal{H} \geq \mathcal{L}$, sendo $\mathcal{L}$ e $\mathcal{H}$ o primeiro e o último compartimento, respectivamente.

A interface do A2M oferece métodos para anexar valores, procurar valores no *log*, obter o fim do *log*, truncar e avançar os compartimentos armazenados na memória.

A seguir é explicado o protocolo A2M-PBFT-EA. O protocolo A2M-PBFT-EA é uma extensão do PBFT (CASTRO; LISKOV, 1999) que usa A2M para proteger a parte da execução e acordo (explicando o sufixo EA da sigla: *Execution* + *Agreement*), ou seja, garante que réplicas não podem enganar sobre seus resultados computados localmente para uma particular operação do cliente solicitada.

A figura 14 mostra o funcionamento do protocolo A2M-PBFT-EA no caso livre de faltas. Os passos do protocolo são enumerados a seguir:

1. O cliente envia um pedido para o servidor primário.

2. O servidor primário, ao receber o pedido do cliente, atribui ao mesmo um número de sequência, registra no seu *log* A2M uma mensagem PRE-PREPARE e obtém uma certificação dessa mensagem. Em seguida, a réplica primária envia através de *multicast* uma mensagem $\langle \text{PRE-PREPARE}, att\langle \text{PRE-PREPARE} \rangle \rangle$ para todas as réplicas *backups*, onde $att\langle \text{PRE-PREPARE} \rangle$ é a mensagem PRE-PREPARE certificada pelo módulo A2M.

3. A réplica *backup*, ao receber a mensagem $\langle \text{PRE-PREPARE}, att\langle \text{PRE-PREPARE} \rangle \rangle$, verifica a autenticação A2M dessa mensagem PRE-PREPARE recebida do líder, e envia por *multicast* às outras réplicas uma mensagem $\langle \text{PREPARE}, att\langle \text{PREPARE} \rangle \rangle$ correspondente à sua mensagem PRE-PREPARE recebida anteriormente.

4. Ao receber $f + 1$ mensagens válidas $\langle \text{PREPARE}, att\langle \text{PREPARE} \rangle \rangle$ de diferentes réplicas *backups* (incluindo sua própria mensagem), cada réplica, incluindo o servidor primário, envia por *multicast* uma mensagem $\langle \text{COMMIT}, att\langle \text{COMMIT} \rangle \rangle$ às outras réplicas.

5. Cada réplica, após receber $f+1$ mensagens $\langle \text{COMMIT}, att\langle \text{COMMIT} \rangle \rangle$ de diferentes réplicas (contando com sua própria mensagem COMMIT), executa a operação solicitada no pedido e envia uma mensagem $\langle \text{REPLY}, att\langle \text{REPLY} \rangle \rangle$ para o cliente.

6. O cliente, ao receber $f+1$ mensagens $\langle \text{REPLY}, att\langle \text{REPLY} \rangle \rangle$ válidas de diferentes réplicas, aceita o resultado recebido.

Para detalhar os conceitos de certificação da mensagem, a fase PREPARE é utilizada como exemplo desse processo. Quando uma réplica

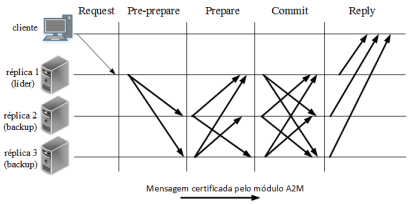

Figura 14: O protocolo A2M no caso livre de faltas. Linhas espessas mostram as mensagens que são certificadas usando A2M.

A2M-PBFT-EA recebe uma mensagem $\langle prepare, v, n, req\rangle$, onde v é a visão atual e n é a ordem proposta pela réplica primária para executar a requisicação *req*, a réplica registra essa mensagem no seu log A2M correspondente.

A função da interface A2M para realizar o armazenamento da mensagem anterior é $advance(m_p, [v|n], 0, h(prep))$, onde m_p é a mensagem PREPARE para ser armazenada no *log*, $[v|n]$ representa o número de entrada no espaço do *log*, 0 é o valor passado que indica o avanço do *log* por números de sequências múltiplos e $h(prep)$ é o resumo criptográfico da mensagem PREPARE.

Após a mensagem ser registrada, sua certificação LOOKUP é extraída $att\langle\text{PREPARE}\rangle = \langle\text{LOOKUP}, m_p, [v|n], [v|n], h(prep), \text{ASSIGNED}, [v|n], d'\rangle_{A2M_i,\mathcal{R},1}$. Essa certificação é uma assinatura digital do módulo A2M da réplica i que pode ser verificada pelo conjunto $\mathcal{R}$, o qual tem como elementos as réplicas do sistema BFT.

Quando uma réplica A2M-PBFT-EA recebe uma mensagem PREPARE certificada, ela verifica sua autenticação A2M, e então confirma que o valor certificado é o *hash* da mensagem PREPARE incluída. Quando uma réplica recebe $f + 1$ mensagens que combinam para o mesmo número de sequência n e visão v, a fase PREPARE é concluída. A fase COMMIT é similar à fase PREPARE descrita.

3.1.5 Efficient Byzantine Fault-Tolerance - MinBFT e MinZyzzyva

MinBFT e MinZyzzyva são dois algoritmos de replicação de máquina de estado tolerante a faltas bizantinas apresentados em (VERONESE et al., 2013). Os algoritmos MinBFT e MinZyzzyva requerem apenas $2f + 1$ réplicas, onde f é o número de servidores que podem

falhar. A redução do número de réplicas foi alcançada pela utilização de um componente confiável chamado *Unique Sequential Identifier Generator* (USIG).

O USIG é um serviço local existente em cada servidor para atribuir às mensagens identificadores únicos e sequenciais. Em operações sem faltas, MinBFT e MinZyzzyva têm o número de passos de comunicação igual a 4 e 3, respectivamente; o primeiro é um algoritmo não especulativo e o segundo é especulativo.

A interface do serviço USIG tem duas funções:

- *createUI(m)*: retorna um certificado USIG que contém um identificador único UI assinado pelo componente confiável para a mensagem *m*. O identificador único é basicamente uma leitura do contador que é incrementado sempre que a função *createUI* é chamada.

- *verify(PK, UI, m)*: verifica se o identificador único UI é válido para a mensagem *m*, ou seja, se o certificado USIG corresponde à mensagem e ao restante dos dados no UI.

O serviço USIG pode ser implementado utilizando o certificado baseado em *hashes* criptográficos ou criptografia de chave pública, conforme é descrito a seguir:

- *USIG-Hmac*: Um certificado contendo um código de autenticação de mensagem baseado em *hash* (HMAC), obtido usando a mensagem e uma chave secreta conhecida pelo USIG e por todos os outros que precisam verificar os certificados gerados.

- *USIG-Sign*: O certificado contém uma assinatura obtida usando a mensagem e a chave privada do USIG.

A execução do serviço USIG deve ser isolada, o código não pode ser alterado e é assumido que o componente não pode ser corrompido.

3.1.5.1 MinBFT

O protocolo não especulativo MinBFT é mostrado na figura 15. O algoritmo apresenta os seguintes passos:

1. Um cliente envia um pedido para todos os servidores.

2. A réplica primária atribui um número de sequência (ordem) para o pedido e o envia para todos os servidores através da mensagem PREPARE.

3. Cada réplica envia por meio de *multicast* uma mensagem COMMIT às outras réplicas, após receber da réplica primária uma mensagem PREPARE.

4. Quando um servidor aceita um pedido, ele executa a operação correspondente e envia uma resposta para o cliente.

5. O cliente aguarda $f+1$ respostas válidas para o pedido e completa a operação.

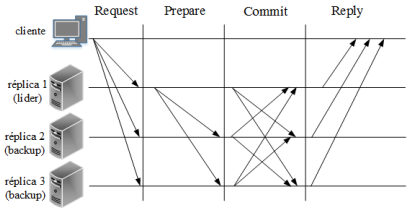

Figura 15: O protocolo MinBFT no caso livre de faltas.

As operações são executadas em configurações sucessivas chamadas de visões. Cada visão tem uma réplica primária e as outras são réplicas *backups*. A primária é o servidor s_p com $p = v \ mod \ n$, onde v é o número de visão atual e n é o número total de réplicas do sistema BFT.

Um cliente c solicita a execução de uma operação op ao enviar uma mensagem $\langle \text{REQUEST}, c, seq, op \rangle$, onde seq é o identificador do pedido, utilizado para garantir a execução uma única vez.

Quando a réplica primária recebe um pedido do cliente, ela utiliza a mensagem PREPARE para enviar através de *multicast* o pedido às outras réplicas. O papel da réplica primária é atribuir um número de sequência para cada pedido. Esse número é o valor do contador retornado pelo serviço USIG no identificador único UI.

A ideia básica é que um pedido m é enviado pela réplica primária s_i às outras réplicas *backups* através da mensagem $\langle \text{PREPARE}, v, s_i, m, UI_i \rangle$, e cada servidor s_j reenvia o pedido para todos os outros servidores por meio da mensagem $\langle \text{COMMIT}, v, s_j, s_i, m, UI_i, UI_j \rangle$, onde UI_j é obtido por chamar a função *createUI*. Cada mensagem PREPARE ou

COMMIT tem um identificador único UI obtido pela chamada da função *createUI*. Dessa forma, duas mensagens diferentes não podem ter o mesmo identificador. Servidores verificam se os identificadores das mensagens que eles recebem são válidos para essas mensagens, usando a função *verifyUI*.

Cada réplica do sistema BFT, ao receber $f + 1$ mensagens válidas COMMIT de diferentes servidores, aceita esse pedido, executa sua operação e envia a resposta para o cliente.

O cliente espera $f + 1$ respostas $\langle \text{REPLY}, s, seq, res \rangle$ de diferentes servidores s que combinam o resultado *res*, as quais garantem que pelo menos uma resposta provém de um servidor correto.

3.1.5.2 MinZyzzyva

O algoritmo BFT MinZyzzyva é especulativo, sendo uma versão modificada do primeiro algoritmo BFT especulativo Zyzzyva (KOTLA et al., 2007). Esse algoritmo utiliza 3 passos para executar um pedido do cliente no caso livre de faltas, porém quando há suspeita de falta são necessários 5 passos para completar a requisição do cliente.

A ideia da especulação é que servidores respondem a pedidos dos clientes sem realizar um acordo sobre a ordem na qual os pedidos são executados. Os servidores, de forma otimista, aceitam a ordem proposta pelo servidor primário, executam o pedido, e respondem imediatamente para o cliente.

Essa execução é especulativa, pois aquela pode não ser a ordem real em que o pedido deveria ser executado. Se os servidores são inconsistentes entre eles, clientes detectam essas inconsitências e ajudam os servidores para convergirem sobre uma única ordem total dos pedidos, possivelmente através de reverter (*rollback*) algumas das execuções. Clientes somente baseiam-se nas respostas que são consistentes com esta ordem total.

O protocolo MinZyzzyva no caso livre de faltas (mostrado na figura 16-a) funciona da seguinte maneira:

1. Um cliente envia um pedido através da mensagem REQUEST para a réplica primária s_p.

2. O primário recebe o pedido, atribui-lhe um identificador único UI_p contendo o número de sequência chamando a função *createUI*, e encaminha o pedido e UI_p às outras réplicas.

3. Cada réplica recebe o pedido, verifica se UI_p é válido, atribui

outro identificador único UI_s para o pedido, especulativamente executa-o e envia a resposta por meio da mensagem RESPONSE para o cliente (com os dois identificadores UI).

4. O cliente recebe as respostas e apenas aceita mensagens com UI_p e UI_s válidos.

5. Se o cliente recebe $2f+1$ respostas válidas, o pedido é completado.

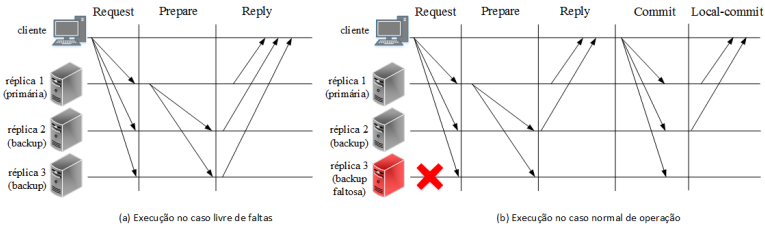

Figura 16: O protocolo MinZyzzyva.

Quando a rede é lenta ou pelo menos um servidor é faltoso, o cliente pode nunca receber $2f + 1$ respostas válidas de diferentes réplicas. Quando um cliente envia um pedido, ele define um tempo limite (*timer*). Se o tempo expira e o cliente recebeu entre $f + 1$ e $2f$ respostas válidas, então ele envia uma mensagem COMMIT que contém um certificado *commit* com estas respostas (identificadores UI_p e UI_s) para todos os servidores. Um certificado *commit* é composto por $f + 1$ respostas válidas de $f + 1$ servidores diferentes.

Cada réplica, ao receber um certificado válido *commit* de um cliente, reconhece-o com uma mensagem LOCAL-COMMIT. O cliente, ao receber $f + 1$ mensagens LOCAL-COMMIT, completa o pedido (figura 16-b).

3.1.6 Byzantine fault-tolerant state machine replication with twin virtual machines - TwinBFT

Um algoritmo (TwinBFT) para replicação de máquina de estado tolerante a faltas bizantinas foi apresentado em (DETTONI et al., 2013). A abordagem utiliza a tecnologia de virtualização para reduzir o número total de réplicas físicas necessárias de $3f + 1$ para $2f + 1$, e o algoritmo utiliza 3 passos de comunicação no caso normal de operação, sem a participação do cliente no acordo.

No modelo de sistema, cada máquina física executa um conjunto de máquinas virtuais gêmeas. Considerando um sistema adotando $f = 1$, cada máquina física executará duas máquinas virtuais (*Virtual Machines* - VMs) e cada máquina virtual atua como um detector de falhas para sua gêmea.

Quando o número de máquinas virtuais em cada máquina hospedeira é maior que dois, a quantidade de VMs é obtida por $nVM \geq 2fVM+1$, onde fVM é o número de máquinas virtuais que podem falhar. Dessa forma, a resposta correta é garantida pela maioria das VMs $(fVM + 1)$, pois pelo menos uma resposta pertence a uma máquina virtual correta.

Para realizar a comunicação entre diferentes VMs dentro de um mesmo *host* é utilizado um espaço de memória compartilhada, chamado *postbox*. A comunicação entre VMs de diferentes *hosts* é feita apenas por troca de mensagens.

Os passos do protocolo TwinBFT, conforme mostra a figura 17, são descritos abaixo:

1. Cliente envia o pedido para ambos os processos (VM1 e VM1') na réplica primária.

2. O líder primário v_i atribui um número de sequência e insere uma mensagem ORDER na *postbox*.

3. A réplica backup primária v_i' lê a mensagem da *postbox*, consegue o número de sequência e insere na *postbox* a mensagem ORDER contendo o pedido original e o número de sequência recebido do líder.

4. Os processos (VM1 e VM1') assinam a mensagem ORDER lida da *postbox* e enviam a mensagem às outras réplicas.

5. Cada processo nas réplicas (VM), ao receber a mensagem ORDER, executa a operação e insere na *postbox* uma mensagem assinada REPLY .

6. Cada processo, ao ler 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. O cliente, ao receber $f + 1$ respostas corretamente assinadas de diferentes réplicas, aceita a resposta.

