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

Valdir Stumm Júnior

SMIT: Uma Arquitetura Tolerante a Intrusões Baseada em Virtualização

Dissertação submetida à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Mestre em Ciência da Computação.

Prof. Lau Cheuk Lung, Dr.
(Orientador)

Florianópolis, Março de 2010

SMIT: Uma Arquitetura Tolerante a Intrusões Baseada em Virtualização

Valdir Stumm Júnior

Esta Dissertação foi julgada adequada para a obtenção do título de Mestre em Ciência da Computação, área de concentração Segurança em Sistemas Computacionais e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

Prof. Mário Antônio Ribeiro Dantas, Dr. (Coordenador)

Banca Examinadora

Prof. Lau Cheuk Lung, Dr.
(Orientador)

Prof. Emerson Ribeiro de Mello, Dr. (IFSC)

Prof. Joni da Silva Fraga, Dr. (UFSC)

Prof. Mário Antônio Ribeiro Dantas, Dr. (UFSC)

Prof. Raimundo da Silva Barreto, Dr. (UFAM)

"I was reading about how countless species are being pushed toward extinction by man's destruction of forests. Sometimes I think the surest sign that intelligent life exists elsewhere in the universe is that none of it has tried to contact us."

Calvin - Bill Watterson

Para minha esposa Cristiana, meus pais, Valdir e Gládis,
e minhas irmãs, Aline e Andrea.

Agradecimentos

Em primeiro lugar, agradeço a Deus, por proporcionar tudo que tem ocorrido em minha vida.

Agradeço à minha família: minha esposa Cristiana por todo o suporte, carinho e amor que sempre deu, principalmente nas horas mais complicadas. Aos meus pais, Valdir e Gládis, pelo apoio que sempre deram em minhas decisões e por tornar tudo isso possível apesar das diversas batalhas que enfrentam diariamente.

Não posso deixar de mencionar meus colegas do LaPeSD e mestrado, os quais espero manter como amigos pelo resto da vida: Tiago Rocha por todos os momentos engraçados que ajudaram a tornar mais agradável o período do mestrado, à Marília pela impagável companhia diária no RU, à Denise pelos agradáveis bate-papos acompanhados de um *espresso* no meio da tarde, ao Jéferson pela companhia que tornou o trajeto noturno de volta para casa menos solitário, e ao Aldelir por sempre contribuir com sua experiência. Além desses, um agradecimento ao amigo e vizinho Elias, por sempre ajudar a lembrar de onde saímos e de que nada sabemos (*HAU!*).

Aos professores membros da banca, Professores Raimundo Barreto, Emerson Mello e Mário Dantas, por todas as valiosas contribuições que deram a este trabalho, pelas críticas de caráter construtivo que me auxiliaram em uma melhor elaboração deste e pelo prazer de conhecê-los pessoalmente. Gostaria também de fazer um agradecimento especial ao Professor Joni da Silva Fraga, que tive a honra e a felicidade de conhecer e poder trabalhar em conjunto, por sua paciência, atenção e conhecimento inestimáveis. Agradeço também ao Professor Miguel Correia, que mesmo não conhecendo pessoalmente sempre foi muito atencioso e acessível, sempre acrescentando qualidade e chamando a atenção para pontos importantes do trabalho. Agradeço ao meu orientador, Professor Lau Cheuk Lung, por ter sugerido o tema deste trabalho, por ter confiado em mim para desenvolvê-lo, pela paciência com minha eventual teimosia e pela transparência com a qual sempre conduziu a relação orientador-orientando.

Para finalizar, agradeço a CAPES por tornar possível minha dedicação integral ao mestrado através da bolsa que me foi concedida, e à Verinha, por sua simpatia e paciência intermináveis e por sua constante disposição em nos ajudar.

Sumário

Sumário xi

Lista de Figuras xiv

Lista de Tabelas xv

Lista de Algoritmos xvi

Lista da Acrônimos xviii

Resumo xix

Abstract xx

1 Introdução 1

1.1 Contextualização 1
1.2 Definição do Problema 2
1.3 Abordagem Proposta 3
1.4 Objetivos 3
1.5 Estrutura do Trabalho 4

2 Conceitos Básicos 5

2.1 Confiança no Funcionamento 6
2.1.1 Ameaças 6
2.1.2 Atributos 11
2.1.3 Meios 12
2.1.4 Tolerância a Faltas em Sistemas Distribuídos . . 14
2.1.5 Tolerância a Intrusões 19
2.1.6 Consenso em Sistemas Distribuídos 21
2.1.7 Acordo Bizantino 22
2.1.8 Considerações sobre Confiança no Funcionamento 22
2.2 Modelos de Sistemas Distribuídos 23
2.2.1 Modelo de Interação 23
2.2.2 Modelo de Falhas 25
2.2.3 Modelo de Segurança 25
2.2.4 Considerações sobre Modelos de Sistema 26
2.3 Tecnologia de Virtualização 27
2.3.1 Propriedades dos VMMs 28

2.3.2 Tipos de VMMs 29
2.3.3 Considerações sobre Virtualização 30
2.4 Considerações sobre o Capítulo 31

3 Trabalhos Relacionados 33
3.1 Técnicas de RME Tolerante a Faltas Bizantinas 34
3.1.1 *Practical Byzantine Fault Tolerance* 35
3.1.2 Zyzzyva - Speculative Byzantine Fault Tolerance 40
3.2 Modelos Híbridos 44
3.2.1 *Trusted Timely Computing Base* 45
3.2.2 VM-FIT 47
3.2.3 LBFT 50
3.3 Conclusões do Capítulo 53

4 SMIT - Uma abordagem para TI usando Memória Compartilhada 57
4.1 Motivação e Descrição Geral 58
4.2 Modelo de Sistema 59
4.2.1 Modelo de Interação 60
4.2.2 Modelo de Falhas 62
4.2.3 Modelo de Segurança 63
4.3 Componente *Postbox* 64
4.3.1 Funcionamento da *Postbox* 64
4.3.2 Detalhes de Construção da *Postbox* 66
4.4 Propriedades de SMIT 67
4.5 Protocolo 67
4.5.1 Tipos de Mensagens 69
4.5.2 O Lado Cliente 70
4.5.3 O Lado das Réplicas 71
4.6 Correção de SMIT 73
4.7 *Distributed* SMIT 76
4.7.1 Abordagem Primário-*Backup* 77
4.8 Comparações com Trabalhos Relacionados 79
4.9 Considerações sobre o Capítulo 80

5 Protótipo e Resultados 83
5.1 Implementação da *Postbox* 83
5.1.1 *n-write/1-read* 84
5.1.2 *n-write/n-read* 85
5.1.3 Protocolo de Coleta de Lixo 86
5.1.4 Implementação e Resultados 88
5.2 Camada de Replicação 91

5.3 Cliente 91
5.4 Avaliação de Desempenho 93
5.4.1 Tempo de Resposta 94
5.4.2 Mecanismos Criptográficos 97
5.4.3 *Throughput* 99
5.4.4 Tempo de Recuperação - DSMIT 99
5.5 Considerações Finais 100

6 Conclusões e Perspectivas Futuras **103**

Referências Bibliográficas **107**

Lista de Figuras

2.1 Classes de Faltas [Verissimo e Rodrigues 2001] 10
2.2 Tipos de VMMs 30

3.1 Diagrama de funcionamento do PBFT 37
3.2 Execução do protocolo de acordo do Zyzzyva 42
3.3 Arquitetura da TTCB 45
3.4 Arquitetura RESH 48
3.5 Arquitetura REMH 49
3.6 Diagramas de execução dos protocolos LBFT_1 e LBFT_2 52

4.1 Visão Geral da Arquitetura SMIT 59
4.2 Exemplo de utilização da *Postbox* 66
4.3 Diagrama de execução do SMIT 69
4.4 Arquitetura SMIT Distribuída (DSMIT) 78

5.1 Abordagem *Multi-write/Single-read* 86
5.2 Abordagem *Multi-write/Multi-read* 87
5.3 Comparação de resultados entre abordagens *nw1r* e *nwnr* 91
5.4 Tempo de Resposta por Operação 96
5.5 Tempo de Resposta por Operação - Diversidade de SO . 98
5.6 Tempo de resposta por operação variando mecanismo criptográfico 99
5.7 *Throughput* verificado através da variação na carga . . . 100

Lista de Tabelas

3.1 Comparação entre os trabalhos relacionados 55

4.1 Comparação entre SMIT e os trabalhos relacionados . . 80

Lista de Algoritmos

4.1 Envio de requisição do cliente para o grupo de réplicas S 71
4.2 Algoritmo de Recebimento de Requisições 72
4.3 Leitura da *Postbox* e execução das requisições 73
5.1 *Daemon* de Controle da *Postbox* - *nw1r* 85
5.2 *Daemon* de Controle da *Postbox* - *nwnr* 86

Lista de Códigos

5.1 Código Python para escrita na *Postbox* 90
5.2 Código Python da implementação da Camada de Replicação 92
5.3 Interface de acesso à *Postbox* - lado servidor 92
5.4 Código Python da implementação do Cliente 93

Lista de Acrônimos

AVI *Attack-Vulnerabilty-Intrusion*

BFT *Byzantine Fault Tolerance*

DSMIT *Distributed SMIT*

FLP *Fischer, Lynch and Paterson*

KVM *Kernel Virtual Machine*

LBFT *Lightweight Byzantine Fault Tolerance*

MAC *Message Authentication Code*

PBFT *Practical Byzantine Fault Tolerance*

REMH *Redundant Execution on Multiple Hosts*

RESH *Redundant Execution on a Single Host*

RME Replicação Máquina de Estados

SMIT *Shared Memory Intrusion Tolerance*

SO Sistema Operacional

TCB *Trusted Computing Base*

TI Tolerância a Intrusões

VM *Virtual Machine*

VMBFT *Virtual Machine based Byzantine Fault Tolerance*

VMM *Virtual Machine Monitor*

VM-FIT *Virtual Machine Fault and Intrusion Tolerance*

Resumo

Diversas pesquisas para desenvolvimento de soluções práticas de suporte a aplicações distribuídas tolerantes a faltas bizantinas têm sido realizadas nos últimos anos. Embora seja um conceito apresentado há cerca de trinta anos, somente na última década foram apresentadas soluções para tolerância a faltas bizantinas com foco na viabilidade prática de implementação. Dado o aumento da conectividade em todos tipos de sistemas computacionais, estudos recentes apresentam propostas que visam garantir que o sistema se mantenha correto mesmo na ocorrência de uma intrusão, isto é, de um ataque que obteve sucesso ao explorar uma vulnerabilidade. O principal desafio dos trabalhos na área de tolerância a faltas bizantinas/intrusões reside em propor arquiteturas e protocolos cuja implementação possa ser concretizada em termos práticos. Os fatores proibitivos de propostas anteriores dizem respeito ao custo de implantação e execução dessas propostas e também à complexidade do código necessário para a implementação dos protocolos. Alguns trabalhos têm considerado o uso de tecnologias de virtualização para a construção de ambientes de computação confiável, tomando proveito de recursos específicos de ambientes virtualizados em prol de protocolos de suporte mais simples.

Nesse contexto, o presente trabalho apresenta SMIT, uma arquitetura para tolerância a intrusões baseada em tecnologias de virtualização e na utilização de memória compartilhada entre máquinas virtuais. Tal área de memória é utilizada para a simplificação dos protocolos de execução de requisições de clientes nas réplicas de serviço, concretizando o consenso sobre a ordem de execução de operações em tais réplicas em um menor número de passos de comunicação do que propostas anteriores. Devido à utilização da memória compartilhada, também foi possível reduzir o número mínimo de réplicas requeridas para garantia da correção do modelo sob a presença de f réplicas faltosas. Tal redução se deu de $3f + 1$ para $2f + 1$. Para demonstrar a viabilidade prática da proposta, foram realizados experimentos práticos sobre um protótipo implementado da arquitetura.

Palavras chave: Tolerância a Intrusões, Virtualização, Memória Compartilhada.

Abstract

Much research aiming the development of practical solutions to support Byzantine Fault Tolerance (BFT) distributed applications has been made in recent years. Although it is a concept presented thirty years ago, only in the last decade studies presenting BFT solutions aiming the practical feasibility were proposed. Given the increasing connectivity in all kind of computing systems, recent studies are presenting solutions to ensure that the systems remain available even in the event of an intrusion, ie, an attack that succeed exploiting a vulnerability. The main challenge of the studies in the Byzantine Fault/Intrusion Tolerance field is to propose architectures and protocols whose implementation can be achieved in a practical way. Prohibitive factors of previous studies were related to the deployment and the implementation costs of these proposals and also to the code complexity needed to implement the protocols. Some studies have considered the use of virtualization technologies for building trusted computing environments, taking advantage of specific features of virtualized environments to simplify the protocols.

In this context, this paper presents SMIT, an architecture for intrusion tolerance based on virtualization technologies and on a shared memory area between virtual machines. This memory is used for the simplification of protocols running on the service replicas to process client requests, achieving the consensus over the execution order of operations in such replicas in a number of communication steps smaller than previous researches. Due to the use of shared memory, it is also possible to reduce the minimum number of replicas required to ensure the correctness of the model under the presence of f faulty replicas. This reduction took place from $3f + 1$ to $2f + 1$. To demonstrate the practical feasibility of the proposal, practical experiments on a prototype of the architecture have been made.

Keywords: Intrusion Tolerance, Virtualization, Shared Memory.

Capítulo 1

Introdução

1.1 Contextualização

O papel que infra-estruturas e sistemas computacionais desempenham em nossa sociedade vem crescendo em importância com o passar das últimas décadas. A dependência e a confiança depositadas sobre esses sistemas vêm aumentando consideravelmente, dia após dia. Dada a relevância desses sistemas para o funcionamento de serviços básicos para o nosso cotidiano, é necessário que estes se comportem corretamente mesmo sob a presença de faltas, as quais podem acarretar em grandes prejuízos, desde financeiros até humanos. Recentemente, as faltas em sistemas computacionais têm aparecido mais freqüentemente sob a forma de intrusões, que são o resultado de um ataque que obtém sucesso ao explorar uma ou mais vulnerabilidades [Correia 2005]. Para garantir que esses sistemas permaneçam disponíveis, corretos e seguros mesmo sob a presença de faltas e vulnerabilidades, é necessário que sejam desenvolvidos mecanismos para viabilizar a tolerância a intrusões nesses ambientes.

Apesar da ocorrência de falhas em sistemas computacionais se apresentarem mais freqüentemente ligadas à faltas de *crash* do que a faltas bizantinas (maliciosas ou intrusões), os prejuízos são normalmente muito mais severos quando da ocorrência destas últimas. Para exemplificar, podemos citar o caso de um intruso malicioso realizando operações financeiras (não-autorizadas) de grande porte em servidores de um banco, ou ainda, um intruso concretizando atos terroristas em um sistema computacional de uma usina nuclear. A literatura tem mostrado que, para tratar do problema dos ataques maliciosos sobre o software, as soluções devem adotar técnicas de Replicação Máquina de Estados [Schneider 1990] aliadas às técnicas de diversidade de projeto [Hiltunen, Schlichting e Ugarte 2003]. Essas propostas são fundamentadas na observação de que faltas bizantinas com intenção maliciosa (intrusão) ocorrem pela tentativa do uso das vulnerabilidades da porção do sistema computacional que é implementada em software (ou seja, sistema operacional, *drivers*, serviços, aplicações, etc.).

Visando garantir segurança para esses sistemas, o desenvolvimento de soluções para suporte a aplicações distribuídas tolerantes a faltas tem sido alvo de pesquisas há mais de vinte e cinco anos [Lamport, Shostak e Pease 1982, Schneider 1990]. No entanto, somente nos últimos dez anos é que surgiram propostas com viabilidade prática para Replicação Máquina

de Estados (RME) [Schneider 1990] tolerante a faltas bizantinas (BFT) [Castro e Liskov 1999, Yin et al. 2003, Kotla et al. 2007].

1.2 Definição do Problema

Os protocolos para BFT, em geral, consistem na replicação de um determinado serviço em um conjunto de máquinas que se comunicam, objetivando prover um serviço seguro e confiável, mesmo sob a presença de um número limitado f de membros maliciosos (intrusos) que se comportem de forma diferente da especificação correta do protocolo. Esses protocolos visam possibilitar a implementação de serviços replicados capazes de atender aos requisitos de confiabilidade, integridade e disponibilidade, que são fundamentais para se alcançar a Confiança no Funcionamento (*Dependability* [Avizienis et al. 2004]).

Vários trabalhos com propostas para BFT têm apresentado soluções elegantes para, por exemplo, reduzir a complexidade de mensagens, reduzir o número de passos de comunicação para alcançar o acordo [Kotla et al. 2007], e para otimizar o uso dos recursos computacionais disponíveis [Yin et al. 2003, Correia, Neves e Verissimo 2004, Luiz et al. 2007]. As soluções também procuram circunscrever algumas impossibilidades teóricas (p. ex. condição FLP [Fischer, Lynch e Paterson 1985] e a necessidade de, no mínimo, $3f+1$ réplicas para tolerar faltas bizantinas [Lamport, Shostak e Pease 1982]). Para isso, tais soluções vão desde assumir premissas de sincronia mais relaxadas (porém, práticas e realistas) até a definição de arquiteturas híbridas com o uso de componentes seguros (os *wormholes* [Correia, Neves e Verissimo 2004, Correia et al. 2007]).

Recentemente, foram propostos alguns trabalhos que fazem uso de tecnologias de virtualização para implementação de sistemas tolerantes a intrusões em uma única máquina física [Bessani et al. 2007, Reiser e Kapitza 2007]. Nessas propostas, cada réplica é executada em um ambiente virtualizado, sendo que todo o conjunto de réplicas reside no mesmo computador. Essas propostas permitem implementar o conceito de diversidade de serviços e de sistemas operacionais em nível de máquinas virtuais [Avizienis e Kelly 1984]. Ou seja, usar diversidade de projeto ao implementar cada réplica de um protocolo BFT em uma máquina virtual (VM - *virtual machine*) distinta (com sistema operacional distinto das outras VMs), com todas as VMs executando dentro de uma única máquina física. Uma vantagem evidente dessa abordagem é a redução do custo de replicação para implementação de um serviço tolerante a intrusões pelo uso de uma única máquina física. Como contraponto, a máquina real é um ponto singular de falha: se a mesma sofre um *crash*, o serviço se torna indisponível. No entanto, para contornar tal situação, pode-se usar técnicas de replicação

tradicionais, tolerantes a faltas de *crash*, para o sistema físico.