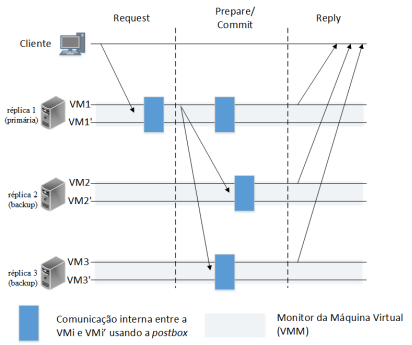

Figura 17: O protocolo TwinBFT no caso livre de faltas.

O cliente c envia uma mensagem $\langle \text{REQUEST}, c, seq, op \rangle$ para ambos os processos na réplica primária (VM1 e VM1') para executar a operação op na sequência seq em relação ao cliente.

Qualquer processo da VM_i e VM_i' no host primário, ao receber a requisição do cliente, gera um novo número de sequência n e cria uma mensagem $\langle \langle \text{ORDER}, p_i, v, n, dm \rangle, m \rangle$, onde v é o número da visão atual, dm o resumo da mensagem m e p_i é o processo líder da VM_i. Assim que o processo da VM_i' lê a mensagem ORDER inserida na *postbox* por p_i e possui sua mensagem REQUEST correspondente, esse processo obtém o número de sequência proposto por p_i, cria uma mensagem ORDER, assina-a e insere a mensagem assinada na *postbox*.

Quando cada um dos processos (VM1 e VM1') lê a mensagem ORDER da *postbox*, verifica se todos os parâmetros correspondem aos processados localmente e, se todos esses valores estiverem corretos, adiciona sua própria assinatura à mensagem de sua gêmea e envia-a às outras réplicas *backups*.

Cada processo VM_i e VM_i', ao receber a mensagem ORDER, verifica se a mensagem é válida, e caso for positivo, executa a operação e cria a mensagem $\langle \text{REPLY}, p_i, v, seq, c, res \rangle$, com o resultado res da operação executada, e insere a mensagem na *postbox*. Quando um processo lê a mensagem REPLY da *postbox*, compara os seus parâmetros, e, se forem idênticos aos processados localmente, assina a mensagem e a envia ao cliente.

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

3.1.7 ByzID: Byzantine Fault Tolerance from Intrusion Detection

O trabalho de (DUAN et al., 2014) propõe uma abordagem que utiliza detecção de intrusões para diminuir o número de réplicas utilizadas no sistema BFT. O protocolo ByzID utiliza um sistema de detecção de intrusão confiável (IDS) baseado em especificação, ou seja, o IDS tem o conhecimento do comportamento desejado de um determinado sistema e considera como uma violação qualquer sequência de operações que não seguem as especificações desejadas do sistema BFT.

O modelo do sistema utiliza um número total de réplicas igual a $2f+1$, onde f é o número de servidores que podem falhar, para executar a operação solicitada pelos clientes. Cada réplica tem um componente confiável IDS que monitora os pacotes de entrada e saída da rede da réplica. O IDS não pode modificar as mensagens, ele apenas captura os pacotes de rede relacionados às mensagens do protocolo ByzID e analisa-os de acordo com a sua especificação.

O IDS atua como um oráculo distribuído e dispara alertas (*triggers*) quando a réplica não segue as especificações do protocolo ByzID. No caso de um alerta, a réplica detectada deveria ser recuperada ou removida através de um procedimento de reconfiguração. Durante esse processo, as mensagens da réplica faltosa deveriam ser bloqueadas através de um *firewall* local.

As especificações do detector de faltas bizantinas são:

- Consistência: O primário envia mensagens ORDER consistentes para outras réplicas.
- Ordem Total: O primário envia pedidos em ordem total às réplicas *backups*.
- *Fairness*: O primário ordena pedidos na ordem FIFO.
- Ação oportuna: O primário ordena os pedidos dos clientes utilizando um tempo limite.

O protocolo ByzID, ilustrado na figura 18, tem os seguintes passos:

1. Cliente envia um pedido para a réplica primária.
2. A réplica primária atribui um número de sequência para o pedido e envia uma mensagem ORDER às réplicas *backups*.

3. Cada réplica *backup*, ao receber a mensagem ORDER, envia uma mensagem ACK para a réplica primária, executa o pedido e envia uma resposta para o cliente.

4. A réplica primária, ao receber todas as mensagens ACK das réplicas *backups*, completa o pedido. Caso contrário, ela retransmite a mensagem ORDER.

5. O cliente, ao receber $f + 1$ respostas válidas, completa o pedido.

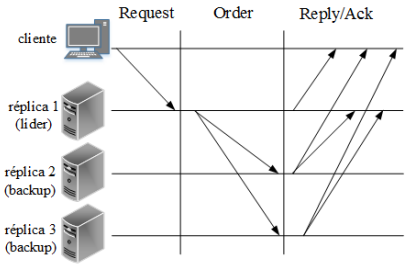

Figura 18: O protocolo ByzID no caso livre de faltas.

Um cliente c envia para a réplica primária p_0 uma mensagem $\langle \text{REQUEST}, o, \mathcal{T}, c \rangle$, onde o é a operação solicitada e $\mathcal{T}$ é o *timestamp* com um valor específico para esse pedido. Quando a réplica primária recebe um pedido do cliente, atribui-lhe um número de sequência $\mathcal{N}$ e envia uma mensagem $\langle \text{ORDER}, \mathcal{N}, m, v, c \rangle$ às réplicas *backups*, onde m é o pedido do cliente, v é o número de configuração/visão atual, e c é um identificador do cliente.

O IDS na réplica primária verifica as especificações mencionadas anteriormente. Se as especificações são violadas, o IDS bloqueia as mensagens correspondentes e gera um alerta para que a réplica primária seja reconfigurada.

Quando a réplica p_i recebe uma mensagem $\langle \text{ORDER}, \mathcal{N}, m, v, c \rangle$, essa réplica envia uma mensagem $\langle \text{ACK}, \mathcal{N}, \mathcal{D}(m), v, c \rangle$ à replica primária com o mesmo $\mathcal{N}$, m, v, e c da mensagem ORDER correspondente. Uma réplica *backup* p_i aceita a mensagem ORDER se o pedido m é válido, sua configuração atual é v, e $\mathcal{N} = \mathcal{N}' + 1$, onde $\mathcal{N}'$ é o número de sequência do seu último pedido aceito. Se a réplica p_i aceita a mensagem ORDER, ela executa a operação o em m e envia para o cliente uma mensagem $\langle \text{REPLY}, c, r, \mathcal{T} \rangle$, onde r é o resultado da operação o, e $\mathcal{T}$ é o *timestamp* do pedido m.

Se a réplica p_i recebe uma mensagem ORDER com número de sequência $\mathcal{N} > \mathcal{N}' + 1$, então ela armazena a mensagem em seu *log* e espera as mensagens com número de sequência entre $\mathcal{N}$ e $\mathcal{N}'$ para serem executadas primeiramente, conforme a sequência correta.

O IDS na réplica *backup* p_i inicia um *timer* quando percebe uma mensagem ORDER. Se p_i não envia uma mensagem ACK no tempo esperado, o IDS gera um alerta.

Quando a réplica primária recebe uma mensagem $\langle \text{ACK}, \mathcal{N}, \mathcal{D}(m)-, v, c \rangle$, ela aceita a mensagem se os campos $\mathcal{N}$, m, v, e c combinam com aqueles em sua mensagem ORDER correspondente. Se a réplica primária recebe uma mensagem ACK de todas as réplicas *backups*, ela completa o pedido do cliente.

O cliente completa um pedido quando recebe $f + 1$ respostas válidas das réplicas do sistema BFT.

3.2 ABORDAGENS BASEADAS EM DETECTORES DE FALTAS

3.2.1 Unreliable Intrusion Detection in Distributed Computations

O trabalho de (MALKHI; REITER, 1997) apresenta a abstração de um detector de faltas não confiável que um processo pode usar para detectar faltas bizantinas de outro processo.

O detector de faltas não confiável pode informar que um processo correto é faltoso ou que um processo faltoso é correto. Entretanto, (MALKHI; REITER, 1997) mostram que se esses detectores satisfizeram determinadas propriedades, então o problema do consenso distribuído pode ser resolvido.

A proposta trata faltas bizantinas, considerando apenas aquelas que impedem o progresso do protocolo, ou seja, quando um processo está esperando por uma mensagem de outro processo que nunca chegará. É assumida que toda comunicação no protocolo é realizada de forma confiável pelos processos corretos.

É definida uma classe de detector de faltas bizantinas $\diamond S(bz)$ que requer em algum momento que todos os processos corretos suspeitem como faltoso cada processo que deixa de enviar mensagens (*Strong Completeness*), e que existe um tempo após o qual qualquer processo correto deixa de ser suspeito por todos os processos corretos (*Eventual Weak Accuracy*).

O modelo do sistema consiste de um grupo de n processos, $p_0, p_1,-$

$\ldots, p_{n-1}$. Processos que seguem as especificações do protocolo são chamados corretos e aqueles que não seguem são chamados faltosos. É assumido que no máximo $[\frac{n-1}{3}]$ (onde n é o número total de processos/réplicas do sistema BFT) processos podem falhar para que o sistema continue funcionando conforme sua especificação. Considera-se que o sistema é assíncrono, ou seja, não há limite conhecido para a duração dos passos de computação ou da transferência de mensagens.

Cada processo tem acesso a um módulo local do detector de faltas que fornece-lhe uma lista de processos suspeitos. Essa lista pode mudar ao longo do tempo e pode ser diferente para diferentes processos. Se um processo p está na lista do detector de faltas do processo q, então é dito que q suspeita p.

A proposta utiliza o detector de faltas bizantinas $\diamond S(bz)$ em um protocolo para resolver o problema do consenso. Cada processo correto inicia sua execução propondo um valor binário, e um processo correto completa o protocolo decidindo sobre um valor binário de forma irrevogável. Um protocolo de consenso garante que as seguintes propriedades sejam satisfeitas:

1. Cada processo correto decide um valor.

2. Todo processo correto decide o mesmo valor, chamado valor do consenso.

3. Se todos os processos corretos mantêm o mesmo valor no início do protocolo, então esse é o valor do consenso.

O protocolo para resolver consenso (como mostrado na figura 19) é executado em rodadas assíncronas. Em cada rodada, um único processo é o líder. Todas as mensagens são identificadas com a rodada para a qual elas pertencem. Ao longo do protocolo, cada processo p_i mantém uma variável v_i que inicialmente contém seu valor de entrada.

Uma rodada inicia com cada processo p_i enviando, através de *broadcast*, uma mensagem contendo o valor escolhido v_i e o número da rodada. Quando o líder recebe $[\frac{2n+1}{3}]$ dessas mensagens, ele escolhe o valor proposto pela maioria dos processos, ou seja, o valor sugerido por pelo menos $[\frac{n-1}{3}]+1$ processos. Em seguida, envia uma mensagem sugerindo o valor escolhido pela maioria como o valor do consenso.

Cada processo então responde aos outros processos com mensagens contendo uma confirmação positiva ou negativa para o valor do consenso sugerido, onde um processo envia uma confirmação negativa se ele suspeita do líder como faltoso.

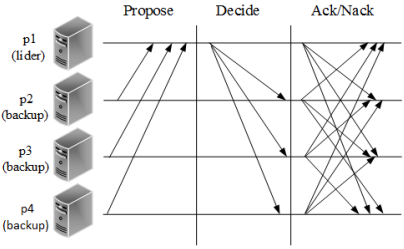

Figura 19: O protocolo para resolver consenso com detector de faltas não confiável.

Cada processo recebe $[\frac{2n+1}{3}]$ destas mensagens de confirmação. Se um processo obtém $[\frac{n-1}{3}]+1$ confirmações positivas, então ele muda seu v_i para o valor sugerido e inicia a próxima rodada.

Além disso, se um processo recebe $\frac{2n+1}{3}$ confirmações positivas para o valor sugerido daquela rodada, então ele imediatamente decide aquele valor e termina o protocolo.

3.2.2 Muteness Failure Detectors: Specification and Implementation

O detector de faltas de mudez $\diamond M_{\mathcal{A}}$ é apresentado em (DOUDOU et al., 1999). Faltas de mudez são faltas maliciosas/bizantinas nas quais um processo para de enviar mensagens do algoritmo, mas pode continuar enviando outras mensagens, por exemplo, mensagens "*I-am-alive*". Intuitivamente, um processo p é mudo em relação ao Algoritmo $\mathcal{A}$ se p para de enviar mensagens de $\mathcal{A}$ para um ou mais processos.

O detector de faltas de mudez $\diamond M_{\mathcal{A}}$ é limitado a uma classe de algoritmos distribuídos, os quais são chamados de algoritmos baseados em rodadas regulares. Essa classe inclui todos os algoritmos que têm um padrão de comunicação baseado em rodada e regular. A maioria dos algoritmos de consenso que utilizam detectores de faltas não confiáveis pertencem a essa classe.

O modelo do sistema considera um sistema distribuído assíncrono, ou seja, não há limite superior sobre o tempo requerido para um processamento ou uma comunicação. O sistema é composto de um conjunto finito $\Omega = p_1, \ldots, p_n$ de $\mathcal{N}$ processos, interconectados totalmente através de um conjunto de canais confiáveis de comunicação.

Cada algoritmo $\mathcal{A}$ gera um conjunto de mensagens que tem al-

guma sintaxe específica. É dito que uma mensagem m é uma mensagem de $\mathcal{A}$ se a sintaxe de m corresponde à sintaxe das mensagens que podem ser geradas por $\mathcal{A}$. Uma mensagem m enviada por um processo bizantino que leva uma semântica faltosa, mas tem uma sintaxe correta, é considerada como uma mensagem de $\mathcal{A}$.

Considerando f o número máximo de processos faltosos tolerados no sistema para que sua especificação continue correta, é $\mathcal{N} - f$ o número mínimo de processos corretos no sistema.

Um detector de faltas de mudez é um oráculo distribuído utilizado para detectar processos mudos. Tanto o conceito de mudez quanto o detector de mudez estão relacionados com algum algoritmo $\mathcal{A}$. Formalmente, o detector de faltas de mudez é expresso em termos das seguintes propriedades:

- *Eventual mute completeness*: existe um tempo após o qual cada processo que é mudo, com relação a $\mathcal{A}$, é suspeitado por um processo correto p.

- *Eventual weak accuracy*: existe um tempo após o qual um processo correto p não é mais suspeitado de ser mudo, com relação a $\mathcal{A}$, pelos outros processos corretos.

Existem algoritmos nos quais o uso do detector $\diamond M_{\mathcal{A}}$ não faz sentido. Mais precisamente, para esses algoritmos, é impossível implementar $\diamond M_{\mathcal{A}}$, mesmo em um modelo síncrono total. Esses algoritmos são aqueles para os quais um processo mudo não pode ser distinguido de um processo correto, mesmo considerando um sistema síncrono total, ou seja, algoritmos onde mudez pode ser um comportamento correto.

Portanto, é definida uma classe de algoritmos, chamada $\mathcal{C}_{\mathcal{A}}$, para a qual o uso do $\diamond M_{\mathcal{A}}$ faz sentido. Essa classe é caracterizada por especificar o conjunto de atributos que devem ser apresentados por qualquer algoritmo $\mathcal{A} \in \mathcal{C}_{\mathcal{A}}$. Esses algoritmos são qualificados como algoritmos baseados em rodadas regulares.

Atributo(a): cada processo correto p possui uma variável identificada como $round_p$ que tem seu intervalo $\mathcal{R}$ para ser o conjunto dos números naturais $\mathbb{N}$.

Atributo(b): em cada rodada existe no máximo um processo q do qual todos processos corretos estão esperando por uma ou mais mensagens. Considera-se q como processo crítico da rodada n e suas mensagens esperadas como mensagens críticas.

Atributo(c): com pelo menos $\mathcal{N} - f$ processos corretos participando no algoritmo $\mathcal{A}$, cada processo p é crítico a cada k rodadas,

$k \in \mathbb{N}$, e se, além disso, p é correto, então ele envia uma mensagem para todos naquela rodada.

As interações entre $\mathcal{A}_p$ e $\diamond M_{\mathcal{A}}$ são mostradas na figura 20. O algoritmo $\mathcal{A}_p$ executado por um processo correto p consulta seu detector $\diamond M_{\mathcal{A}}$ (seta 1). Cada vez que p recebe uma mensagem de algum processo q (seta 2), $\mathcal{A}_p$ entrega "*q-is-not-mute*"para a implementação $\mathcal{I}_{\mathcal{D}}$ (seta 3). No começo de cada rodada, $\mathcal{A}_p$ entrega um conjunto $new\_critical_p$ para $\mathcal{I}_{\mathcal{D}}$ (seta 4), contendo os processos críticos da nova rodada.

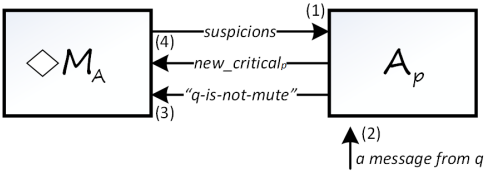

Figura 20: Interações entre $\mathcal{A}_p$ e $\diamond M_{\mathcal{A}}$.

3.2.3 Byzantine Fault Detectors for Solving Consensus

O trabalho de (KIHLSTROM; MOSER; SMITH, 2003) utiliza detectores de faltas não confiáveis para resolver o problema de consenso em sistemas distribuídos assíncronos que estão sujeitos a faltas bizantinas.

O modelo do sistema consiste de n processos. Cada processo no sistema tem um identificador de processo único. O sistema é assíncrono, ou seja, não há limite estipulado sobre o tempo necessário para um processamento ou para comunicação de uma mensagem. Cada processo tem acesso apenas ao seu relógio local, não sincronizado com outros processos.

A comunicação na rede é realizada através de canais, pelos quais pares de processos se comunicam utilizando primitivas *send* e *receive*. Os canais de comunicação são confiáveis: mensagens enviadas entre processos corretos em algum momento são recebidas e não são modificadas pelo meio de comunicação. Entretanto, mensagens podem ser atrasadas e entregues em uma ordem diferente em que foram enviadas.

Processos podem ser corretos ou faltosos. Processos corretos sempre comportam-se de acordo com a especificação desejada do sistema. Processos faltosos exibem comportamentos arbitrários. O número máximo de processos bizantinos k para que o sistema funcione

corretamente é $k \leq [\frac{n-1}{3}]$. Dessa forma, pelo menos $[\frac{2n+1}{3}]$ processos são corretos.

É assumido que um mecanismo de autenticação é disponível para que um processo possa verificar o emissor original da mensagem, mesmo se outro processo retransmitir a mensagem.

Os detectores de faltas bizantinas são definidos em termos de faltas bizantinas que são detectáveis, ou seja, desvios detectáveis do algoritmo que utiliza o detector de faltas.

É definida uma nova propriedade *completeness* para um detector de faltas bizantinas usado pelo algoritmo $\mathcal{A}$, conforme descrito abaixo:

- *Eventual strong Byzantine completeness* para algoritmo $\mathcal{A}$: existe um tempo após o qual cada processo que não segue a especificação do algoritmo $\mathcal{A}$ é suspeitado por todos os processos corretos.

Além da propriedade *completeness* definida acima, o trabalho (KIHLSTROM; MOSER; SMITH, 2003) utiliza as propriedades *eventual accuracy* definidas por (CHANDRA; TOUEG, 1996).

Duas classes de detectores de faltas bizantinas são definidas:

- Um detector de faltas bizantinas está na classe $\diamond P(Byz, \mathcal{A})$ se ele satisfaz as propriedades *eventual strong Byzantine completeness* e *eventual strong accuracy* para o algoritmo $\mathcal{A}$.

- Um detector de faltas bizantinas está na classe $\diamond S(Byz, \mathcal{A})$ se ele satisfaz as propriedades *eventual strong Byzantine completeness* e *eventual weak accuracy* para o algoritmo $\mathcal{A}$.

(CHANDRA; TOUEG, 1996) definiram as propriedades *eventual strong completeness* e *eventual weak completeness* para o modelo de faltas por *crash* e apresentaram um algoritmo que transforma um detector de faltas $\mathcal{D}$ que satisfaz *eventual weak completeness* em um detector de faltas $\mathcal{D}'$ que satisfaz *eventual strong completeness*.

Entretanto, em um sistema tolerante a faltas bizantinas, embora o algoritmo de transformação de (CHANDRA; TOUEG, 1996) transforme a propriedade acima em *eventual strong Byzantine completeness* para o algoritmo $\mathcal{A}$, o algoritmo de transformação não pode garantir que *eventual weak accuracy* ou *eventual strong accuracy* sejam preservadas na presença de até mesmo uma única falta bizantina.

Para tratar esse problema, é considerada uma alternativa para o algoritmo de transformação, no qual um processo p adiciona outro processo q na lista de suspeitos, somente se p recebeu $k+1$ mensagens suspeitando de q (incluindo a sua própria) de acordo com o detector

$\mathcal{D}$ de cada processo. Dessa forma, um algoritmo preserva *eventual weak* ou *strong accuracy*, mas não transforma a propriedade *eventual weak completeness* em *eventual strong Byzantine completeness* para o algoritmo $\mathcal{A}$.

O problema é que a propriedade dada acima requer apenas que exista um processo correto que suspeita o processo faltoso e não requer as $k+1$ mensagens. Para resolver esse problema, foi definida uma nova propriedade *weak completeness* para um detector de faltas usado por um algoritmo $\mathcal{A}$, como segue:

- *Eventual weak Byzantine $(k+1)$- completeness* para o algoritmo $\mathcal{A}$: existe um tempo após o qual cada processo que não segue a especificação do algoritmo $\mathcal{A}$ é permanentemente suspeitado por pelo menos $k+1$ processos corretos.

Mais duas classes foram definidas de detectores de faltas bizantinas:

- Um detector de faltas está na classe $\diamond Q(Byz, \mathcal{A})$ se ele satisfaz as propriedades *eventual weak Byzantine $(k+1)$- completeness* e *eventual strong accuracy* para o algoritmo $\mathcal{A}$.
- Um detector de faltas está na classe $\diamond W(Byz, \mathcal{A})$ se ele satisfaz as propriedades *eventual weak Byzantine $(k+1)$- completeness* e *eventual weak accuracy* para o algoritmo $\mathcal{A}$.

Na tabela 1, são mostradas as quatro classes de detectores de faltas bizantinas.

Tabela 1: Classes de detectores de faltas bizantinas

Completeness para Algoritmo A	*Accuracy*	
	Eventual strong	*Eventual weak*
Eventual strong Byzantine	*Eventually perfect* $\diamond P(Byz, \mathcal{A})$	*Eventually strong* $\diamond S(Byz, \mathcal{A})$
Eventual weak Byzantine (k + 1)	$\diamond Q(Byz, \mathcal{A})$	$\diamond W(Byz, \mathcal{A})$

3.2.4 The Case for Byzantine Fault Detection

O trabalho de (HAEBERLEN; KOUZNETSOV; DRUSCHEL, 2006) considera faltas bizantinas, ou seja, um servidor pode exibir com-

portamentos arbitrários em sistemas distribuídos. É apresentada uma abordagem que tem como objetivo detectar faltas bizantinas. Cada servidor tem seu detector correspondente que monitora outros servidores para avisar/alertar comportamentos maliciosos.

É assumido que o detector em um servidor correto pode observar todas as mensagens enviadas e recebidas por esse servidor. Isso significa que algumas faltas bizantinas não são tratadas e, portanto, não podem ser detectadas.

Como exemplos de comportamentos que podem ser detectados, é discutido um protocolo simples que tem apenas dois métodos: um método *put*, que é usado para armazenar um objeto em um servidor, e um método *get*, que é usado para recuperá-lo. A figura 21(a) mostra um exemplo simples de troca de mensagens, no qual o servidor B recebe um objeto do servidor A e depois o entrega para o servidor C.

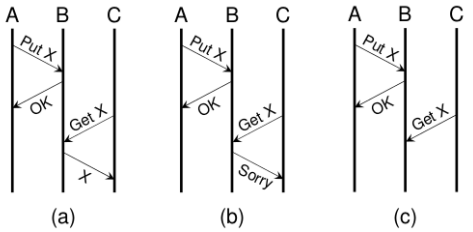

Figura 21: Exemplos de trocas de mensagens: Servidor B é (a)*correct*, (b)*detectably faulty*, (c)*detectably ignorant*. (HAEBERLEN; KOUZNETSOV; DRUSCHEL, 2006)

Considerando uma situação em que o servidor B é faltoso e quer impedir o servidor C de obter o objeto, existem duas maneiras para B alcançar isso:

1. Não seguir a especificação correta do protocolo e negar o pedido do servidor C (figura 21(b)); esse comportamento é chamado de *detectably faulty*.

2. Fingir que não recebeu a mensagem do pedido (figura 21(c)); esse comportamento é chamado de *detectably ignorant*.

Nos dois casos anteriores, a falta pode ser detectada porque ela afeta a troca de mensagens observadas pelos servidores corretos. Entretanto, um servidor pode se tornar faltoso, mas continuar seguindo o protocolo exatamente como se fosse correto. Essa falta não pode ser

detectada pela abordagem proposta. De forma similar, se uma falta é completamente interna para um servidor ou apenas afeta mensagens enviadas para outros servidores faltosos, essa falta não pode ser observada pelos servidores corretos e, portanto, não pode ser detectada.

O modelo de sistema considera que cada servidor é composto por uma máquina de estado A_i e um módulo detector B_i, como mostra a figura 22. Um servidor i é correto se ele segue as especificações de ambos A_i e B_i. Caso contrário, o servidor é faltoso.

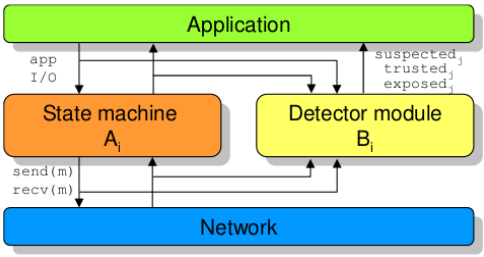

Figura 22: Comunicação entre aplicação, protocolo, e módulo do detector no servidor i. (HAEBERLEN; KOUZNETSOV; DRUSCHEL, 2006)

Os servidores se comunicam entre eles através de trocas de mensagens. É assumido que as mensagens são identificadas de uma forma única. Não existem restrições sobre o tempo de processamento e atrasos de comunicação.

Um execução E é uma sequência de eventos, de tal modo que em E, cada mensagem m é enviada e recebida no máximo uma vez, e cada $receive_i(m)$ é precedido pelo método $send_j(m)$ correspondente. São considerados distintos eventos associados com a máquina de estados A_i e eventos associados com o módulo do detector B_i.

Um histórico de um servidor i é definido como uma sequência de eventos de A_i. Considera-se que $M(E)$ denota o conjunto de mensagens recebidas pelos servidores em uma execução E. É assumido que existe um mapa histórico φ que associa cada mensagem $m \in M(E)$ com um histórico de $sender(m)$. Para um servidor correto, $\varphi(m)$ é o prefixo de execução local $E|sender(m)$ e incluindo $send(m)$. Dessa forma, para qualquer mensagem m enviada por um servidor correto, φ é válido, e para cada par de mensagens m e m' enviadas por um servidor correto, $\varphi(m)$ e $\varphi(m')$ são consistentes.

Uma mensagem m pode ser observada em E se existe um servi-

dor correto i e uma sequência de mensagens $m_1, \ldots, m_k$, da seguinte maneira:

i. $m_1 = m$.
ii. $receive(m_k) \in E|A_i$
iii. Para todo $j = 2, \ldots, k$: $receive(m_{j-1}) \in \varphi(m_j)$

Em outras palavras, m pode ser observada se precede pelo menos um evento em um servidor correto. Quando o módulo do detector B_i em um servidor correto i encontra evidência de comportamento faltoso em outro servidor j, ele envia uma indicação de falha para seu processo de aplicação local.

Existem três tipos de indicações: $trusted_j$, $suspected_j$ e $exposed_j$. Se a saída do módulo B_i é $suspected_j$, há evidência que j está ignorando certas entradas, por exemplo, ao recusar aceitar uma solicitação de serviço de um servidor correto. Se a saída de B_i é $exposed_j$, existe uma prova que j é faltoso, ou seja, o servidor j desviou da especificação de sua máquina de estado A_j. Por último, B_i gera $trusted_j$ enquanto nenhuma das outras condições se mantém.

3.2.5 Enhanced Fault-Tolerance through Byzantine Failure Detection

No modelo síncrono padrão de computação, não é possível resolver consenso ou *broadcast* na presença de faltas bizantinas se N é menor do que $3F$, onde N é o número de processos e F é o limite superior sobre o número de processos bizantinos. No modelo assíncrono, consenso e *broadcast* são impossíveis de serem resolvidos, mesmo na presença de um único processo faltoso (*crash*).

O trabalho de (BAZZI; HERLIHY, 2009) considera os problemas clássicos de acordo e *broadcast* no modelo síncrono e uma variante do modelo bizantino em que alguns processos bizantinos são em algum momento detectados e silenciados.

A proposta mostra que consenso e *broadcast* podem ser resolvidos se N, o número total de processos, é $N \geq 2f + 3F + 1$, onde f é o número de processos bizantinos transientes (detectados), e F o número de processos bizantinos permanentes (nunca detectados).

O modelo proposto considera N processos que se comunicam por troca de mensagens. Os processos arbitrários podem, por exemplo, enviar mensagens maliciosas ou faltar por *crash*. Existem, no máximo, f processos bizantinos transientes que podem ser em algum momento

silenciados (*crash*), e no máximo, F processos bizantinos permanentes que podem não ser detectados.

Processos faltosos não podem se passar por processos corretos, ou seja, processos bizantinos não podem enviar mensagens que parecem ter sido originadas por processos corretos.

No protocolo *broadcast* proposto, chamado *one-shot*, processos tentam receber um valor enviado por um *broadcaster* b. Cada vez que este protocolo é executado, todos os processos corretos recebem o mesmo valor, ou falham explicitamente na decisão, enviando $\star$. O protocolo *one-shot* deve satisfazer as seguintes propriedades:

- *Weak Validity*: se o *broadcaster* é correto e um processo correto recebe um valor diferente de $\star$, então o valor recebido é o valor do *broadcaster*.

- *Strong Validity* na ausência de faltas: se o *broadcast one-shot* é chamado pelos processos corretos em um ponto na execução antes que todos processos faltosos tenham falhado por *crash* e depois do qual não há novos *crashes*, então nenhum processo correto recebe $\star$.