Apesar de práticas, até mesmo as abordagens mais recentes ainda apresentam um elevado custo relacionado à sua implementação, seja devido ao custo necessário para implementar a replicação física dos equipamentos que compõem o sistema (por exemplo, *3f+1* máquinas físicas), ou seja com relação à complexidade dos protocolos que devem ser implementados, o que gera altos custos para o desenvolvimento destes. A complexidade de implementação de tais protocolos se deve, em geral, à grande quantidade de mensagens que devem ser trocadas e operações que devem ser realizadas para efetivar a execução de uma requisição. Além do custo de implementação, complexidade de código torna o software mais suscetível à ocorrência de falhas [Nagappan, Ball e Zeller 2006]. Assim sendo, é desejável que protocolos para tolerância a faltas sejam especificados visando diminuir a complexidade do código necessário para seu desenvolvimento, uma vez que um *bug* presente no protocolo de replicação pode comprometer a correção de toda a arquitetura tolerante a faltas.

1.3 Abordagem Proposta

O presente trabalho apresenta uma solução simples, prática e eficiente para tolerância a intrusões em sistemas de computação distribuída utilizando uma abordagem baseada em virtualização e memória compartilhada. Tal como fora anteriormente proposto em outros estudos [Reiser e Kapitza 2007, Chun, Maniatis e Shenker 2008], este trabalho pretende implementar a replicação dos sistemas em máquinas virtuais, ao invés de utilizar replicação física de equipamentos, de modo a diminuir os custos envolvidos na replicação de sistemas. Diferentemente dos trabalhos acima citados, a abordagem seguida no presente trabalho é baseada na utilização de uma abstração de memória compartilhada entre as máquinas virtuais, a qual apresenta como grande vantagem uma simplificação do protocolo de execução de requisições e de consenso entre as réplicas. Tal simplificação é obtida através das propriedades da abstração de memória compartilhada, que possibilitam a diminuição do número de mensagens trocadas entre réplicas em busca de consenso sobre a ordem de execução das operações.

1.4 Objetivos

O principal objetivo do presente trabalho é investigar um modelo baseado em replicação, usando uma arquitetura de máquinas virtuais, que permita a implementação de um sistema tolerante a intrusões, utilizando memória compartilhada entre as máquinas virtuais para estabelecimento de consenso entre as réplicas. Uma vez definido o modelo, o presente tra-

balho pretende apresentar a especificação de um protocolo para execução de serviços replicados sobre tal modelo, com foco na simplicidade de implementação e buscando reduzir o número mínimo de réplicas necessárias para $2f + 1$.

De acordo com o Objetivo Geral, alguns objetivos específicos se fazem necessários:

1. Levantamento e estudo de referências teóricas, de modo a formar uma base sólida de conceitos relacionados ao estudo de sistemas tolerantes a intrusões.
2. Especificação de um protocolo de consenso baseado na arquitetura descrita acima, bem como a elaboração de suas provas de correção.
3. Implementação de um protótipo baseado na arquitetura e no protocolo propostos.
4. Testes de execução e avaliação do desempenho obtido com o protótipo.
5. Especificação e implementação de protocolos que, juntamente ao protocolo de consenso, garantam a execução segura de serviços no modelo proposto.

1.5 Estrutura do Trabalho

O restante deste documento está organizado da seguinte forma:

- **Capítulo 2 - Conceitos Básicos**: apresenta os conceitos básicos que formam a base teórica necessária para o desenvolvimento deste trabalho.
- **Capítulo 3 - Trabalhos Relacionados**: apresenta os principais trabalhos relacionados ao estudo aqui apresentado.
- **Capítulo 4 - SMIT - Uma abordagem para TI usando Memória Compartilhada**: descreve os principais detalhes relativos ao desenvolvimento de SMIT, a arquitetura aqui proposta.
- **Capítulo 5 - Protótipo e Resultados**: apresenta o protótipo implementado, bem como a realização de experimentos e análise dos resultados obtidos.
- **Capítulo 6 - Conclusões e Perspectivas Futuras**: apresenta as considerações finais sobre o trabalho aqui apresentado, indicando caminhos futuros para continuidade do que foi aqui apresentado.

Capítulo 2

Conceitos Básicos

No contexto de sistemas computacionais, a busca pela garantia da execução correta de serviços é uma das prioridades no desenvolvimento destes. Garantir que um sistema permaneça operando corretamente, aconteça o que acontecer, é alvo de inúmeras pesquisas na área da Ciência da Computação. Dentro desse contexto, o conceito de Confiança no Funcionamento foi inserido de modo a definir de forma mais precisa as propriedades e requisitos que sistemas computacionais devem cumprir para que apresentem uma execução correta, independentemente da ocorrência de eventos indesejados. Dentre as áreas que mais se utilizam desse conceito, está a área de Sistemas Distribuídos, visto que várias soluções que propõem a execução correta de serviços são implementadas de forma distribuída para garantir propriedades como disponibilidade, isolamento, etc.

Com o advento da tecnologia de virtualização de sistemas, muitos sistemas distribuídos passaram a tomar proveito de técnicas de virtualização, seja para diminuir os custos de infra-estrutura ou para aumentar a confiabilidade desses sistemas. Assim, através do uso de máquinas virtuais, muitos sistemas que eram distribuídos ao longo de várias máquinas físicas puderam ser migrados para plataformas baseadas em uma única máquina, sem perder as propriedades de isolamento entre os serviços das máquinas virtuais e eficiência na execução.

O objetivo deste capítulo é apresentar e detalhar os conceitos básicos para a compreensão deste trabalho, porque servirão como base teórica para os capítulos posteriores. Inicialmente, será apresentado na Seção 2.1 o conceito de Confiança no Funcionamento (*Dependability*), bem como seus detalhes e propriedades, visando formar uma base teórica para o entendimento dos capítulos seguintes, com relação às técnicas utilizadas para garantir a correta execução de sistemas. Em seguida, a Seção 2.2 apresenta os principais aspectos que devem ser levados em consideração ao construir um sistema distribuído. Na seqüência, a Seção 2.3 apresenta os principais conceitos relativos à Virtualização de Sistemas e utilização de Máquinas Virtuais, visto que o foco desse trabalho é prover tolerância a faltas através da utilização de tecnologias de virtualização. Finalizando o capítulo, a Seção 2.4 faz um apanhado geral sobre os conceitos apresentados nas Seções anteriores a ela e apresenta a relação desses conceitos com o trabalho aqui apresentado.

2.1 Confiança no Funcionamento

Sistemas computacionais são empregados em diversas áreas do conhecimento, visando facilitar e agilizar a realização de tarefas básicas para o pleno funcionamento dessas áreas. Freqüentemente, tais sistemas são também utilizados para dar suporte à atividades críticas, básicas para o nosso cotidiano. Alguns exemplos são sistemas de controle de geração de energia nuclear, de distribuição de energia, de controle de atividades médico-hospitalares, de controle financeiro, dentre outras. Caso tais sistemas venham a falhar, os prejuízos acarretados por essas falhas podem ser enormes, podendo causar enormes perdas financeiras e, até mesmo, perdas humanas, ou prejuízos ao meio-ambiente. Dessa forma, é necessário que esses sistemas críticos sejam confiáveis. Nesse contexto, a Confiança no Funcionamento (*Dependability*) de um sistema é uma propriedade que diz que se pode, justificadamente, depositar confiança sobre o serviço oferecido por este sistema [Laprie, Avizienis e Kopetz 1992].

Esta seção apresenta os fundamentos básicos em Confiança no Funcionamento de sistemas computacionais, seguindo o modelo apresentado em [Avizienis et al. 2001], o qual descreve o conceito através das ameaças, de seus atributos e dos meios utilizados para que se obtenha Confiança no Funcionamento.

2.1.1 Ameaças

Sistemas computacionais que seguem estritamente a sua especificação são considerados sistemas corretos. A não-correspondência à especificação do sistema faz com que este apresente comportamento incorreto, visto que poderá proceder de forma diferente do comportamento esperado para o sistema. Tal falta de correspondência pode ser gerada por um grande número de possíveis ameaças, tais como projetos mal-implementados, componentes comprometidos, entidades maliciosas sob controle do sistema, dentre outras. A taxonomia de ameaças à Confiança no Funcionamento de um sistema é dividida em três categorias básicas [1] [Avizienis et al. 2001]: a **falta**, que é a causa do erro, por exemplo: defeito de projeto em um *chip* aritmético. O **erro**, que é o estado do sistema que pode levar a uma falha na sua execução, por exemplo: execução de uma operação no *chip* defeituoso. A **falha**: é a manifestação do erro, causando desvio da especificação do serviço, por exemplo: resultado incorreto retornado ao software que invocou a operação no *chip*.