- *Agreement*: todos os processos corretos recebem o mesmo valor.

- *Termination*: todo processo correto que chama o protocolo *one-shot* deve receber um valor ou falhar explicitamente em receber um valor.

O protocolo de *broadcast* apresentado é construído através de *pipeline* de execuções do algoritmo *one-shot*. A cada rodada é criado um *fork* do algoritmo *one-shot* e ao final de sua execução o valor retornado é armazenado em uma lista a. O valor recebido pelo *one-shot* será o valor do primeiro *round* a retornar um valor diferente de $\star$.

O protocolo de consenso apresentado funciona corretamente se o número total de processos N é $N > 2f + 3F$. É assumido, $N > 2f + n$, onde $n > 3F$.

Cada processo tem uma preferência atual, v_d, inicializada para seu valor de entrada e o registro de outros processos faltosos, os quais falharam para enviar uma mensagem. O protocolo prossegue em uma série de iterações. Em cada iteração, cada processo p envia por *broadcast* sua preferência e obtém os resultados.

Se F dos processos corretos concordam com um mesmo valor, p informa um *landslide* para esse valor, e utiliza esse valor como sua

preferência. Se, em vez disso, uma maioria dos processos corretos concordam com um mesmo valor, p informa uma *majority* para esse valor, e utiliza esse valor como sua preferência. Caso contrário, p deixa sua preferência inalterada.

Em seguida, p envia os resultados da primeira troca. Se F dos processos corretos concordam que houve um *landslide*, p decide esse valor. Se, em vez disso, uma maioria concorda que houve um *landslide* ou uma *majority*, então p assume esse valor como sua preferência, mas não decide.

O algoritmo tem seu término garantido desde que em algum momento, após um número suficiente de iterações, p examine o conjunto contendo os $N - F$ processos corretos.

3.3 CONSIDERAÇÕES FINAIS

Este capítulo apresentou os principais trabalhos relacionados com a abordagem proposta. A primeira abordagem RME com fins práticos descrita na literatura foi o PBFT (CASTRO; LISKOV, 1999). Apesar de prático, o PBFT exige a utilização de $3f+1$ réplicas para que se possa mascarar até f faltas. Muitas abordagens posteriores (CORREIA; NEVES; VERISSIMO, 2004; KOTLA et al., 2007; CHUN et al., 2007; VERONESE et al., 2013; DETTONI et al., 2013; DUAN et al., 2014) foram motivadas em oferecer alternativas com menor custo.

Os trabalhos (MALKHI; REITER, 1997; DOUDOU et al., 1999; KIHLSTROM; MOSER; SMITH, 2003; HAEBERLEN; KOUZNETSOV; DRUSCHEL, 2006; BAZZI; HERLIHY, 2009) relacionados com detectores de faltas foram apresentados pois seus conceitos serviram como uma base para propor uma nova abordagem RME tolerante a faltas bizantinas, a qual é intitulada como VmiBFT.

Uma comparação é apresentada na tabela 2 entre os trabalhos relacionados mostrando as principais propriedades que cada protocolo utiliza.

Tabela 2: Comparação entre as propriedades de protocolos relacionados BFT

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

[1]Caso de suspeita de falta.

4 TOLERÂNCIA A FALTAS BIZANTINAS USANDO TÉCNICAS DE INTROSPECÇÃO DE MÁQUINAS VIRTUAIS

Este trabalho propõe um algoritmo tolerante a faltas bizantinas através da técnica de Replicação de Máquina de Estados (RME), chamado VmiBFT. O algoritmo VmiBFT trabalha em conjunto com um módulo de introspecção de máquina virtual (VMI), que faz a função de um detector de intrusões. A proposta utiliza a técnica de virtualização para replicar o serviço de uma aplicação em cada máquina física em um sistema distribuído; dessa forma, cada máquina física do sistema BFT tem uma única máquina virtual que executa uma réplica do serviço.

A técnica de RME é utilizada para implementar serviços tolerantes a faltas em sistemas distribuídos. Com o uso de redundância, um serviço consegue funcionar corretamente, mesmo na presença de um número limitado de faltas (SCHNEIDER, 1990). Na literatura, podem ser encontrados diversos trabalhos que baseiam-se nessa técnica (e.g., (CASTRO; LISKOV, 1999; CORREIA; NEVES; VERISSIMO, 2004; KOTLA et al., 2007; CHUN et al., 2007; VERONESE et al., 2013; DETTONI et al., 2013; DUAN et al., 2014)).

Introspecção de máquina virtual é uma técnica utilizada para examinar, monitorar e manipular o estado de uma máquina virtual de um lugar externo à VM com o propósito de analisar o *software* executado na VM (GARFINKEL; ROSENBLUM, 2003). Vários trabalhos utilizaram essa técnica para realizar o monitoramento com o intuito de detectar intrusões nas máquinas virtuais do sistema (e.g., (GARFINKEL; ROSENBLUM, 2003; JIANG; WANG, 2007; LI et al., 2010; HAY; NANCE, 2008; XU et al., 2007)).

Nesta dissertação, é proposto um detector confiável usando os mecanismos de VMI que permitiu melhorar a resiliência do sistema BFT. O detector pode agregar diferentes técnicas utilizando VMI para detectar intrusões em uma réplica monitorada, tais como: examinar a integridade de arquivos em disco, verificar a integridade dos códigos executáveis na memória principal, incluindo aquelas técnicas citadas no parágrafo anterior.

O modelo proposto utiliza um conjunto de $2f + 1$ máquinas físicas para tolerar f faltas, sendo que cada máquina física tem sua VM correspondente, que executa o algoritmo VmiBFT, e seu detector, o qual verifica a integridade da VM localizada nessa mesma máquina. O protocolo apresentado precisa de apenas três passos de comunicação

para executar um pedido do cliente, no caso livre de faltas.

4.1 MODELO

O modelo proposto é parcialmente síncrono, ou seja, o algoritmo VmiBFT utiliza um tempo máximo Δ para receber as mensagens, porém este tempo pode variar e não é conhecido. A figura 23 apresenta esse modelo, no qual 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 VM correspondente, na qual executa um processo ou réplica do sistema e um detector de intrusão para verificar a integridade da máquina virtual.

O serviço replicado comunica-se com seu detector correspondente, o qual tem a função de verificar se a réplica permanece correta em sua especificação de acordo com as atividades que estão sendo monitoradas, tais como: processos, arquivos e conexões de rede. Logo, se ocorrer uma falha de *crash* no detector, sua réplica não conseguirá progredir na comunicação com outras réplicas do sistema BFT, pois a mensagem é enviada às outras réplicas somente depois que esta for assinada pelo seu detector. Nessa situação, consequentemente essa réplica é atingida pela falta por *crash* de seu detector. Caso essa réplica seja o líder, o protocolo de troca de visão será executado, senão, o sistema BFT continua em progressão, considerando o limite de f faltas suportadas pelo sistema.

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, através das técnicas VMI que lhe foram implantadas, analisa as informações do servidor monitorado para detectar intrusões e define se o estado do servidor está correto ou corrompido.

2. **Módulo PS**: O módulo política de segurança (PS) é utilizado para definir as regras específicas de segurança de acordo com o objetivo do detector. O módulo PS 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 VMI para serem analisados posteriormente ou armazenar *hashes* de arquivos que podem ser utilizados para verificar a integridade de arquivos e/ou execução segura de código.

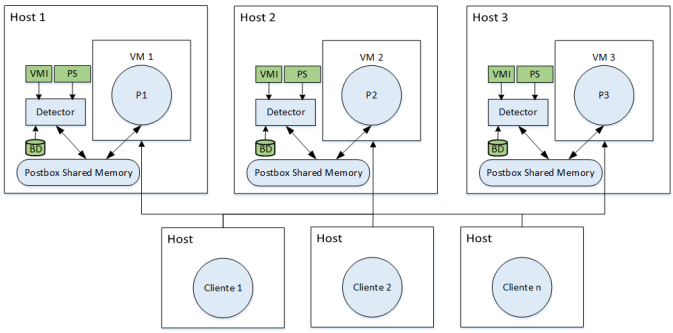

Figura 23: Modelo de Sistema para o VmiBFT.

Os papéis assumidos pelas réplicas do sistema BFT podem ser:

1. **Réplica líder (primária)**: responsável por definir uma ordem para a mensagem recebida do cliente.

2. **Réplica *backup* (seguidora)**: processa mensagens na ordem proposta pelo líder.

A réplica líder também processa a própria mensagem ordenada. Se a réplica líder é faltosa, uma das réplicas *backups* corretas será escolhida para ser o novo líder.

Para realizar a comunicação entre o detector e a máquina virtual, o sistema *host* compartilha localmente um espaço de memória para sua correspondente máquina virtual, chamada *postbox*, similar a outras propostas na literatura (DETTONI et al., 2013; STUMM et al., 2010). A comunicação entre as réplicas em diferentes *hosts* é feita através da rede, apenas por troca de mensagens. As mensagens nessa rede podem falhar, serem atrasadas, entregues fora de ordem ou duplicadas.

Para garantir autenticidade e integridade das mensagens é utilizada criptografia de chave pública (assinatura digital) (RIVEST; SHAMIR; ADLEMAN, 1978). A chave pública de cada detector é distribuída entre os detectores e os clientes para realizar a verificação da

assinatura. É assumido que uma réplica comprometida não tem acesso à chave privada do seu detector. Na prática, isso pode ser alcançado pelo isolamento do detector que é fornecido pelo VMM. É considerado também que um atacante não pode quebrar as técnicas criptográficas utilizadas como, por exemplo, resistência à colisão de *hashes*.

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 à rede WAN (*Wide Area Network*). Isto pode ser feito através de técnicas de *firewall*, 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* (STUMM et al., 2010).

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 (KIM; SPAFFORD, 1994; HAY; CID; BRAY, 2008), verificar a integridade dos códigos executáveis na memória principal (GARFINKEL; ROSENBLUM, 2003; LITTY; LIE, 2006b), interceptação de chamadas de sistema (LI et al., 2010).

4.2 DETECTOR CONFIÁVEL

Sistemas computacionais são ameaçados por atacantes que exploram vulnerabilidades presentes em sistemas operacionais para ganhar acesso privilegiado. Ao obter esse acesso, é possível conseguir informações ou até mesmo comprometer o funcionamento correto de um determinado sistema alvo. Estes sistemas também podem falhar devido a erros de *software*/*hardware*.

Sistemas de detecção de intrusões (IDS - *Intrusion Detection Systems*) são ferramentas que contribuem para melhorar a segurança de um sistema de computação. Tais sistemas monitoram continuamente a atividade do sistema, procurando ataques e evidências de intrusão. Existem duas abordagens principais para detecção de intrusão segundo Laureano et al. (LAUREANO; MAZIERO; JAMHOUR, 2007):

1. IDS baseado em rede (NIDS - *Network Intrusion Detection Systems*), os quais são baseados na observação do tráfego de rede que passa através dos sistemas para ser monitorado.

2. IDS baseado em *host* (HIDS - *Host Intrusion Detection Systems*), os quais são baseados em monitorar atividades locais no *host*, como processos, conexões de rede, chamadas de sistema, arquivos de logs, etc.

O NIDS é mais difícil de ser comprometido através de um ataque, porém não tem uma boa visibilidade do sistema monitorado. O HIDS é mais vulnerável para ataques, mas oferece uma alta visibilidade (GARFINKEL; ROSENBLUM, 2003).

O detector proposto é um componente confiável baseado nas técnicas de VMI. Através dessas técnicas, um IDS pode ser executado em local externo ao sistema monitorado, sendo mais resistente contra ataques e tendo uma boa visibilidade por obter o estado da máquina virtual através do monitor de máquina virtual (GARFINKEL; ROSENBLUM, 2003).

O detector apresentado neste trabalho utiliza uma abordagem híbrida de detecção, ou seja, ele pode ser usado para observar o tráfego de rede (NIDS) e/ou monitorar as atividades locais no servidor monitorado (HIDS). Esse módulo também pode usar *off-the-shelf IDS*, tais como: Snort (SNORT, 2015), Bro (PAXSON, 1999), OSSEC (OSSEC, 2015), Tripwire (TRIPWIRE, 2015).

Para verificar a integridade da máquina virtual, o detector pode utilizar mecanismos propostos na literatura conforme mencionados na seção 4.1, e também verificar as especificações das mensagens utilizadas pelo protocolo, tais como:

- Verificar a integridade/autenticidade;
- Verificar a ordem proposta pelo líder;
- Verificar a visão atual.

O isolamento do detector oferecido pelo VMM faz com que sua segurança seja mais robusta, protegendo o detector contra ataques, mesmo quando a máquina virtual que está sendo monitorada é comprometida após uma vulnerabilidade explorada por um atacante. Dessa forma, o detector, utilizando os recursos de VMI, consegue ter uma alta visibilidade do sistema monitorado e manter-se seguro contra ataques. Com isso, supera um IDS que não usa os benefícios oferecidos pela VMI.

O detector confiável pode ser implantado como um módulo localizado no VMM, no sistema *host* ou em outra máquina virtual, mas, nesse último local, a máquina virtual tem que ser específica, segura e com privilégios administrativos para monitorar uma VM alvo. Um exemplo de VMM que funciona dessa forma é o Xen (XEN, 2015), que possui apenas uma máquina virtual com privilégios de administrador (Dom 0), enquanto as outras VMs não são privilegiadas (NANCE; BISHOP; HAY, 2008). A figura 24 mostra os locais onde os detectores podem ser implantados.

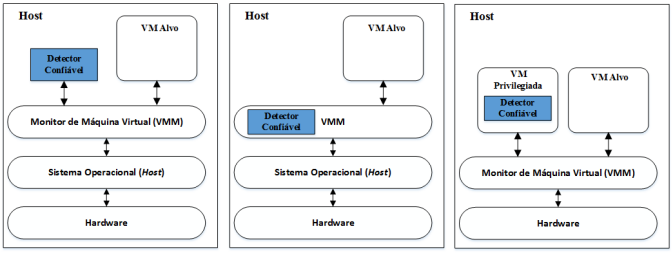

Figura 24: Localizações do detector confiável.

Um IDS baseado em VMI tem os recursos necessários para realizar tanto o monitoramento para detectar ataques quanto para tomar uma ação adequada para que um ataque não seja bem sucedido.

4.3 *POSTBOX*: MEMÓRIA COMPARTILHADA

Com base nos trabalhos realizados por (DETTONI et al., 2013; STUMM et al., 2010), que utilizam uma memória compartilhada, este trabalho também utiliza o conceito de memória compartilhada, identificada como *postbox*.

A *postbox* é utilizada pelo detector confiável e sua máquina virtual correspondente para realizar as trocas de mensagens. Essa comunicação deve ser confiável, garantida pelo VMM. O monitor de máquina virtual deve criar essa memória compartilhada através do sistema *host*, para que o detector e sua correspondente máquina virtual se comuniquem. O controle de acesso e concorrência na *postbox* são de responsabilidade do VMM.

A *postbox* deve ter uma interface que ofereça os seguintes serviços:

- ***append(value):boolean***

- ***read():Message***

A função *append* armazena um valor na *postbox*, e retorna um valor *boolean*, o que indica se a operação foi executada com sucesso ou não. A função *read* retorna a última mensagem armazenada na *postbox*. Essa interface é utilizada pelo detector e sua VM para realizar a comunicação entre eles.