[1] No Brasil, existe outra taxonomia, que descreve as ameaças através de: falha, erro e defeito [Weber 2003].

2.1.1.1 Falta

Dada a complexidade de sistemas computacionais em geral, estes podem apresentar faltas com origens bastante diversas. As faltas podem ocorrer devido a uma grande variedade de eventos, como um desastre natural que causa o desligamento do sistema, um ser humano que desliga os equipamentos acidentalmente (ou até mesmo intencionalmente), um projeto mal-elaborado, uma interferência eletromagnética que pode causar perda de dados em transferência sem fio, um erro de programação, etc. Dada essa diversidade de fontes de faltas, de acordo com [Laprie, Avizienis e Kopetz 1992], as faltas podem ser classificadas com relação a cinco pontos de vista principais:

- **Causas Fenomenológicas**: físicas ou interferência humana.
- **Natureza**: acidental ou intencional.
- **Fase de Criação ou Ocorrência**: fase de desenvolvimento ou fase de operação.
- **Situação com Relação aos Limites do Sistema**: internas ou externas.
- **Persistência**: permanentes ou temporárias.

Os cinco pontos de vista apresentados acima possibilitam o detalhamento de uma falta com relação a sua origem. A seguir, um exemplo ilustrando um evento e suas relações com as classes de faltas apresentadas acima: um terremoto causa a destruição de um *datacenter*, fazendo com que todos os equipamentos sejam desligados e destruídos. A causa fenomenológica é física (o terremoto), a natureza do incidente é acidental, a fase de ocorrência é durante a fase de operação do sistema. Com relação aos limites do sistema, a falta é externa, pois é causada por fatores externos, e, se considerarmos a ocorrência do terremoto como a falta, a sua persistência é temporária, embora os danos sejam permanentes.

Já em [Avizienis et al. 2004], além das classes de faltas apresentadas acima, foram introduzidas novas classes, complementando as que foram anteriormente apresentadas. Além disso, foram feitas as combinações das classes de faltas, visando identificar a taxonomia completa de possibilidades de ocorrência de faltas. Através da combinação das classes apresentadas, foi possível o estabelecimento de 31 combinações possíveis, não sendo descartadas futuras combinações. As classes que complementam as apresentadas acima são:

- **Dimensão**: faltas de hardware ou de software.
- **Objetivo**: faltas maliciosas ou não-maliciosas.
- **Intenção**: deliberada ou não-deliberada.
- **Capacidade**: acidentais ou causadas por incompetência.

Outra forma de caracterizar as faltas é com relação às interações entre os componentes de um sistema, ou seja, as ações que são realizadas e que são visíveis externamente ao componente. Tal abordagem é mais interessante para uso em sistemas distribuídos, devido à natureza de tais sistemas, que são formados por componentes que interagem através de comunicação. Esse modelo de faltas permite uma especificação de um sistema distribuído de forma mais precisa, com relação às faltas que tal sistema poderá tolerar, tornando mais fácil a sua compreensão. Em [Verissimo e Rodrigues 2001], são apresentados dois grupos de classes de faltas: *omissivo*, faltas que ocorrem pela não-execução de determinada interação por parte de um componente, e *assertivo*, faltas que ocorrem pelo componente realizando determinada ação fora da sua especificação. No âmbito do grupo omissivo, o seguinte grupo de faltas é composto [Verissimo e Rodrigues 2001]:

- ***Crash***: um componente que pára sua execução permanentemente.
- **Omissão**: um componente que omite uma ação, entre intervalos de tempo não-uniformes.
- **Temporais**: um componente que atrasa ou antecipa, de forma não prevista pela especificação, a realização de uma ação.

Já o grupo de faltas assertivo pode ser considerado mais perigoso que o anterior. Ao invés de afetar apenas o fato de uma ação ser realizada ou não, as faltas desse grupo ocorrem por fuga da especificação, ou seja, um componente que realiza sua ação, porém de forma diferente do que foi determinado em sua especificação correta. O grupo de faltas assertivo é dividido da seguinte forma [Verissimo e Rodrigues 2001]:

- **Sintáticas**: ocorrem quando o resultado da ação de um componente fugir à sua especificação.
- **Semânticas**: ocorrem quando o significado do resultado de uma ação, mesmo que esta seja sintaticamente correta, fugir do funcionamento esperado para a aplicação.

Para explicar melhor as classes acima, considere um *display* de velocímetro digital de um carro popular. Este equipamento é especificado para mostrar valores compostos por até 3 dígitos, como “80”, “100”, etc, em km/h. Caso esse *display* apresente o valor “ola”, este estará fugindo do formato de sua especificação, apresentando assim uma falta sintática. Já no caso de o *display* apresentar o valor “500”, este estará seguindo a sua especificação, pois apresenta 3 dígitos. Porém, tal valor é semanticamente incorreto para um carro popular, visto que é sabido que sua velocidade pode chegar a, no máximo, 220 km/h.

Ainda, para complementar os dois grupos de faltas apresentados (omissivas e assertivas), existem as faltas *arbitrárias*, que são as faltas sobre as quais não se pode fazer hipótese alguma. Essas faltas compreendem ambos os domínios de faltas omissivas e assertivas e ocorrem por possíveis inconsistências e imprevisibilidade na percepção das faltas por parte dos componentes. Por exemplo, em um sistema de difusão de mensagens pela rede, onde existe um grupo de 3 componentes para o qual um componente mestre decide enviar mensagens com a ação a ser tomada por todos. Caso o mestre envie uma mensagem com um valor “a” para o componente 1, se omita de enviar a mensagem ao componente 2, que permanece aguardando, e envie um valor “b” ao componente 3, teremos os 3 componentes apresentando inconsistência entre as ordens recebidas. Assim, pode-se dizer que o comportamento do componente mestre é arbitrário, pois além de fugir da especificação, seu comportamento pode ser visto de forma inconsistente pelos membros envolvidos na interação. As faltas arbitrárias foram apresentadas como *faltas bizantinas*, no artigo que descreve o Problema dos Generais Bizantinos [Lamport, Shostak e Pease 1982], que é um problema semelhante ao exemplo recém descrito. A relação das classes detalhadas acima é apresentada na Figura 2.1, onde podemos ver que a classe mais abrangente é a classe de faltas arbitrárias, que compreende todos os outros domínios de faltas.

2.1.1.2 Erro

Erros são estados do sistema que podem levar ou não à ocorrência de uma falha. Como no exemplo referido anteriormente na Seção 2.1.1, a execução de uma operação em um *chip* aritmético faltoso leva o sistema a atualizar seu estado com valores incorretos, causando assim o **erro**, que pode vir a se manifestar em uma falha, caso este seja utilizado, por exemplo, para apresentar ao usuário o saldo total de sua conta bancária. A possibilidade de um erro gerar uma falha depende de três fatores [Laprie, Avizienis e Kopetz 1992, Avizienis et al. 2004]:

- **A composição do sistema e a utilização de redundância**: a redun-

Figura 2.1: Classes de Faltas [Verissimo e Rodrigues 2001]

dância pode evitar que um erro se transforme em uma falha.

- **As atividades do sistema**: pois um erro pode ser sobrescrito antes de gerar algum estrago ao sistema.
- **A definição de falha**: determinada ocorrência pode ser considerada uma falha para um usuário, enquanto que para outro, não.

2.1.1.3 Falha

Uma falha, conforme já descrito anteriormente, é a manifestação de um erro para o usuário, onde usuário significa a entidade que depende da execução do sistema para realização de uma tarefa. Como descrito na Seção 2.1.1.2, determinado comportamento do sistema ser considerado uma falha depende do efeito que ele gera para seus usuários. Uma falha pode ocorrer de diversas formas, chamadas de **modos de falha**, que são descritos através de três pontos de vista [Laprie, Avizienis e Kopetz 1992, Avizienis et al. 2004]:

- **Domínio de falhas**: valor (o valor retornado pelo serviço não segue as especificações deste) ou tempo (o tempo de entrega do serviço não segue suas especificações).
- **Percepção por seus usuários**: consistentes (todos os usuários percebem a mesma falha) ou inconsistentes (uma falha é percebida de formas diferentes pelos usuários, podendo até não ser percebida, sendo também chamada de falha bizantina [Lamport, Shostak e Pease 1982]).

- **Conseqüências para o ambiente**: benignas (conseqüências da falha são da mesma ordem de magnitude que o serviço correto) ou catastróficas (conseqüências da falha são muito maiores do que o benefício oferecido pelo serviço correto).

2.1.1.4 Relações entre Falta, Erro e Falha

Sendo eventos diretamente dependentes uns dos outros, o processo de ocorrência e manifestação de faltas, erros e falhas é definido da seguinte forma [Laprie, Avizienis e Kopetz 1992]: uma falta é **dormente** (inativa) enquanto não causa um erro. A partir do momento em que uma falta gera um erro, esta passa a ser **ativa**. Por exemplo, uma falta de projeto em um software só passa a ser ativa no momento em que a seção de código que apresenta tal falta for executada, antes disso, essa falta de projeto é uma falta dormente. Um erro pode apresentar dois estados: **latente** ou **detectado**. Um erro está no estado latente enquanto não for detectado. Uma falha ocorre quando um erro atinge o serviço oferecido pelo sistema e pode ser propagada para outros componentes.