4.4 PROTOCOLO PROPOSTO

Nesta seção é apresentado o algoritmo tolerante a faltas bizantinas utilizando a técnica de Replicação de Máquina de Estado (RME). Para garantir a corretude de sistemas BFT baseados na abordagem RME, as seguintes propriedades devem ser satisfeitas:

- ***Safety***: todas as réplicas corretas executam as mesmas requisições na mesma sequência.

- ***Liveness***: qualquer requisição enviada pelos clientes em algum momento será terminada, ou seja, o protocolo sempre faz progresso.

Na abordagem RME, *safety* pode ser garantida através das propriedades: estado inicial, acordo, ordem total, determinismo, conforme é explicado na subseção 4.4.1. Para garantir a propriedade *liveness* é preciso ter uma certa sincronia entre as réplicas.

O algoritmo VmiBFT provê as propriedades *safety* e *liveness*, considerando que o sistema BFT não tenha mais do que $f = [\frac{n-1}{2}]$ réplicas faltosas, ou seja, o sistema BFT é composto por uma maioria de processos corretos para executar as requisições enviadas pelos clientes.

Para dificultar que f réplicas do sistema sejam comprometidas concomitantemente, as técnicas de diversidade de *software* (AVIZIENIS; KELLY, 1984) podem ser utilizadas pelo sistema BFT. Essas técnicas permitem que réplicas falhem independentemente, ou seja, a falha de uma réplica não implica na falha de outras. Isso pode ser alcançado utilizando diferentes sistemas operacionais e diferentes versões de software entre as réplicas.

4.4.1 Algoritmo VmiBFT

O algoritmo proposto utiliza o modelo RME (SCHNEIDER, 1990) 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 (CORREIA, 2005):

- **Estado inicial**: todos os servidores corretos começam no mesmo estado.
- **Acordo**: todos os servidores corretos executam as mesmas operações.
- **Ordem total**: todos os servidores corretos 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 informar 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. Se as réplicas faltosas são as *backups*, o protocolo continua em progresso e funcionando corretamente, desde que não mais do que f réplicas *backups* sejam faltosas. O protocolo de troca de visão é explicado na subseção 4.4.3.

4.4.2 Algoritmo: caso normal de operação

Os passos do protocolo são apresentados na Figura 25. 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 ($2f+1$). Na primeira fase o cliente envia o pedido para a réplica líder, depois as réplicas iniciam a segunda fase recebendo o pedido ordenado e executando-o de acordo com a ordem e sequência correta para ser seguida; e na última fase cada réplica envia a resposta para o cliente que solicitou o serviço.

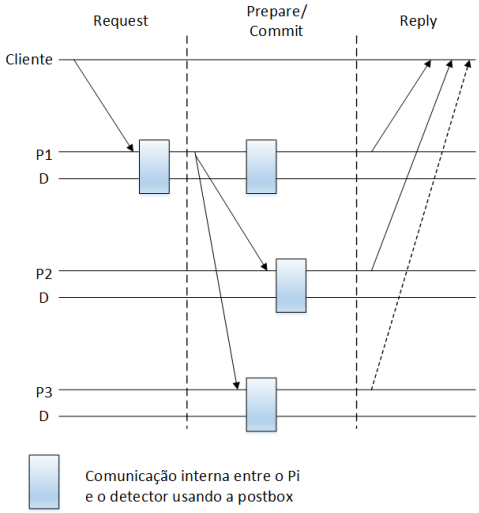

Figura 25: Passos do protocolo em execução normal com $f = 1$.

O *algorithm* 1 é descrito a seguir:

- Cliente envia a requisição para a réplica primária (linha 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).
- O líder envia para todas as réplicas *backups* a mensagem ORDER assinada pelo detector de intrusão (linha 10).
- 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).

- Cada processo aguarda o seu detector assinar a resposta (linhas 17-19).
- Cada processo envia a resposta para o cliente (linha 20), que, ao receber ao menos $f + 1$ respostas válidas[1] de diferentes réplicas, aceita a resposta.

Algorithm 1: BFT Algorithm in a normal operation case

```
1  while true do
2      msg ← receive();
3      if received(REQUEST) then
4          if is the leader then
5              n ← n + 1;
6              postbox.append(⟨ ⟨ORDER, p_i, v, n, dm⟩ , msg⟩);
7              repeat
8                  signedOrder ← getSignedOrder();
9              until receiving the asked message signed by the detector;
10             multicast(⟨ ⟨ORDER, p_i, v, n, dm,
               compromisedReplica⟩k_DetPrivate, msg⟩);
11         end
12     end
13     else if received(ORDER) then
14         if the order signed by the detector informs that the leader is
           correct then
15             reply ← execute(signedOrder);
16             postbox.append(⟨REPLY, p_i, v, seq, c, res ⟩);
17             repeat
18                 signedReply ← getSignedReply();
19             until receiving the reply signed by the detector;
20             reply_to_client(⟨REPLY, p_i, v, seq, c, res,
               compromisedReplica⟩k_DetPrivate);
21         end
22     end
23 end
```

A troca de mensagens entre o detector confiável e sua correspondente réplica usando a *postbox* é mostrada na figura 26. A réplica sendo representada pelo processo p_i envia uma mensagem na *postbox*. O detector lê a mensagem inserida pela réplica na *postbox*, realiza a operação necessária de acordo com a mensagem recebida e, em seguida, insere na *postbox* a resposta assinada com sua chave privada k. Por último, a réplica lê a resposta inserida por seu correspondente detector na *postbox*.

As mensagens assinadas pelo detector são geradas através de assinatura digital (RIVEST; SHAMIR; ADLEMAN, 1978). Cada réplica

[1] Respostas assinadas corretamente pelo detector.

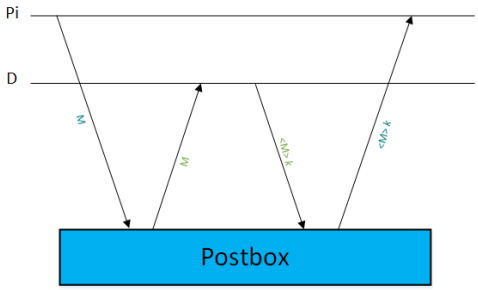

Figura 26: Comunicação interna entre p_i e o detector usando a *postbox*.

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 a chave pública do detector de cada réplica do sistema BFT para verificarem a assinatura. As instruções executadas pelo detector no *algorithm* 2 são:

1. O detector de intrusão lê a mensagem da *postbox* (linha 2).

2. Se a mensagem é ORDER e a réplica é líder, então o detector verifica a integridade da máquina virtual e em seguida concatena com a mensagem recebida uma assinatura realizada com sua chave privada informando se a máquina virtual está ou não comprometida (linhas 3-7). Se a réplica for *backup*, nesse caso, a mensagem ORDER assinada pelo líder é verificada (linhas 9-12).

3. Se a mensagem é REPLY, então o detector verifica a integridade da máquina virtual e, logo após, concatena com a mensagem recebida uma assinatura realizada com sua chave privada informando se a máquina virtual está ou não comprometida (linhas 13-17).

4.4.3 Protocolo de troca de visão

O algoritmo VmiBFT funciona dentro de uma visão, na qual um líder correto foi escolhido através da seguinte função: $novo\_lder = v \ mod \ |R|$, onde v é a nova visão, na qual os pedidos serão executados e $|R|$ corresponde ao número total de réplicas do sistema BFT. Escolher um novo líder é essencial para garantir continuidade em relação aos pedidos que o sistema BFT recebe dos clientes.

Algorithm 2: Algorithm executed by the Detector

```
1  while true do
2      msg ← postbox.read();
3      if received(ORDER) AND leader then
4          // next line returns a boolean value
5          compromisedReplica ← verifiesIntegrityOfVM();
6          postbox.append(⟨⟨ORDER, p_i, v, n, dm,
7          compromisedReplica ⟩_PrivateKey, msg⟩);
8      end
9      else
10         validSignature ← verifiesSignature(SIGNED_ORDER);
11         postbox.append(validSignature);
12     end
13     if received(REPLY) then
14         compromisedReplica ← verifiesIntegrityOfVM();
15         postbox.append(⟨REPLY, p_i, v, seq, c, res,
16         compromisedReplica⟩_PrivateKey);
17     end
18 end
```

O protocolo de troca de visão é executado quando $f + 1$ réplicas *backups* suspeitam da réplica líder como faltosa. Nessa situação, cada réplica *backup* correta, ao ser informada pelo seu detector que a réplica líder está comprometida, insere na *postbox* a mensagem $<$ VIEW_CHANGE$, p_i, v + 1 >$, sendo $v + 1$ a nova visão e p_i é a sua identificação. Cada réplica *backup* correta, ao receber a mensagem VIEW_CHANGE assinada pelo seu detector, envia a mensagem VIEW_CHANGE às outras réplicas. Outro cenário no qual o protocolo de troca de visão pode ser disparado é quando $f + 1$ réplicas *backups* recebem uma requisição diretamente do cliente e verificam que esta requisição não foi repassada previamente pelo líder.

O algoritmo de troca de visão 3 (*Algorithm 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 seu 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 receberem a mensagem válida NEW_VIEW do novo líder, aceitam a réplica como novo líder (linhas 20-24).

Algorithm 3: View change algorithm

```
1  while true do
2      msg ← receive();
3      if received(VIEW_CHANGE) then
4          if valid message then
5              // C = set of VIEW_CHANGE messages
6              C ← C ∪ msg;
7              // |C| = number of VIEW_CHANGE messages
8              if (|C| ≥ f + 1) then
9                  // |R| = number of system replicas
10                 if i = msg.v mod |R| then
11                     postbox.append(⟨NEW_VIEW, p_i, msg.v⟩);
12                     repeat
13                         signedNewView ← getSignedNewView();
14                     until receiving a new view message signed by
                       the detector;
15                     multicast(⟨NEW_VIEW, p_i, msg.v,
                       compromisedReplica⟩k_DetPrivate);
16                 end
17             end
18         end
19     end
20     else if received(NEW_VIEW) then
21         if new leader is correct then
22             v ← msg.v;
23         end
24     end
25 end
```

4.5 CORREÇÃO DO ALGORITMO VMIBFT

Esta seção demonstra que o algoritmo VmiBFT satisfaz as propriedades *safety* e *liveness*, descritas na seção 4.4. A seguir são descritos os Axiomas, Lemas e Teoremas:

Axioma 1. Apenas um número limitado de f réplicas podem falhar durante a execução do serviço. No caso em que ocorrer uma falta de *crash* no detector, sua réplica correspondente é consequentemente afetada, sendo essa incluída dentro das f faltas suportadas.

Axioma 2. Quando o detector verifica que a réplica não está correta através da verificação de integridade e/ou alguma especificação do protocolo está incorreta, sua réplica correspondente é considerada faltosa.

Axioma 3. A mensagem é considerada inválida e sua requisição não é executada por nenhuma réplica correta, quando o detector informa na mensagem que a réplica está comprometida.

Axioma 4. Existem pelo menos $f + 1$ réplicas corretas, ou seja, uma maioria de réplicas corretas.

Axioma 5. O cliente retransmite a requisição até receber uma resposta válida, caso não receba $f + 1$ respostas válidas num tempo pré-determinado Δ.

Axioma 6. Existe um tempo máximo Δ para entrega das mensagens na rede, porém esse tempo é variável e não conhecido.

Axioma 7. Não é possível uma réplica faltosa falsificar a assinatura de seu correspondente detector. Também não é possível uma réplica faltosa se passar por outra réplica, ou seja, enviar mensagens que são identificadas com *id* de outra réplica sem ser percebida.

Axioma 8. O serviço executado pelas réplicas leva para um estado equivalente entre elas, ou seja, é determinístico, e todas as réplicas iniciam o protocolo no mesmo estado.

A seguir são apresentados os Lemas de acordo com os Axiomas declarados anteriormente:

Lema 1. Toda mensagem m correta enviada por um cliente, eventualmente, será entregue, executada e respondida pelas réplicas corretas.

Prova: De acordo com os Axiomas 5 e 6, a mensagem m enviada por um cliente será entregue para as réplicas dentro de um tempo pré-determinado Δ. Se a réplica primária é faltosa, o protocolo de troca de visão será executado para escolher uma réplica correta como o novo líder. Logo, as réplicas *backups* receberão, em algum momento, a mensagem enviada por um líder correto. Em seguida, todas as réplicas corretas executam e enviam o pedido do cliente. Dessa forma, o cliente receberá, eventualmente, $f + 1$ respostas válidas.

Lema 2. Se uma réplica correta qualquer r_i executa uma requisição da mensagem m, então a requisição da mensagem m será executada por todas as réplicas corretas na mesma ordem que r_i executou a solicitação da mensagem m.

Prova: Conforme o Axioma 3, se as réplicas corretas recebem uma mensagem que não é válida, elas não executam essa mensagem. Logo, todas as réplicas corretas executam a mensagem m na mesma ordem definida. Seria uma contradição com os Axiomas 1, 2 e 3, no caso de existir réplicas corretas executando a mensagem m em ordens diferentes.

Lema 3. Se uma réplica correta r_i executa uma requisição r da mensagem m produzindo um resultado res, então pelo menos $f + 1$ réplicas corretas fornecerão a mesma resposta res.

Prova: De acordo com o Axioma 8 e o Lema 2, todas as réplicas corretas executam a requisição r na mesma ordem e mudam para um estado consistente entre elas, após a execução de r. Seria uma con-

tradição com o Axioma 4 se não houvesse pelo menos $f + 1$ réplicas corretas executando a requisição r.

Teorema 1. As réplicas corretas executam os pedidos dos clientes em uma ordem total.

Prova: Conforme os Lemas 1, 2 e 3, os pedidos enviados pelos clientes, eventualmente, serão executados na mesma ordem por $f + 1$ réplicas corretas, e suas respostas enviadas para os clientes, posteriormente. Dessa forma, todas as requisições são executadas pelas réplicas corretas em ordem total, demonstrando que o algoritmo VmiBFT satisfaz a propriedade *safety*.

Teorema 2. Os clientes, em algum momento, recebem as respostas para seus pedidos.

Prova: De acordo com o Axioma 6, uma mensagem não pode ser atrasada indefinidamente. Mesmo se f réplicas faltosas deixem de enviar ou atrasem suas respostas, existirão $f + 1$ respostas que serão enviadas pelas réplicas corretas em algum momento, conforme o Lema 1. E pelo Lema 3, os clientes receberão $f + 1$ respostas corretas de seus pedidos. Dessa forma, é demonstrado que o algoritmo proposto mantém o progresso em relação à entrega das respostas correspondentes aos pedidos dos clientes, satisfazendo a propriedade *liveness*.

4.6 CONSIDERAÇÕES FINAIS

Neste capítulo foi apresentado um novo algoritmo tolerante a faltas bizantinas, chamado VmiBFT. O algoritmo proposto utiliza a técnica de replicação de máquina de estados e faz uso da tecnologia de virtualização para replicar o serviço e também para isolar o detector da aplicação monitorada. O algoritmo VmiBFT trabalha em conjunto com um detector de intrusões que é um componente confiável do sistema.

O detector confiável é baseado nas técnicas VMI para verificar a integridade da réplica monitorada. Diversas formas de verificar a integridade de um servidor monitorado foram propostas na literatura, tais como, examinar a integridade do sistema de arquivos em disco (KIM; SPAFFORD, 1994; HAY; CID; BRAY, 2008), verificar a integridade dos códigos executáveis na memória principal (GARFINKEL; ROSENBLUM, 2003; LITTY; LIE, 2006b), interceptação de chamadas de sistema (LI et al., 2010).