Devido à natureza de sistemas computacionais, os quais são formados por vários componentes aninhados e interligados, a ocorrência de uma falha em um componente leva a uma falta em um outro componente com o qual o primeiro possui interface. Por exemplo, um componente A é interligado a um componente B, o qual necessita ser alimentado por A. Uma vez que o componente A falha, os valores recebidos por B podem passar a ser incorretos. Assim, a falha ocorrida no componente A representa uma falta para o componente B, que por sua vez poderá se propagar em um erro e, conseqüentemente, em uma falha e em uma falta em um outro componente que dependa de entradas de B. Assim, podemos estabelecer a seguinte relação, onde os grupos fechados por parênteses significam componentes aninhados:

((falta → erro → falha) → falta → erro → falha) → falta → **...**

2.1.2 Atributos

O conceito de Confiança no Funcionamento é um conceito composto, que requer, para que seja alcançado, que o sistema apresente os seguintes atributos básicos [Laprie, Avizienis e Kopetz 1992, Avizienis et al. 2001, Avizienis et al. 2004] (os termos em língua inglesa foram expressos entre parênteses para evitar ambigüidade na tradução):

- **Disponibilidade (*availability*)**: prontidão para oferecer um serviço correto.

- **Confiabilidade (*reliability*)**: continuidade no oferecimento do serviço correto.

- **Segurança (*safety*)**: a não-ocorrência de conseqüências catastróficas para o ambiente.

- **Integridade (*integrity*)**: a não-ocorrência de alterações indevidas no conteúdo da informação.

- **Manutenabilidade (*maintainability*)**: a aptidão para submeter-se a reparos e melhorias.

Os itens acima listados são os atributos básicos para a obtenção de Confiança no Funcionamento (*Dependability*) em um sistema computacional. Ou seja, é necessário que o sistema permaneça sempre disponível, oferecendo seu serviço correto, que este ofereça continuidade no serviço oferecido, que este não sofra problemas que gerem conseqüências ruins para seus usuários e ambiente, que a informação envolvida no sistema não seja alterada sem permissão e que o sistema possa ser corrigido e melhorado. É importante perceber que os atributos acima listados são dependentes da aplicação que o requer, sendo que, dependendo do caso, alguns atributos podem ser relaxados ou até mesmo não-necessários [Avizienis et al. 2001].

Somando o atributo de **Confidencialidade (*confidentiality*)**, que consiste na não-ocorrência de divulgação da informação a entidades não-autorizadas, aos atributos de disponibilidade e integridade, ambos com relação a usuários autorizados, obtemos o conceito de **Segurança (*security*)**.

2.1.3 Meios

A construção de aplicações e sistemas que provêm Confiança no Funcionamento requer a utilização de técnicas de projeto e implementação, de modo que tais sistemas sejam capazes de evitar ou lidar com a presença de faltas. Diversas técnicas foram propostas e implementadas para que os sistemas fossem capazes de prover Confiança no Funcionamento para seus usuários, e tais técnicas foram divididas em quatro categorias principais [Laprie, Avizienis e Kopetz 1992, Avizienis et al. 2001, Avizienis et al. 2004]:

- **Prevenção de Faltas**: prevenir a ocorrência ou introdução de faltas.

- **Tolerância a Faltas**: evitar falhas no serviço quando da ocorrência de faltas.

- **Remoção de Faltas**: reduzir o número e a severidade de faltas.
- **Previsão de Faltas**: estimar o número de faltas, futura incidência destas e suas possíveis conseqüências.

Os próximos parágrafos fazem uma breve descrição de cada um dos meios utilizados para obtenção de Confiança no Funcionamento. Dada a importância da Tolerância a Faltas para este trabalho, esse conceito será apresentado separadamente na Seção 2.1.4.

Prevenção de faltas A prevenção de faltas é uma categoria que é concretizada, principalmente, através de técnicas de controle de qualidade no projeto e desenvolvimento de sistemas [Avizienis et al. 2001]. Para sistemas implementados na camada de software, a prevenção de faltas é realizada através de técnicas de engenharia de software, como métodos de programação segura, revisão de código, testes de software, dentre outras. Além disso, outras técnicas são aplicadas durante a fase de execução do sistema, como utilização de *firewall*, sistemas de proteção contra vírus, *trojans*, *malwares*, dentre outros. Uma técnica bastante utilizada em sistemas distribuídos é a recuperação pró-ativa [Castro e Liskov 2000, Reiser e Kapitza 2007], que consiste na "*limpeza*" preventiva de sistemas periodicamente, sendo obtida através do *reboot* dos sistemas, com aplicação de atualizações e até mesmo mudança da base de código, para evitar que as falhas de segurança sejam descobertas. Para sistemas implementados em hardware, a modularização e a definição de regras de projeto são meios utilizadas para prevenção de faltas na fase de desenvolvimento. Para evitar faltas de natureza física durante a fase de operação do sistema, são utilizadas técnicas de blindagem, para evitar, por exemplo, que interferências eletromagnéticas possam afetar o sistema.

Remoção de faltas A remoção de faltas consiste na detecção, localização e remoção das faltas encontradas. As técnicas de remoção podem ser aplicadas durante a fase de desenvolvimento e também durante a fase de operação do sistema [Avizienis et al. 2004]. Durante a fase de projeto e desenvolvimento, são realizadas verificações sobre o sistema, que buscam determinar se o sistema assegura as propriedades que se propõe a cumprir. Caso sejam encontradas faltas nessa fase, então é realizado o diagnóstico, visando determinar as causas e é feita a correção dessa falta. Após a correção, normalmente são aplicados testes para verificar se o sistema permanece funcionando, como testes de regressão [Leung e White 1989]. Durante a fase de projeto, podem ser aplicadas técnicas de remoção estáticas (por exemplo, verificação formal) ou dinâmicas (por exemplo, simulações). Já durante a fase de operação do sistema, são aplicados mecanismos de manutenção corretiva e preventiva. Enquanto que

a manutenção corretiva visa remover faltas que já produziram erros no sistema, a manutenção preventiva é aplicada visando descobrir e remover faltas antes que estas causem erros [Avizienis et al. 2004].

Previsão de faltas A previsão de faltas ocorre pela avaliação do comportamento do sistema com relação à ocorrência de faltas [Laprie, Avizienis e Kopetz 1992]. Tal avaliação possui duas abordagens: qualitativa, que avalia e classifica as possíveis ocorrências de faltas e combinações de eventos que podem levar a ocorrência de faltas, e quantitativa, que realiza uma avaliação em termos probabilísticos.

2.1.4 Tolerância a Faltas em Sistemas Distribuídos

Dada a complexidade de sistemas computacionais em geral, evitar a ocorrência de faltas se torna uma tarefa muito complicada. Tais sistemas são, em geral, formados por diversos componentes (muitas vezes de diferentes fabricantes) interligados que se comunicam visando a realização de tarefas. A ocorrência de uma falta em um desses componentes, ou na comunicação entre eles, pode comprometer o funcionamento do sistema computacional como um todo.

Dada a severidade de possíveis conseqüências e a impossibilidade de evitar a ocorrência de todas as faltas, uma abordagem diferente das acima apresentadas consiste em os sistemas tolerarem a existência de uma falta, isto é, aceitar que faltas podem ocorrer, e evitar que tais faltas gerem falhas no sistema. Tal abordagem é chamada de Tolerância a Faltas [Avižienis 1967]. A utilização de técnicas de redundância é fundamental para a obtenção de um sistema tolerante a faltas [Randell 1975, Verissimo e Rodrigues 2001]. A redundância pode ser utilizada através de três abordagens: *redundância espacial*, que consiste na replicação de componentes, *redundância temporal*, que consiste na repetição da realização de uma mesma tarefa, e *redundância de valor*, que consiste na adição de informações de controle adicionais aos dados a serem utilizados [Verissimo e Rodrigues 2001].

Uma das formas de concretização da tolerância a faltas se dá através do processamento de erros, o qual possui três abordagens: detecção de erros, recuperação de erros e mascaramento de erros. Destas, focaremos no *mascaramento de erros*, visto que é a abordagem que mais interessa para o entendimento deste trabalho. O mascaramento de erros consiste em prover redundância suficiente em um sistema, de modo que a falha de um dos componentes redundantes não seja percebida por seu usuário. Ou seja, após a ocorrência de uma falha, um componente sobressalente assume o controle das operações, de modo que tal falha não venha a ser percebida, não interrompendo o serviço para seu usuário.

Dada a dificuldade em evitar a ocorrência de faltas, em geral, a descrição de protocolos distribuídos tolerantes a faltas é feita através de sua *Resistência* (*Resilience*), que expressa o número máximo de servidores do serviço replicado que podem falhar para que o serviço se mantenha correto [Correia 2005]. Esse parâmetro é diretamente dependente das classes de faltas toleradas, bem como das premissas de sincronia adotadas no modelo de sistema. Por exemplo, em sistemas distribuídos assíncronos que seguem a abordagem Replicação Máquina de Estados (detalhada na Seção 2.1.4.1), que se comunicam através da passagem de mensagens pela rede, onde é admitido que os processos apresentem comportamento bizantino [Lamport, Shostak e Pease 1982], o número máximo de faltas toleradas é $f = [\frac{N-1}{3}]$, sendo N o número total de servidores [Bracha e Toueg 1985]. Já no caso de faltas do grupo de faltas *Omissivas*, o número máximo de faltas toleradas é $f = [N - 1]$. Quando se trata de faltas de valor, a resistência de um sistema e $f = [\frac{N-1}{2}]$. Outra forma de se expressar os valores apresentados acima é caracterizar a resistência de um sistema através do número mínimo de réplicas (N) que o sistema deve possuir para que tolere f réplicas faltosas. Para faltas bizantinas, temos que $N \geq 3f + 1$, para faltas de valor, $N \geq 2f + 1$, e para faltas omissivas, $N \geq f + 1$.

No caso de algumas (f) réplicas sofrerem faltas, são necessários mecanismos para que o cliente não utilize os resultados providos por réplicas faltosas em sua execução. Um dos mecanismos utilizados é a *votação*, que utiliza o conceito de *quorum*. Um *quorum* é utilizado de modo que um cliente necessite de um *quorum* de $f + 1$ respostas iguais vindas de diferentes réplicas. Ou seja, o *quorum* necessário de respostas com conteúdo idêntico corresponde a um número maior que o número de faltas tolerada pelo sistema, garantindo assim que dentro do conjunto de respostas recebidas, ao menos uma é garantidamente procedente de uma réplica correta. Assim, o cliente realiza uma votação sobre as respostas recebidas e define como resultado correto aquele que ocorrer em um conjunto mínimo de $f + 1$ mensagens.

As próximas Seções (2.1.4.1, 2.1.4.2, 2.1.4.3, 2.1.4.4) apresentam uma revisão sobre algumas técnicas de replicação bastante utilizadas para tolerância a faltas em sistemas distribuídos. Dentre essas, a técnica conhecida como Replicação Máquina de Estados (Seção 2.1.4.1) é a mais importante para este trabalho, visto que é nela que este se baseia, bem como a maioria das arquiteturas e protocolos para tolerância a faltas bizantinas existentes [Castro e Liskov 1999, Castro e Liskov 2000, Castro e Liskov 2002, Yin et al. 2003, Reiser et al. 2006, Kotla et al. 2007, Luiz et al. 2007, Veronese et al. 2008]. Além das abordagens de replicação apresentadas nas seções referidas acima, o Capítulo 3 apresenta o con-

junto de trabalhos com os quais o estudo apresentado neste documento se relaciona, todos os quais voltados para a tolerância a faltas bizantinas em sistemas distribuídos baseados na técnica de Replicação Máquina de Estados.

2.1.4.1 Replicação Máquina de Estados

No âmbito de Sistemas Distribuídos, um dos principais paradigmas relativos a Tolerância a Faltas é a abordagem *Replicação Máquina de Estados* (RME) [Schneider 1990]. Tal abordagem é caracterizada por garantir as propriedades do serviço empregando a abordagem de redundância espacial, mais especificamente, através da replicação da máquina que oferece o serviço, o servidor. A RME é definida em [Schneider 1990] como um método genérico para a implementação de serviços tolerantes a faltas através da replicação de servidores e coordenação da interação dos clientes com as réplicas servidoras. Assim, a construção de sistemas distribuídos que envolvem replicação pode ser facilitada realizando a especificação desses em termos de máquinas de estado.

Em [Schneider 1990], uma máquina de estados é definida através de sua caracterização semântica: "*as saídas de uma máquina de estados são unicamente determinadas pela seqüência de requisições que esta processa, independentemente de tempo e de qualquer outra atividade no sistema*". Ou seja, uma máquina de estados é um componente de comportamento determinístico, cujo estado e saídas dependem apenas de seu estado inicial e do conjunto de requisições que esta processou. Para garantir que as máquinas de estados de um sistema replicado mantenham sempre seus estados mutuamente consistentes, é necessário que as requisições enviadas pelos clientes sejam recebidas e processadas na mesma ordem por todas as réplicas, que é o conceito de *Coordenação de Réplicas*, o qual pode ser decomposto nos seguintes requisitos [Schneider 1990]:

- **Acordo**: todas as máquinas de estados corretas recebem todas as requisições.
- **Ordem Total**: todas as máquinas de estados corretas processam todas as requisições recebidas na mesma ordem.

Assim, através da abordagem RME, um serviço pode ser implementado como um sistema distribuído, composto por servidores que replicam o serviço (tanto os dados como as instruções), todos tendo o serviço a ser oferecido implementado como uma máquina de estados determinística. Assim, cada comando enviado pelos clientes para as réplicas de serviço deve ser enviado utilizando um protocolo de difusão atômica, para

garantir que todas as réplicas corretas recebam as mesmas requisições e na mesma ordem. Após o recebimento, as réplicas executam os comandos contidos nas requisições, modificando ou não seus estados internos e enviando a resposta para a requisição ao cliente. Dessa forma, as réplicas de serviço mantêm seus estados consistentes. Um dos grandes desafios atuais é a criação de protocolos que garantam o Acordo e a Ordem Total na abordagem RME, o qual é também o objetivo do presente trabalho, assim como outros trabalhos anteriores [Castro e Liskov 1999, Castro e Liskov 2000, Castro e Liskov 2002, Yin et al. 2003, Chun et al. 2007, Kotla et al. 2007, Luiz et al. 2007, Veronese et al. 2008]. A combinação da técnica de RME com um protocolo de difusão atômica (*atomic multicast*) para envio das requisições dos clientes é também conhecida como *Replicação Ativa* [Guerraoui e Schiper 1996, Verissimo e Rodrigues 2001].

Um conceito fundamental para a garantia da resistência em sistemas distribuídos que utilizam a abordagem RME é a *Diversidade de Projeto* [Avizienis e Kelly 1984, Hiltunen, Schlichting e Ugarte 2003], que consiste em projetar e implementar um serviço seguindo várias abordagens, independentes entre si, ou seja, cada serviço sendo projetado e implementado por uma equipe diferente, e preferencialmente, geograficamente dispersas. Isso diminui a probabilidade de as máquinas de estados na abordagem RME apresentarem as mesmas falhas de projeto e implementação, evitando que a descoberta de uma falha em uma máquina signifique o comprometimento de todas as máquinas. Além disso, a utilização de equipamentos de hardware, de sistemas operacionais e de linguagens de programação diferentes em cada máquina também contribui para que as faltas ocorram de forma independente nas máquinas que compõem um sistema replicado, o que caracteriza o conceito de *Independência de Faltas*.

2.1.4.2 Replicação Semi-ativa

Assim como na abordagem de Replicação Ativa, na abordagem *Replicação Semi-ativa*, o cliente submete a requisição para todas as réplicas. Porém, uma réplica específica, também chamada de *Líder*, é quem determina a ordem na qual essas mensagens serão processadas. Após determinar tal ordem, esta envia uma mensagem para as réplicas restantes, também chamadas de *Seguidoras*, contendo a ordem de execução para as requisições recebidas. Então, todas as réplicas executam as requisições recebidas na mesma seqüência, a qual foi determinada pelo Líder, que é a única réplica que envia a resposta ao cliente. Também conhecida como *Líder-Seguidor* [Verissimo e Rodrigues 2001], essa abordagem se caracteriza pela existência de categorias diferentes de réplicas. Enquanto

na abordagem de Replicação Ativa todas as réplicas assumem o mesmo papel, na Replicação Semi-ativa existe uma divisão, sendo necessária a definição de uma das réplicas para assumir o papel de líder.

2.1.4.3 Replicação Primário-*Backup*

Diferentemente das abordagens apresentadas nas Seções 2.1.4.1 e 2.1.4.2, na abordagem *Primário-Backup*, apenas uma das réplicas realiza o processamento da requisição do cliente [Budhiraja et al. 1993], por isso, essa técnica é também conhecida como *Replicação Passiva*. Assim, o cliente envia a requisição para apenas uma das réplicas, conhecida como *Primária*. Tal réplica executa a requisição, altera seu estado e envia uma mensagem para as réplicas *backup*, contendo o estado resultante da execução da requisição, para que tais réplicas possam armazená-las em seus estados internos. Ao receber tal mensagem, as réplicas *backup* atualizam seus estados internos e enviam uma mensagem de confirmação de mensagem recebida para a primária, a qual então envia a resposta para o cliente.

Essa abordagem faz uso de detectores de falhas para que seja possível determinar quando a réplica primária sofreu um *crash*. Tal mecanismo de detecção pode ser implementado da seguinte forma, para um cenário que tolera a falta de apenas uma máquina [Budhiraja et al. 1993]: a primária envia periodicamente uma mensagem de notificação (vazia) para a máquina *backup*, de modo que esta última possa saber que a primária permanece viva. A cada τ segundos, a primária deve enviar uma mensagem para a *backup*. Assim, o tempo entre duas mensagens de notificação é $\tau + \delta$, sendo δ a latência para que uma mensagem vazia trafegue de uma máquina para outra [Budhiraja et al. 1993]. Se uma mensagem de notificação não for recebida pelo *backup* após $\tau + \delta$ segundos, a réplica *backup* assume o papel da réplica primária e informa os clientes sobre tal mudança.