O modelo de sistema apresentado utiliza um conjunto de $2f + 1$ máquinas físicas para tolerar f faltas, sendo que cada *host* tem uma única máquina virtual, na qual o serviço é replicado.

5 RESULTADOS E ANÁLISES

Este capítulo apresenta os resultados de uma avaliação experimental executada em dois cenários. No primeiro cenário, o algoritmo VmiBFT é executado de acordo com o modelo proposto, enquanto que o segundo cenário roda o algoritmo proposto sem utilizar a virtualização.

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 4.1. Para realizar e verificar a integridade dos arquivos, o detector baseou-se numa base de dados contendo os *hashes* dos arquivos monitorados.

Os experimentos foram realizados em três servidores com processador Intel®Core^{TM}i7 @ 3.50 GHz com Debian Server 7.8 e 4 GB de memória, utilizando a interface de linha de comando *VBoxManage* 4.1.18 do *VirtualBox* como VMM. No primeiro cenário, cada máquina virtual rodava o SO Ubuntu Server 12.04 LTS , uma CPU virtual e 2 GB de memória. As máquinas virtuais estavam conectadas em uma rede Ethernet 10/100.

Para realizar a avaliação, foi calculada a latência do algoritmo proposto no caso normal de operação, onde a réplica líder é correta, ao passo que as outras réplicas *backups* podem falhar, considerando o número limitado de f faltas. 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 5.000 requisições enviadas por um cliente.

5.1 CENÁRIO 1

Esta seção apresenta resultados de uma avaliação experimental da arquitetura proposta, conforme o modelo apresentado na figura 23 da seção 4.1.

Os arquivos monitorados foram armazenados em um espaço em disco visto pelo sistema hospedeiro e sua correspondente máquina vir-

tual (sistema convidado). Esse recurso é oferecido pelo VMM *VirtualBox* através do serviço *Shared Folders*.

Outro recurso oferecido pelo *VirtualBox* é a configuração dos adaptadores virtuais de rede por meio dos modos de operação na VM. Duas interfaces virtuais foram criadas, utilizando o seguinte modo de operação:

- **Modo *Bridge***: Esse modo é utilizado pela **interface de rede 1**. Nessa configuração a máquina virtual consegue conectar-se diretamente na rede externa e assim receber um endereço IP válido. O sistema hospedeiro e também todas as máquinas da rede à qual a máquina hospedeira pertence conseguirão ver normalmente a VM pela rede, como se a VM fosse uma máquina real.

- **Modo *Host Only***: Esse modo é utilizado pela **interface de rede 2**. Nessa configuração, o VMM monta uma rede contendo somente o hospedeiro e sua correspondente máquina virtual, sem a necessidade do adaptador de rede físico do hospedeiro. O *VirtualBox* cria no sistema hospedeiro uma interface virtual de rede, semelhante à interface de *loopback*. Esta interface proporciona a conectividade entre a VM e o sistema hospedeiro. O sistema hospedeiro e sua correspondente VM serão vistos como se estivessem conectados a uma mesma rede física, porém, como a rede interna está conectada somente à interface virtual do hospedeiro, o acesso à rede externa não é possível.

A *postbox* foi implementada utilizando a interface de rede 2 (*Host Only*), na qual o detector e a máquina virtual se comunicam através do protocolo TCP/IP. A *postbox* somente pode ser acessada pelo detector e sua correspondente máquina virtual. As máquinas da rede à qual a máquina hospedeira pertence e outras máquinas da rede externa não terão acesso a comunicação realizada pela *postbox*, mesmo se conhecessem o endereço pelo qual o detector e sua correspondente VM se comunicam.

O algoritmo VmiBFT foi executado em diferentes implementações, conforme mostra a tabela 3.

Em seguida é detalhado o que cada tipo de implementação representa:

- Sem *overhead*: o algoritmo proposto é executado sem a implementação dos seguintes serviços: comunicação entre o detector e sua correspondente máquina virtual através da *postbox*, verificação de integridade realizada pelo detector e assinatura digital.

Tabela 3: Implementações realizadas utilizando o algoritmo VmiBFT

	Processa mensagens	**Serviço *postbox***	**Serviço detector**	**Serviço assinatura**
Sem *overhead*	✓	-	-	-
Overhead postbox	✓	✓	-	-
***Overhead* detector**	✓	✓	✓	-
***Overhead* assinatura**	✓	✓	-	✓
Overhead full	✓	✓	✓	✓

- *Overhead postbox*: o algoritmo proposto é executado sem a implementação dos seguintes serviços: verificação de integridade realizada pelo detector e assinatura digital.
- *Overhead* detector: o algoritmo proposto é executado sem a implementação do serviço de assinatura digital.
- *Overhead* assinatura: o algoritmo proposto é executado sem a implementação do serviço de verificação de integridade realizada pelo detector.
- *Overhead full*: o algoritmo proposto é executado com a implementação de todos os serviços mencionados anteriormente.

Através das informações obtidas, pode-se observar que a latência de 14,94 ms obtida pela execução do algoritmo VmiBFT com todos os serviços (*overhead full*) vem de grande parte da sobrecarga do serviço de assinatura digital; outra parte é obtida da sobrecarga do serviço realizado pelo detector; e, por último, o *overhead* da *postbox* representou um acréscimo menor no resultado da latência. Na figura 28 é mostrado o histograma da latência referente ao algoritmo VmiBFT (*overhead full*), sendo o desvio padrão σ dessa distribuição igual a 1,87 e coeficiente de variação igual a 12,5 %. A maioria dos pedidos teve latência entre 14 e 15 ms.

A porcentagem de sobrecarga obtida de cada serviço, com base no resultado de latência de 4,3 ms da execução do algoritmo proposto sem *overhead* é mostrada na figura 29. Comparando com o resultado de latência sem *overhead*, a *postbox* teve uma sobrecarga de 47,91 %, procedendo o detector com 71,63 % e representando o maior *overhead* com o valor de 227,91 % está o serviço de assinatura digital.

É importante lembrar que o resultado de latência pode variar dependendo do tipo de verificação realizada pelo detector e também da implementação da *postbox*. Otimizações podem ser feitas para melhorar o resultado obtido de latência como, por exemplo, utilizar criptografia

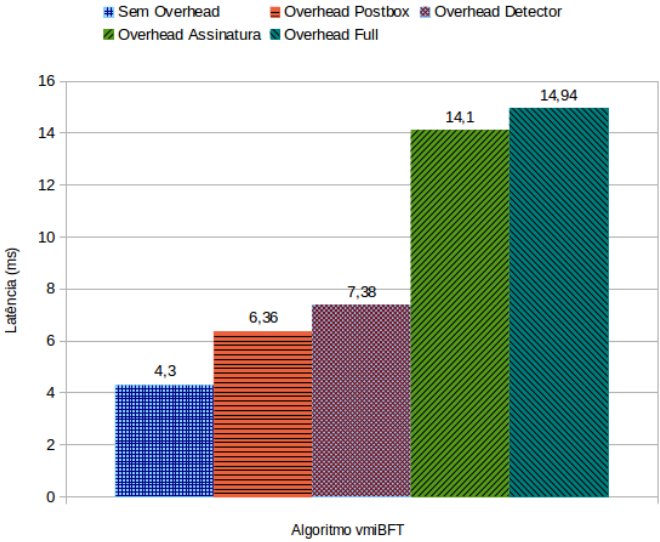

Figura 27: Latência em operação normal.

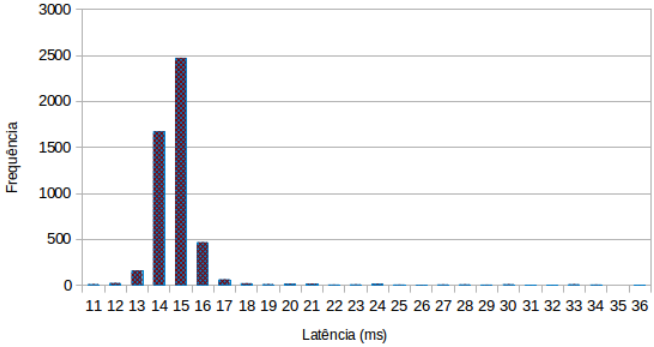

Figura 28: Histograma da latência do algoritmo vmiBFT (*overhead full*).

assimétrica entre as réplicas com seu correspondente detector e criptografia simétrica entre as réplicas e os clientes.

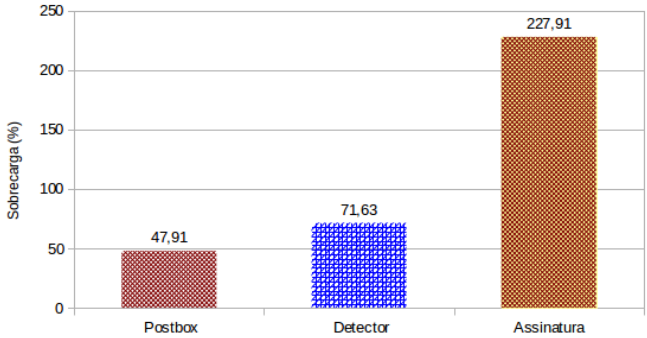

Figura 29: Sobrecarga dos módulos do algoritmo proposto.

5.2 CENÁRIO 2

Para calcular a sobrecarga que a virtualização apresentou no algoritmo proposto, foram executados os mesmos experimentos realizados no cenário 1, mas sem utilizar a virtualização, como mostra a figura 30. Neste cenário, o sistema BFT é vulnerável a ser comprometido, pois um atacante tem acesso direto ao *host*. Como o detector é executado na máquina física e não tem o isolamento fornecido pela virtualização, esse módulo pode ter seu funcionamento alterado por um ataque bem sucedido, logo, o detector não é um componente confiável do sistema.

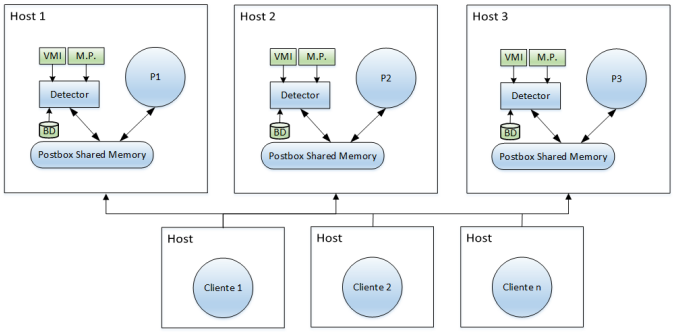

Figura 30: Modelo sem virtualização.

Conforme era esperado, a execução do algoritmo proposto em um

ambiente sem virtualização apresentou um desempenho melhor comparado ao cenário que utiliza os recursos da tecnologia de virtualização. O resultado da latência em cada tipo de implementação é exibido na figura 31. A figura 32 mostra a relação entre frequência e latência para o algoritmo VmiBFT (*overhead full*). A maioria dos pedidos teve latência entre 10 e 11 ms. O desvio padrão σ dessa distribuição é igual a 0,82 e coeficiente de variação igual a 8,0 %.

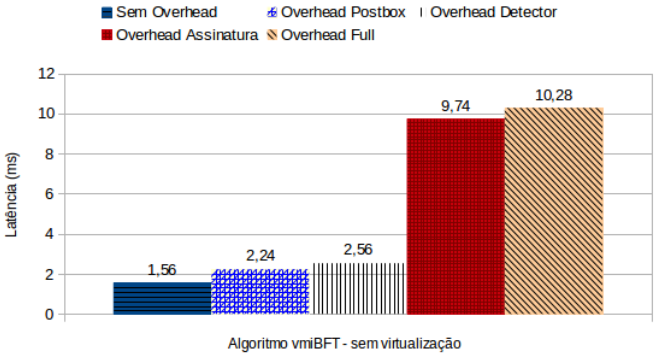

Figura 31: Latência em operação normal.

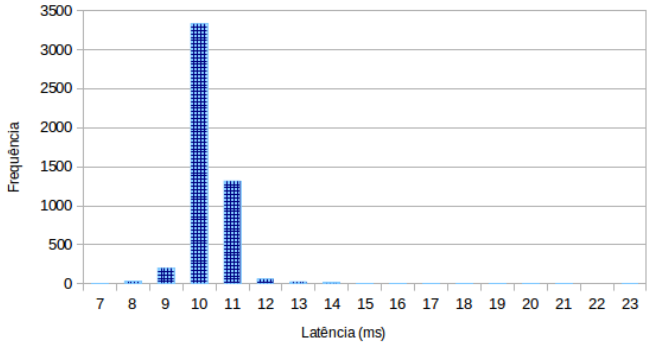

Figura 32: Histograma da latência do algoritmo vmiBFT (*overhead full*) - cenário 2 .

A porcentagem obtida do valor adicional de cada serviço executado em um ambiente virtualizado mostrado na figura 27, com base nos

resultados do cenário 2 ilustrado na figura 31, é mostrada na figura 33. A partir desses resultados pode se observar que a tecnologia de virtualização utilizada nos experimentos evidentemente incorre num custo adicional, de modo que o serviço do detector é o que apresenta maior perda de desempenho, seguido da *postbox*, e, por último, o serviço de assinatura. A perda de desempenho no serviço do detector e *postbox* pode ser explicada devido as operações realizadas de E/S. A virtualização de processamento é bem gerida pelas CPUs modernas, mas a virtualização de E/S ainda é majoritariamente tratada por *software*.

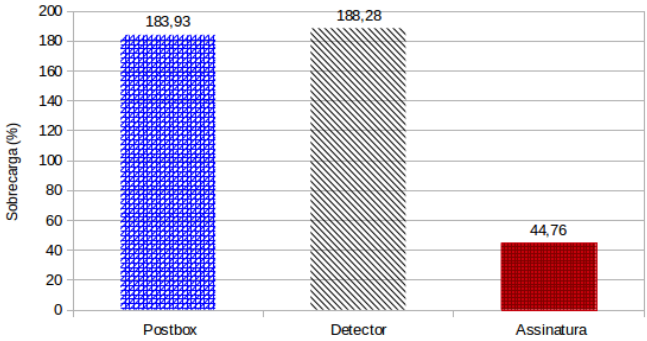

Figura 33: Sobrecarga da virtualização.

5.3 COMPARAÇÕES DE NOSSAS PROPOSIÇÕES COM TRABALHOS RELACIONADOS

Uma comparação é apresentada na tabela 4 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 penúltima coluna. A proposta, além de contribuir para que se 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.

O PBFT foi o primeiro trabalho prático de replicação de máquina de estado tolerante a faltas bizantinas. No PBFT, foi utilizado um esquema eficiente de autenticação que consiste em usar códigos de

Tabela 4: 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	Comp. confiável: *Hardware/Software*
PBFT	$3f+1$	$3f+1$	5	$2n^2-n+1$	não	-
Zyzzyva	$3f+1$	$3f+1$	$3\ ;\ 5$ [1]	$(2n);(4n)$[1]	não	-
BFT-T0	$2f+1$	$2f+1$	5	$2n$	sim	*Hardware*
A2M-PBFT-EA	$2f+1$	$2f+1$	5	$2n^2-n+1$	sim	*Hardware/Software*
MinBFT	$2f+1$	$2f+1$	4	n^2+2n-1	sim	*Hardware/Software*
MinZyzzyva	$2f+1$	$2f+1$	$3\ ;\ 5$[1]	$(3n-1);(5n-1)$[1]	sim	*Hardware/Software*
TwinBFT	$2f+1$	$4f+2$	3	$5n-2$	não	-
ByzID	$2f+1$	$2f+1$	3	$2n+n-1$	sim	*Hardware/Software*
VmiBFT	$2f+1$	$2f+1$	3	$2n$	sim	*Software*

[1]Caso de suspeita de falta.

autenticação de mensagem (MACs) durante operação normal e criptografia de chave pública apenas quando houver faltas.