Essa abordagem é bastante vantajosa no quesito custo, visto que não é realizada redundância de processamento, sendo necessário o processamento por apenas uma das réplicas, enquanto que as réplicas restantes permanecem inativas, realizando apenas as ações de coleta de estado e de detecção de falha da primária, que são tarefas que exigem muito menos poder de processamento do que a execução de um serviço completo.

2.1.4.4 Replicação Semi-Passiva

A abordagem de *Replicação Semi-Passiva* [Défago, Schiper e Sergent 1998] é apresentada como uma proposta visando diminuir os custos da utilização de Replicação Passiva. Tais custos estão envolvidos principalmente no mecanismo para que as réplicas entrem em acordo sobre qual

réplica é a primária, ou seja, um mecanismo de controle de pertinência de grupo. A abordagem Semi-Passiva não necessita de tal mecanismo, pois apresenta como diferencial a utilização do *paradigma do coordenador rotativo* [Chandra e Toueg 1996].

Diferentemente da Replicação Passiva, onde os clientes enviam as requisições apenas para a réplica primária, na Replicação Semi-Passiva, os clientes não têm conhecimento sobre qual réplica cumpre o papel de primária, enviando suas requisições para todas as réplicas e recebendo respostas de todas elas. Apesar de todas as réplicas receberem as requisições, apenas a primária realiza o processamento da requisição, e repassa o estado resultante do processamento para as réplicas *backup*, as quais aceitam tal valor como resposta para a requisição do cliente. Quando ocorre o *crash* na primária, uma *backup* (eleita através do paradigma de coordenador rotativo) assume o papel da primária e executa o protocolo recém descrito para as mensagens ainda não processadas.

2.1.5 Tolerância a Intrusões

O conceito de Tolerância a Intrusões (TI) foi apresentado inicialmente há mais de duas décadas [Fraga e Powell 1985]. Esta área de pesquisa tornou-se bastante ativa na última década [Verissimo, Neves e Correia 2003, Sousa et al. 2007, Bessani et al. 2007, Bessani et al. 2008, Reiser e Kapitza 2007, Stumm Júnior et al. 2009], sendo alvo de diversas pesquisas e de grandes projetos relacionados à confiança no funcionamento de sistemas distribuídos. A TI consiste na aplicação do paradigma de tolerância a faltas no domínio da segurança [Verissimo, Neves e Correia 2003, Correia 2005]. Ou seja, consiste em aceitar que o sistema apresenta vulnerabilidades eternamente, podendo sofrer ataques que podem obter sucesso, mas garantir que o sistema como um todo não apresente falha em sua operação e segurança.

Para descrever a manifestação de intrusões em um sistema computacional, [Verissimo, Neves e Correia 2003] apresentou o modelo de faltas composto AVI (*Attack-Vulnerability-Intrusion*), que descreve uma intrusão como resultado de um ataque que obteve sucesso em explorar uma vulnerabilidade, dando origem à relação:

$$\text{Ataque} \rightarrow \text{Vulnerabilidade} \rightarrow \text{Intrusão} \rightarrow \text{Erro} \rightarrow \text{Falha}$$

É importante perceber que tanto ataques, quanto vulnerabilidades e intrusões são consideradas faltas, cada uma em sua camada. Uma *vulnerabilidade* é uma falta existente nos componentes do sistema, seja em sua especificação, projeto, implementação, configuração. Um *ataque* é uma falta de interação que busca explorar uma ou mais vulnerabilidades,

de forma maliciosa. Uma *intrusão* é o resultado de um ataque que obteve sucesso ao explorar as vulnerabilidades do sistema. Tal intrusão pode gerar um erro no estado do sistema, que, caso não sejam utilizadas técnicas de mascaramento, pode se propagar em uma falha do sistema atacado. Assim, a tolerância a intrusões é aplicada, de modo a evitar que um erro gerado por uma intrusão se propague em uma falha.

A descrição acima apresentada, além de auxiliar no entendimento do processo de manifestação de uma intrusão, também auxilia no entendimento dos mecanismos que podem ser utilizados para prevenção. Por exemplo, para evitar a ocorrência de ataques, podem ser utilizados mecanismos de prevenção de ataques, bem como metodologias de desenvolvimento que previnam o surgimento de vulnerabilidades ou mecanismos de prevenção de intrusões. Tais mecanismos, utilizados conjuntamente, podem ajudar a prevenir a ocorrência de intrusões.

Em se tratando de uma falta de origem maliciosa, o termo *intrusão* é utilizado com freqüência como sinônimo de *falta bizantina*. No presente trabalho, tanto intrusão como falta bizantina se referem a tentativas maliciosas que obtiveram sucesso ao explorar uma ou mais vulnerabilidades de um sistema. Em alguns sistemas, os protocolos de BFT tradicionais não são suficientes para prover tolerância a intrusões, pois não oferecem meios para garantir a *confidencialidade*, requisito que pode ser essencial para determinados sistemas. Para garantir que um intruso não consiga acesso a dados confidenciais armazenados no servidor atacado, além de técnicas de criptografia básicas para ocultar os dados, pode ser necessário o uso de outros mecanismos. Uma abordagem proposta em [Fraga e Powell 1985] ataca esse problema realizando a *fragmentação* dos dados através de vários servidores, de modo que um intruso não obtenha informação significativa ao invadir apenas um dos servidores. Outra abordagem utilizada é a *partilha de segredos* [Shamir 1979], que consiste em evitar que f processos maliciosos consigam acessar determinados dados secretos, sendo necessários no mínimo $f + 1$ processos combinados para revelar os dados.

O presente documento apresenta uma arquitetura para tolerância a intrusões, voltada ao problema da consistência entre servidores replicados, provendo tolerância a intrusões de forma a aumentar a resistência a ataques maliciosos de uma única máquina física. Assim, através da utilização de virtualização, replicamos o serviço em várias VMs, de modo que uma entidade maliciosa não consiga corromper o serviço apenas através da invasão de um sistema, como seria o caso de um serviço único sendo executado sobre a máquina física. A questão da confidencialidade não é tratada nesse trabalho.

2.1.6 Consenso em Sistemas Distribuídos

Em ambientes de computação distribuída, a obtenção de consenso sobre determinados valores é de fundamental importância para a manutenção da consistência desses sistemas. O problema do consenso consiste em um grupo de processos que busca obter um valor de decisão comum a todos, onde cada processo realiza sua proposta, sendo que todos os processos decidem pelo mesmo valor. O problema do consenso é definido através das seguintes primitivas:

- ***propose(v)***: propõe o valor v para os outros processos.
- ***decide(v)***: significa que o processo decidiu pelo valor v.

Em [Chandra e Toueg 1996], o problema do consenso foi definido através das seguintes propriedades, as quais devem ser cumpridas para que o problema obtenha solução:

- **Terminação**: todo processo correto acaba por decidir um valor.
- **Integridade Uniforme**: todo processo correto decide apenas uma vez.
- **Acordo**: todos os processos corretos decidem pelo mesmo valor.
- **Validade Uniforme**: se um processo decidir por um valor, então esse valor foi proposto por algum processo.

Para sistemas distribuídos puramente assíncronos, foi provado em [Fischer 1983] que o problema do consenso não apresenta solução quando da presença de no mínimo um membro que sofre falta de *crash*. Tal prova é conhecida como impossibilidade FLP. Apesar de tal impossibilidade, vários trabalhos apresentaram propostas para a concretização do consenso em ambientes distribuídos, através do relaxamento das premissas de sincronia. Embora relaxadas, tais premissas são mantidas de forma prática e realista, de modo que tais sistemas possam ser implementados na prática.

Em ambientes baseados em RME, o consenso é parte fundamental para que os sistemas mantenham as propriedades descritas na Seção 2.1.4.1. Por exemplo, para garantia da ordem total, requerida pela primeira propriedade, é necessário que as máquinas de estado cheguem ao consenso sobre a ordem de execução das requisições emitidas pelos clientes.

2.1.7 Acordo Bizantino

Outro problema, também relacionado à coordenação em sistemas distribuídos é o Acordo Bizantino, onde são considerados também membros com comportamento bizantino. O problema foi descrito como *O Problema dos Generais Bizantinos* [Lamport, Shostak e Pease 1982], que nada mais é do que uma abstração para o problema do acordo em sistemas distribuídos quando da presença de membros maliciosos. Diferentemente do *Consenso*, onde todos os membros fazem suas propostas de valor de decisão, no *Problema dos Generais Bizantinos* apresentado em [Lamport, Shostak e Pease 1982], apenas um general envia a proposta de decisão para os seus $n - 1$ tenentes. Considerando que tanto o general quanto os tenentes podem ser traidores, que eles se comunicam apenas através de mensageiros, o objetivo é que os membros honestos cheguem a um plano comum de ação sobre atacar ou não a cidade inimiga. Assim, o problema foi descrito através das condições de *Consistência Interativa* [Lamport, Shostak e Pease 1982]:

- **IC1**: todos os tenentes leais obedecem à mesma ordem.
- **IC2**: se o general é leal, então, todos os tenentes leais devem seguir sua ordem.