O trabalho Zyzzyva propôs um algoritmo especulativo para diminuir o número de passos do protocolo BFT no caso livre de faltas, porém quando há suspeita de falta, o protocolo precisa de 5 passos para completar o pedido.

O BFT-TO propôs um componente confiável através da utilização de um *hardware* confiável para diminuir o número de réplicas do sistema BFT para $2f+1$. Mas, na prática, o componente foi implementado como um módulo do *kernel*. No BFT-TO, foram utilizadas duas placas de rede em cada servidor, adicionando um custo a mais para implantar o sistema BFT. O componente confiável fornece um serviço de ordem.

No trabalho A2M foi proposto um componente confiável para projetar protocolos livres de equívocos que permitiu reduzir o número de réplicas para $2f+1$. Este componente confiável pode ser executado em um *hardware* confiável ou garantir sua segurança por meio da virtualização. Porém, na prática, foi implementado como uma biblioteca no mesmo espaço de endereçamento do protocolo BFT.

Nos trabalhos MinBFT e MinZyzzyva, o componente confiável proposto atribui às mensagens identificadores únicos e sequenciais. Esse componente pode ser baseado em *hardware* confiável como *trusted platform module* (TPM) ou utilizando virtualização. Com a utilização deste componente foi reduzido o número de réplicas para $2f+1$. Os algoritmos MinBFT e MinZyzzyva têm o número de passos de comunicação igual a 4 e 3, respectivamente, para executar um pedido. Porém, o algoritmo MinZyzzyva é especulativo e, quando há suspeita de falta, são necessários 5 passos para executar o pedido.

O TwinBFT utilizou virtualização para reduzir o número de réplicas e passos de comunicação. Isso foi possível através da utilização

de um conjunto de máquinas virtuais gêmeas em cada servidor, onde cada máquina virtual atua como um detector de falhas para sua gêmea.

No trabalho ByzID foi proposto um componente confiável que pode ser baseado em *hardware* confiável ou através de virtualização. Contudo, na prática, o componente e o protocolo BFT foram executados como processos sob o mesmo SO. Através desse componente, que verifica o tráfego de mensagens de rede para detectar desvio da especificação correta do protocolo, a proposta apresentou um algoritmo BFT que pode ser executado em $2f + 1$ réplicas e utilizar 3 passos para executar um pedido.

A principal contribuição do trabalho VmiBFT é propor um modelo que utiliza um componente confiável através das técnicas de introspecção de máquina virtual (VMI). Este componente pode ser usado para verificar a integridade da máquina virtual (exemplos de verificações são mencionadas na seção 4.1) e também para analisar as especificações das mensagens utilizadas pelo protocolo. O componente proposto utiliza virtualização para garantir sua segurança, porém o modelo apresentado poderia ser adaptado facilmente para executar este componente em um *hardware* confiável como é feito em alguns trabalhos relacionados. Outra contribuição deste trabalho é poder agregar diversos tipos de verificações de integridade que se baseiem nas técnicas VMI para serem utilizadas pelo componente confiável. Através desta abordagem, foi possível melhorar a resiliência do sistema BFT para utilizar $2f + 1$ réplicas e 3 passos para executar um pedido.

Diferentemente do trabalho TwinBFT, o VmiBFT utiliza apenas uma máquina virtual em cada servidor. Dessa forma, o número de processos do sistema BFT deste trabalho é menor comparando-se com o TwinBFT. Na prática, os trabalhos ByzID, BFT-TO e A2M utilizaram uma implementação que não garante a segurança do sistema BFT, pois o componente confiável pode ser subvertido por meio de um ataque bem sucedido. O VmiBFT apresentou um modelo baseado na tecnologia de virtualização para separar o componente confiável do protocolo BFT e, na prática, o algoritmo proposto foi executado conforme seu modelo apresentado. Além disso, diferentemente dos algoritmos MinZyzzyva, Zyzzyva, o algoritmo VmiBFT não é especulativo e apresenta um número de passos menor para executar um pedido se comparado com o MinBFT. Diferentemente do BFT-TO, este trabalho não precisa de *hardware* adicional, apresentando um custo menor. Comparando o VmiBFT com o PBFT, o trabalho proposto utiliza um número menor de réplicas e passos de comunicação para executar um pedido.

Entre os trabalhos relacionados que foram analisados, o ByzID

(DUAN et al., 2014) é o mais similar à proposta neste trabalho. Devido a isso, será dada uma ênfase na comparação, apresentando os pontos comuns e as diferenças entre esses trabalhos.

Os principais pontos comuns entre VmiBFT e ByzID são:

- Propuseram uma abordagem unificada que utiliza detecção de intrusões como componente confiável do sistema BFT para melhorar sua resiliência.
- O detector de intrusões não protege contra todos ataques possíveis, somente aqueles que o IDS pode detectar.
- O IDS monitora a réplica primária para garantir a ordem correta das mensagens.
- Cada réplica é associada com um componente IDS separado que atua como um oráculo.
- O modelo de sistema utiliza $2f + 1$ réplicas e o protocolo utiliza 3 passos de comunicação para executar um pedido.
- Os IDSs podem falhar por *crash*.

As principais diferenças entre VmiBFT e ByzID são:

- **ByzID**: IDS baseado em rede, utilizando a detecção para verificar se as mensagens estão em conformidade com as especificações definidas. **VmiBFT**: IDS baseado em *host* e rede, utilizando a detecção para verificar as especificações através das mensagens do protocolo e integridade da réplica.
- **ByzID**: A réplica primária é raiz e as *backups* são folhas. Uma réplica *backup* somente envia ou recebe mensagens da primária. **VmiBFT**: A réplica primária e as *backups* estão no mesmo nível. Uma réplica *backup* pode enviar e receber mensagens tanto da réplica primária como de outra réplica *backup*.
- **ByzID**: Mesmo se um IDS tem uma parada, sua réplica RME pode continuar a processar pedidos. **VmiBFT**: Se um IDS tem uma parada, sua réplica RME não pode processar mais pedidos.
- **ByzID**: Número de mensagens de rede para executar um pedido é igual a 8 para $f = 1$. **VmiBFT**: Número de mensagens de rede para executar um pedido é igual a 6 para $f = 1$.

- **ByzID**: O IDS não modifica as mensagens, ele apenas captura os pacotes de rede do protocolo e analisa se estão em conformidade com a especificação. **VmiBFT**: O IDS modifica as mensagens, por exemplo, o IDS adiciona na mensagem a informação se a réplica está comprometida.

Foi calculado, com base na informação da coluna número de mensagens da tabela 4, o número de trocas de mensagens necessárias para executar uma operação no caso normal, conforme é apresentado na figura 34. 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 ByzID o número de trocas de mensagens adicionais é 6 e no PBFT em cada configuração consecutiva o número de mensagens acrescentadas é cada vez maior.

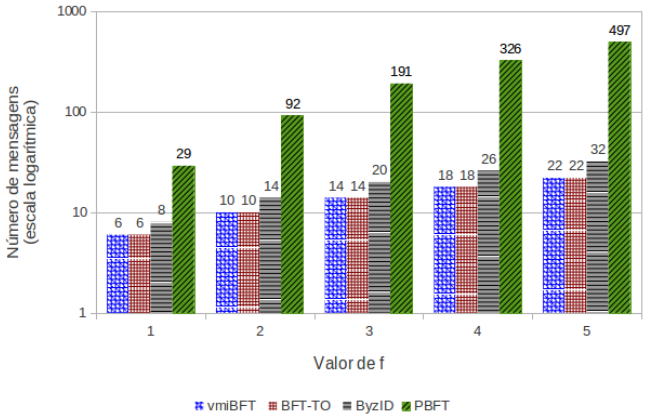

Figura 34: Número de trocas de mensagens, para f = 1,2,...,5.

5.4 CONSIDERAÇÕES FINAIS

Neste capítulo foi mostrada uma abordagem experimental utilizando a verificação de integridade de arquivos em disco como um tipo de detecção para ser usada na arquitetura. O algoritmo VmiBFT foi executado em diferentes implementações: sem *overhead*, *overhead post-box*, *overhead* detector, *overhead assinatura* e *overhead full*.

Também foram realizadas análises dos resultados obtidos da implementação do algoritmo proposto em dois cenários: com virtualização e sem virtualização. No primeiro cenário foi possível calcular a sobrecarga dos serviços utilizados (*postbox*, detector, assinatura) pelo algoritmo VmiBFT. E no segundo cenário foi calculada a sobrecarga que a virtualização causa no algoritmo proposto. Pode se observar que a virtualização incorre num custo adicional, de modo que o serviço do detector e *postbox* tiveram uma perda maior de desempenho.

E, finalmente, comparações foram realizadas entre o algoritmo proposto e trabalhos relacionados analisando as suas principais propriedades.

6 CONCLUSÕES E PERSPECTIVAS FUTURAS

O funcionamento correto de sistemas é fundamental, principalmente para sistemas críticos. Entretanto, sistemas computacionais podem ser comprometidos por meio de ataques bem sucedidos e por erros de *software*/*hardware*.

Uma das técnicas utilizadas para melhorar a segurança de um sistema de computação é replicação de máquina de estados (RME). Nesta abordagem, assume-se que o sistema funciona de acordo com sua especificação, até mesmo na presença de um número limitado de servidores comprometidos.

O primeiro algoritmo de replicação de máquina de estados que pode ser aplicado em sistemas reais, conhecido também como PBFT, foi proposto por Castro e Liskov (CASTRO; LISKOV, 1999); neste modelo são necessários $3f + 1$ servidores para tolerar f servidores faltosos e 5 passos são necessários para execução do protocolo sem falhas.

Após a publicação do PBFT por Castro e Liskov, diversos trabalhos foram propostos para diminuir o custo da abordagem RME. Alguns trabalhos otimizaram o algoritmo de replicação de máquina de estados para utilizar $2f+1$ servidores, melhorando a resiliência do sistema BFT e/ou reduziram o número de passos do protocolo (CORREIA; NEVES; VERISSIMO, 2004; CHUN et al., 2007; VERONESE et al., 2013; DETTONI et al., 2013; DUAN et al., 2014). Na proposta de (KOTLA et al., 2007) foi utilizada uma abordagem diferente e o número de passos de comunicação foi reduzido, apesar de manter o número de réplicas.

Este trabalho apresentou um modelo e um algoritmo tolerante a faltas bizantinas através de replicação de máquina de estados. A proposta faz uso da virtualização e de um componente confiável identificado como detector no modelo BFT. O modelo proposto une a abordagem RME com IDS e utiliza virtualização para replicar o serviço. 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.

O detector baseado nas técnicas 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 expe-

rimental 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.

Foram realizadas análises dos resultados obtidos da implementação do algoritmo proposto em dois cenários. No primeiro cenário foi possível calcular a sobrecarga dos serviços utilizados (*postbox*, detector, assinatura) pelo algoritmo VmiBFT. E no segundo cenário foi calculada a sobrecarga que a virtualização causa no algoritmo proposto.

Uma comparação entre o algoritmo proposto e trabalhos relacionados foi realizada para analisar as principais propriedades que cada algoritmo utiliza em sua proposta.

Este trabalho apresenta alguns pontos que não foram explorados. A seguir são mencionados possíveis trabalhos futuros que podem ser feitos para aperfeiçoamento das soluções propostas:

- Implementação da *postbox* utilizando sistemas de arquivos para armazenar/ler as mensagens. Calcular o *overhead* que esta implementação incide no algoritmo proposto.

- Detector utilizar a verificação de integridade de instruções na memória volátil em tempo de execução como proposto por (LITTY; LIE, 2006a) em conjunto com a verificação de integridade em disco usada no trabalho.

- Calcular a sobrecarga que a virtualização causa no algoritmo VmiBFT utilizando outro monitor de máquina virtual.

- Utilizar criptografia assimétrica entre as réplicas com seu correspondente detector e criptografia simétrica entre as réplicas e os clientes e comparar o desempenho obtido com a solução encontrada no trabalho.

- Elaboração e implementação de um protocolo de recuperação da réplica, quando a mesma for verificada como comprometida pelo seu correspondente detector.

Além deste documento, o desenvolvimento deste trabalho resultou na publicação do seguinte artigo: "Replicação de Máquinas de Estado Usando Detectores de Intrusão e Virtualização" - IV Simpósio Brasileiro de Engenharia de Sistemas Computacionais - SBESC 2014 (MORAIS et al., 2014).

REFERÊNCIAS

AVIZIENIS, A.; KELLY, J. Fault tolerance by design diversity: Concepts and experiments. *Computer*, v. 17, n. 8, p. 67–80, Aug 1984. ISSN 0018-9162.

BARBORAK, M.; DAHBURA, A.; MALEK, M. The consensus problem in fault-tolerant computing. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 25, n. 2, p. 171–220, jun. 1993. ISSN 0360-0300. <http://doi.acm.org/10.1145/152610.152612>.

BAZZI, R. A.; HERLIHY, M. Enhanced fault-tolerance through byzantine failure detection. In: *Proceedings of the 13th International Conference on Principles of Distributed Systems*. Berlin, Heidelberg: Springer-Verlag, 2009. (OPODIS '09), p. 129–143. ISBN 978-3-642-10876-1. <http://dx.doi.org/10.1007/978-3-642-10877-8_12>.

BESSANI, A. N.; FRAGA, J. da S.; LUNG, L. C. Bts: A byzantine fault-tolerant tuple space. In: *Proceedings of the 2006 ACM Symposium on Applied Computing*. New York, NY, USA: ACM, 2006. (SAC '06), p. 429–433. ISBN 1-59593-108-2. <http://doi.acm.org/10.1145/1141277.1141377>.

CASTRO, M.; LISKOV, B. Practical byzantine fault tolerance. In: *Proceedings of the Third Symposium on Operating Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 1999. (OSDI '99), p. 173–186. ISBN 1-880446-39-1. <http://dl.acm.org/citation.cfm?id=296806.296824>.

CHANDRA, T. D.; TOUEG, S. Unreliable failure detectors for reliable distributed systems. *J. ACM*, ACM, New York, NY, USA, v. 43, n. 2, p. 225–267, mar. 1996. ISSN 0004-5411. <http://doi.acm.org/10.1145/226643.226647>.

CHUN, B.-G. et al. Attested append-only memory: Making adversaries stick to their word. *SIGOPS Oper. Syst. Rev.*, ACM, New York, NY, USA, v. 41, n. 6, p. 189–204, out. 2007. ISSN 0163-5980. <http://doi.acm.org/10.1145/1323293.1294280>.

CORREIA, M. Serviços distribuídos tolerantes a intrusões: resultados recentes e problemas abertos. In: GASPARY, L. P.; SIQUEIRA, F. (Ed.). *SBSeg 2005, V Simpósio Brasileiro em Segurança da Informação*

e de Sistemas Computacionais, Livro Texto dos Minicursos. [S.l.]: Sociedade Brasileira de Computação, 2005. p. 113–162.

CORREIA, M.; NEVES, N. F.; VERISSIMO, P. How to tolerate half less one byzantine nodes in practical distributed systems. In: *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems*. Washington, DC, USA: IEEE Computer Society, 2004. (SRDS '04), p. 174–183. ISBN 0-7695-2239-4. <http://dl.acm.org/citation.cfm?id=1032662.1034362>.

CORREIA, M.; NEVES, N. F.; VERÍSSIMO, P. BFT-TO: intrusion tolerance with less replicas. *Comput. J.*, v. 56, n. 6, p. 693–715, 2013. <http://dx.doi.org/10.1093/comjnl/bxs148>.

CORREIA, M.; VERONESE, G. S.; LUNG, L. C. Asynchronous byzantine consensus with 2f+1 processes. In: *Proceedings of the 2010 ACM Symposium on Applied Computing*. New York, NY, USA: ACM, 2010. (SAC '10), p. 475–480. ISBN 978-1-60558-639-7. <http://doi.acm.org/10.1145/1774088.1774187>.

CRISTIAN, F. et al. Atomic broadcast: From simple message diffusion to byzantine agreement. In: *Information and Computation*. [S.l.: s.n.], 1985. p. 200–206.

DETTONI, F. et al. Byzantine fault-tolerant state machine replication with twin virtual machines. In: *Computers and Communications (ISCC), 2013 IEEE Symposium on*. [S.l.: s.n.], 2013. p. 000398–000403.

DOUDOU, A. et al. *Muteness Failure Detectors: Specification and Implementation*. 1999.

DUAN, S. et al. Byzantine Fault Tolerance from Intrusion Detection. In: *Proceedings of the 33rd IEEE International Symposium on Reliable Distributed Systems (SRDS)*. Nara, Japan: [s.n.], 2014.

FAVARIM, F. et al. Exploiting tuple spaces to provide fault-tolerant scheduling on computational grids. In: *Object and Component-Oriented Real-Time Distributed Computing, 2007. ISORC '07. 10th IEEE International Symposium on*. [S.l.: s.n.], 2007. p. 403–411.

GARAY, J. A.; MOSES, Y. Fully Polynomial Byzantine Agreement for Processors in Rounds. *SIAM Journal on Computing*, v. 27, n. 1, p. 247–290, 1998. <http://dx.doi.org/10.1137/S0097539794265232>.

GARFINKEL, T.; ROSENBLUM, M. A virtual machine introspection based architecture for intrusion detection. In: *In Proc. Network and Distributed Systems Security Symposium*. [S.l.: s.n.], 2003. p. 191–206.

HADZILACOS, V.; TOUEG, S. *A Modular Approach to Fault-Tolerant Broadcasts and Related Problems*. [S.l.], 1994.

HAEBERLEN, A.; KOUZNETSOV, P.; DRUSCHEL, P. The case for byzantine fault detection. In: *Proceedings of the 2Nd Conference on Hot Topics in System Dependability - Volume 2*. Berkeley, CA, USA: USENIX Association, 2006. (HOTDEP'06), p. 5–5. <http://dl.acm.org/citation.cfm?id=1251014.1251019>.

HAY, A.; CID, D.; BRAY, R. *OSSEC Host-Based Intrusion Detection Guide*. [S.l.]: Syngress Publishing, 2008. ISBN 159749240X, 9781597492409.

HAY, B.; NANCE, K. Forensics examination of volatile system data using virtual introspection. *SIGOPS Oper. Syst. Rev.*, ACM, New York, NY, USA, v. 42, n. 3, p. 74–82, abr. 2008. ISSN 0163-5980. <http://doi.acm.org/10.1145/1368506.1368517>.

JIANG, X.; WANG, X. "out-of-the-box"monitoring of vm-based high-interaction honeypots. In: *Proceedings of the 10th International Conference on Recent Advances in Intrusion Detection*. Berlin, Heidelberg: Springer-Verlag, 2007. (RAID'07), p. 198–218. ISBN 3-540-74319-7, 978-3-540-74319-4. <http://dl.acm.org/citation.cfm?id=1776434.1776450>.

KIHLSTROM, K.; MOSER, L.; MELLIAR-SMITH, P. The securering protocols for securing group communication. In: *System Sciences, 1998., Proceedings of the Thirty-First Hawaii International Conference on*. [S.l.: s.n.], 1998. v. 3, p. 317–326 vol.3.

KIHLSTROM, K. P.; MOSER, L. E.; SMITH, P. M. M. Byzantine fault detectors for solving consensus. *The Computer Journal*, v. 46, n. 1, p. 16–35, 2003. <http://comjnl.oxfordjournals.org/content/46/1/16.abstract>.

KIM, G. H.; SPAFFORD, E. H. The design and implementation of tripwire: A file system integrity checker. In: *Proceedings of the 2Nd ACM Conference on Computer and Communications Security*. New York, NY, USA: ACM, 1994. (CCS '94), p. 18–29. ISBN 0-89791-732-4.

KOTLA, R. et al. Zyzzyva: Speculative byzantine fault tolerance. *SIGOPS Oper. Syst. Rev.*, ACM, New York, NY, USA, v. 41, n. 6, p. 45–58, out. 2007. ISSN 0163-5980. <http://doi.acm.org/10.1145/1323293.1294267>.

KVM. *VMM: KVM*. 2015. <http://www.linux-kvm.org/page/Main_Page>.

LAMPORT, L. The implementation of reliable distributed multiprocess systems. *Computer Networks (1976)*, v. 2, n. 2, p. 95–114, maio 1978. <http://dx.doi.org/10.1016/0376-5075(78)90045-4>.

LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, ACM, New York, NY, USA, v. 21, n. 7, p. 558–565, jul. 1978. ISSN 0001-0782. <http://doi.acm.org/10.1145/359545.359563>.

LAMPORT, L. Using time instead of timeout for fault-tolerant distributed systems. *ACM Trans. Program. Lang. Syst.*, ACM, New York, NY, USA, v. 6, n. 2, p. 254–280, abr. 1984. ISSN 0164-0925. <http://doi.acm.org/10.1145/2993.2994>.

LAMPORT, L.; SHOSTAK, R.; PEASE, M. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, ACM, New York, NY, USA, v. 4, n. 3, p. 382–401, jul. 1982. ISSN 0164-0925. <http://doi.acm.org/10.1145/357172.357176>.

LARANJEIRA, L.; MALEK, M.; JENEVEIN, R. On tolerating faults in naturally redundant algorithms. In: *Reliable Distributed Systems, 1991. Proceedings., Tenth Symposium on*. [S.l.: s.n.], 1991. p. 118–127.

LAUREANO, M.; MAZIERO, C.; JAMHOUR, E. Intrusion detection in virtual machine environments. In: *Proceedings of the 30th EUROMICRO Conference*. Washington, DC, USA: IEEE Computer Society, 2004. (EUROMICRO '04), p. 520–525. ISBN 0-7695-2199-1. <http://dx.doi.org/10.1109/EUROMICRO.2004.48>.

LAUREANO, M.; MAZIERO, C.; JAMHOUR, E. Protecting host-based intrusion detectors through virtual machines. *Comput. Netw.*, Elsevier North-Holland, Inc., New York, NY, USA, v. 51, n. 5, p. 1275–1283, abr. 2007. ISSN 1389-1286. <http://dx.doi.org/10.1016/j.comnet.2006.09.007>.

LI, B. et al. A vmm-based system call interposition framework for program monitoring. In: *Proceedings of the 2010 IEEE*

16th International Conference on Parallel and Distributed Systems. Washington, DC, USA: IEEE Computer Society, 2010. (ICPADS '10), p. 706–711. ISBN 978-0-7695-4307-9. <http://dx.doi.org/10.1109/ICPADS.2010.53>.

LITTY, L.; LIE, D. Manitou: A layer-below approach to fighting malware. In: *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*. New York, NY, USA: ACM, 2006. (ASID '06), p. 6–11. ISBN 1-59593-576-2. <http://doi.acm.org/10.1145/1181309.1181311>.

LITTY, L.; LIE, D. Manitou: A layer-below approach to fighting malware. In: *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*. [S.l.]: ACM, 2006.

LUIZ, A.; LUNG, L. C.; CORREIA, M. Byzantine fault-tolerant transaction processing for replicated databases. In: *Network Computing and Applications (NCA), 2011 10th IEEE International Symposium on*. [S.l.: s.n.], 2011. p. 83–90.

LUIZ, A. F.; LUNG, L. C.; CORREIA, M. Mitra: Byzantine fault-tolerant middleware for transaction processing on replicated databases. *SIGMOD Rec.*, ACM, New York, NY, USA, v. 43, n. 1, p. 32–38, maio 2014. ISSN 0163-5808. <http://doi.acm.org/10.1145/2627692.2627699>.

MALKHI, D.; REITER, M. Unreliable intrusion detection in distributed computations. In: *Proceedings of the 10th IEEE Workshop on Computer Security Foundations*. Washington, DC, USA: IEEE Computer Society, 1997. (CSFW '97), p. 116–. ISBN 0-8186-7990-5. <http://dl.acm.org/citation.cfm?id=794197.795085>.

MICROSOFT. *VMM: Microsoft Hyper-V*. 2015. <https://www.microsoft.com/Hyper-V>.

MICROSOFT. *VMM: Virtual PC/Server*. 2015. <http://www.microsoft.com/windowsserversystem/virtualserver/>.

MORAIS, P. et al. Replicação de máquinas de estado usando detectores de intrusão e virtualização. In: *Proceedings of the Brazilian Symposium on Computing Systems Engineering' 2014*. [S.l.]: SBESC, 2014.

NANCE, K.; BISHOP, M.; HAY, B. Virtual machine introspection: Observation or interference? *IEEE Security and Privacy*, IEEE Educational Activities Department, Piscataway, NJ, USA, v. 6, n. 5, p. 32–37, set. 2008. ISSN 1540-7993. <http://dx.doi.org/10.1109/MSP.2008.134>.

ONOUE, K.; OYAMA, Y.; YONEZAWA, A. Control of system calls from outside of virtual machines. In: *Proceedings of the 2008 ACM Symposium on Applied Computing*. New York, NY, USA: ACM, 2008. (SAC '08), p. 2116–1221. ISBN 978-1-59593-753-7. <http://doi.acm.org/10.1145/1363686.1364196>.

OSSEC. *HIDS: OSSEC*. 2015. <http://www.ossec.net/>.

PARALLELS. 2015. <http://www.parallels.com/>.

PAXSON, V. Bro: a System for Detecting Network Intruders in Real-Time. *Computer Networks*, v. 31, n. 23-24, p. 2435–2463, 1999. <http://www.icir.org/vern/papers/bro-CN99.pdf>.

PFOH, J. *Leveraging Derivative Virtual Machine Introspection Methods for Security Applications*. Tese (Doutorado) — Technische Universität München, 2013. Doctoral Thesis.

PRESSER, D.; LUNG, L. C.; CORREIA, M. Greft: Arbitrary fault-tolerant distributed graph processing. In: *Big Data (BigData Congress), 2015 IEEE International Congress on*. [S.l.: s.n.], 2015. p. 452–459.

QEMU. *VMM: QEMU*. 2015. <http://www.qemu.org/>.

REITER, M. K. The rampart toolkit for building high-integrity services. In: *Selected Papers from the International Workshop on Theory and Practice in Distributed Systems*. London, UK, UK: Springer-Verlag, 1995. p. 99–110. ISBN 3-540-60042-6. <http://dl.acm.org/citation.cfm?id=647369.723763>.

RIVEST, R. L.; SHAMIR, A.; ADLEMAN, L. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, ACM, New York, NY, USA, v. 21, n. 2, p. 120–126, fev. 1978. ISSN 0001-0782.

ROBIN, J. S.; IRVINE, C. E. Analysis of the intel pentium's ability to support a secure virtual machine monitor. In: *Proceedings of the 9th Conference on USENIX Security Symposium - Volume 9*.

Berkeley, CA, USA: USENIX Association, 2000. (SSYM'00), p. 10–10. <http://dl.acm.org/citation.cfm?id=1251306.1251316>.

SCHLICHTING, R. D.; SCHNEIDER, F. B. *Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems*. 1983.

SCHNEIDER, F. B. Synchronization in distributed programs. *ACM Trans. Program. Lang. Syst.*, ACM, New York, NY, USA, v. 4, n. 2, p. 125–148, abr. 1982. ISSN 0164-0925. <http://doi.acm.org/10.1145/357162.357163>.

SCHNEIDER, F. B. Byzantine generals in action: Implementing fail-stop processors. *ACM Trans. Comput. Syst.*, ACM, New York, NY, USA, v. 2, n. 2, p. 145–154, maio 1984. ISSN 0734-2071. <http://doi.acm.org/10.1145/190.357399>.

SCHNEIDER, F. B. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 22, n. 4, p. 299–319, dez. 1990. ISSN 0360-0300. <http://doi.acm.org/10.1145/98163.98167>.

SCHNEIDER, F. B.; GRIES, D.; SCHLICHTING, R. D. Fault-tolerant broadcasts. *Science of Computer Programming*, v. 4, n. 1, p. 1 – 15, 1984. ISSN 0167-6423. <http://www.sciencedirect.com/science/article/pii/0167642384900091>.

SIEWIOREK, D.; SWARZ, R. *The Theory and Practice of Reliable System Design*. Digital Press, 1982. ISBN 9780932376138. <https://books.google.com.co/books?id=_YtQAAAAMAAJ>.

SMITH, J. E.; NAIR, R. *Virtual machines : versatile platforms for systems and processes*. San Francisco (Calif.): Elsevier, 2005. ISBN 1-55860-910-5. <http://opac.inria.fr/record=b1102162>.

SNORT. *NIDS: Snort*. 2015. <https://www.snort.org/>.

STRONG, H.; DOLEV, D. *Byzantine agreement*. [S.l.]: IEEE Computer Society, 1983.

STUMM, V. et al. Intrusion tolerant services through virtualization: A shared memory approach. In: *Proceedings of the 24th IEEE International Conference on Advanced Information Networking and Applications*. [S.l.: s.n.], 2010. p. 768 –774. ISSN 1550-445X.

TRIPWIRE. *HIDS: Tripwire*. 2015. <http://sourceforge.net/projects/tripwire/>.

VERONESE, G. et al. Efficient byzantine fault-tolerance. *Computers, IEEE Transactions on*, v. 62, n. 1, p. 16–30, Jan 2013. ISSN 0018-9340.

VIRTUALBOX. *VMM: VirtualBox*. 2015. <https://www.virtualbox.org/>.

VMWARE. *VMM: VMware ESX*. 2015. <http://www.vmware.com/>.

VMWARE. *VMM: VMware Workstation*. 2015. <http://www.vmware.com/products/workstation/>.

XEN, V. *Xen Project*. 2015. <http://www.xenproject.org/>.

XU, M. et al. 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*. New York, NY, USA: ACM, 2007. (SACMAT '07), p. 71–80. ISBN 978-1-59593-745-2. <http://doi.acm.org/10.1145/1266840.1266852>.

YIN, J. et al. Separating agreement from execution for byzantine fault tolerant services. *SIGOPS Oper. Syst. Rev.*, ACM, New York, NY, USA, v. 37, n. 5, p. 253–267, out. 2003. ISSN 0163-5980. <http://doi.acm.org/10.1145/1165389.945470>.