Para atingir o acordo sobre o plano de ação, os tenentes se comunicam visando verificar se todos receberam a mesma ordem do general, repassando mensagens do tipo "*eu recebi a ordem do general para atacar/retirar*". Duas situações podem ocorrer:

1. **General traidor**: envia mensagem de atacar para um tenente e de retirar para outro.
2. **Tenente traidor**: mente para os outros tenentes sobre a mensagem que recebeu do General.

Para ambos os casos, foi provado [Lamport, Shostak e Pease 1982] que, para $n < 3f + 1$, é impossível que os membros cheguem a um plano comum quando da presença de f traidores.

2.1.8 Considerações sobre Confiança no Funcionamento

Os conceitos apresentados nas seções anteriores são de fundamental importância para a compreensão do trabalho aqui apresentado. Dentre todos, o conceito de Tolerância a Faltas recebeu um foco especial, visto que nossa proposta faz uso de técnicas de tolerância a faltas para prover

Confiança no Funcionamento. Técnicas de Replicação Máquina de Estados são aplicadas para a concretização do serviço replicado. Além da RME, aplicada em nível de máquinas virtuais em nosso trabalho, também apresentaremos uma abordagem que se utiliza de um modelo híbrido de falhas, utilizando replicação tolerante a faltas de *crash*, com a abordagem primário-*backup*, em nível de máquina física, conforme será detalhado posteriormente no Capítulo 4.

A próxima seção trata dos aspectos básicos que devem ser levados em consideração quando da construção de sistemas distribuídos, os quais são fundamentais para a especificação de nossa proposta.

2.2 Modelos de Sistemas Distribuídos

Sistemas distribuídos são mais facilmente compreendidos quando detalhados através de suas características principais. Dessa forma, a descrição de um sistema distribuído pode ser feita através de um *Modelo de Sistema*, o qual descreve somente os elementos fundamentais relativos a este. Dentre esses elementos, encontram-se as premissas básicas sobre as quais a corretude do sistema está apoiada, bem como as características fundamentais sobre as quais o sistema está sustentado. Além disso, a descrição do modelo permite a verificação formal do cumprimento das garantias que o sistema se compromete a cumprir. Detalhes específicos, como por exemplo, especificações de hardware, são abstraídos dessa descrição, de forma a focar apenas nos detalhes que devem ser garantidos para que a corretude do sistema seja assegurada.

O presente documento irá seguir a abordagem de descrição do modelo de um sistema distribuído apresentado em [Coulouris, Dollimore e Kindberg 2005], segundo a qual os elementos fundamentais desse sistema podem ser descritos sob três pontos de vista diferentes, cada um deles focando em um aspecto específico: modelo de interação, modelo de falhas e modelo de segurança, apresentados a seguir. O modelo de sistema adotado para o projeto de SMIT é apresentado, seguindo a mesma abordagem, na Seção 4.2.

2.2.1 Modelo de Interação

O *Modelo de Interação* de um sistema distribuído descreve as formas de comunicação entre os processos que compõem esse sistema, bem como discute as limitações existentes nos canais por onde os processos interagem.

Em geral, os processos de um sistema distribuído se comunicam através da passagem de mensagens por um canal de comunicação que interliga tais processos. A execução de protocolos distribuídos, em ge-

ral, requer a transmissão de mensagens entre os processos, seja para uma simples transferência de dados, ou para tarefas relativas à coordenação de suas ações. Dada a importância dos canais de comunicação para o funcionamento e performance de um sistema distribuído, é necessário definir o modelo de interação mais apropriado para um sistema. Os sistemas distribuídos podem ser divididos em duas categorias, de acordo com as hipóteses realizadas sobre o tempo de processamento, de transporte de mensagens e dos relógios internos dos processos [Coulouris, Dollimore e Kindberg 2005]:

- **SDs Síncronos** [Hadzilacos e Toueg 1994]:
  - Existe um limite conhecido para o tempo de execução que um processo leva para executar um passo de processamento.
  - Cada processo possui um relógio local que apresenta uma taxa de desvio limitada e conhecida.
  - Existe um limite conhecido para o tempo de atraso de uma mensagem sobre qualquer tipo de canal.
- **SDs Assíncronos** [Hadzilacos e Toueg 1994]: são sistemas nos quais não existe a possibilidade de realização de hipóteses temporais sobre o comportamento dos seus processos.

Um fator que possui um impacto enorme no desenvolvimento de protocolos distribuídos é a dificuldade para o estabelecimento de um relógio global. Os processos componentes de um sistema distribuído possuem seus próprios relógios locais, os quais apresentam uma taxa de deslize (*clock drift rate*), que, segundo [Coulouris, Dollimore e Kindberg 2005] é a quantidade relativa de tempo que um relógio difere de um relógio de referência perfeito. Além de apresentarem taxas de deslize com relação a um relógio de referência, essas taxas variam de máquina para máquina, dificultando assim a manutenção dos relógios dos processos sincronizados.

A ordenação de eventos em um sistema distribuído seria um problema resolvido caso os relógios dos processos componentes fossem sincronizados de forma exata, pois bastaria ordená-los de acordo com as marcações do tempo em que eles ocorreram nos processos de origem. Dada a impossibilidade de tal sincronização, várias abordagens foram propostas para realizar ordenação lógica dos eventos. A primeira abordagem a utilizar essa noção de Relógios Lógicos foi [Lamport 1978], na qual os eventos são numerados através de contadores monotônicos e enviados para outros processos, os quais mantêm seus contadores de acordo com os

valores recebidos de outros processos e de acordo com as regras definidas pelo algoritmo. Porém, essa proposta não garante que um evento e, que possui um contador lógico menor que outro evento e', tenha realmente ocorrido antes de e'.

2.2.2 Modelo de Falhas

O modelo de falhas especifica os modos como os processos e canais de comunicação que compõem um sistema distribuído podem ter o seu comportamento desviado da sua especificação correta (o que caracteriza uma falha) e quais os efeitos que essas falhas têm sobre o sistema como um todo. O modelo de falhas possibilita uma melhor compreensão sobre o grau de confiabilidade que um sistema possui e permite que os sistemas sejam especificados de forma mais concisa e padronizada. Em [Hadzilacos e Toueg 1994], as falhas que os processos e canais de comunicação podem sofrer foram divididas da seguinte forma:

- ***Crash***: um processo sofre uma falha de *crash* quando este executa um passo após o qual passo nenhum será executado, ou seja, o processo pára a sua execução.
- **Omissão**:
  - **Envio**: um processo comete uma falha de omissão no envio de uma mensagem m quando este completa o envio de m, mas m não é inserido em seu *buffer* de mensagens de saída. Assim ocorre uma omissão no envio da mensagem, visto que m não é realmente enviada ao destinatário.
  - **Recepção**: um processo comete uma falha de omissão no recebimento de uma mensagem m, se m foi inserida em seu *buffer* de mensagens de entrada, mas ele não recebe m. Assim, ocorre uma omissão na recepção da mensagem, visto que o processo não recebe efetivamente a mensagem.
- **Arbitrária**: uma falha arbitrária é cometida por um processo quando este executa uma seqüência de passos arbitrariamente diferente de sua especificação, ou seja, o processo se comporta de forma arbitrária.

2.2.3 Modelo de Segurança

O modelo de segurança descreve o sistema através dos objetos que compõem o sistema, bem como as ameaças a esses objetos. Os processos em sistemas distribuídos interagem visando atingir um objetivo comum.

Em geral, essa interação é feita através do envio de mensagens. Algumas preocupações devem ser levadas em consideração na interação entre dois processos s e c, sendo s o servidor e c o cliente. Quando s recebe uma mensagem m de c solicitando a execução de uma determinada ação, s e c devem se preocupar com as seguintes questões:

1. **s**: m foi realmente remetida por c?
2. **s**: m não foi alterada por outra entidade durante a transmissão?
3. **s**: c possui direitos de executar a operação requerida em m?
4. **c**: m não foi alterada por outra entidade durante a transmissão?
5. **c**: O conteúdo de m poderá ser lido por alguém que não seja s?
6. **c**: Dados sigilosos contidos em m foram expostos?

A questão 1 diz respeito à *Autenticidade* da mensagem, ou seja, se a mensagem não possui um campo *remetente* falsificado. Um atacante que consegue falsificar uma mensagem com sucesso (ataque conhecido como *spoofing*), poderá executar operações se passando por outras entidades.

As questões 2 e 4 dizem respeito à *Integridade* da mensagem. Uma entidade que ataca o canal de comunicação e consegue modificar uma mensagem com sucesso, pode, por exemplo, fazer com que uma mensagem que contém uma operação de leitura se transforme em uma operação de escrita.

A questão 3 diz respeito ao *Controle de Acesso*. É necessário que exista um mecanismo de controle de acesso implementado no serviço, de modo que somente membros autorizados a realizar determinadas operações o façam.

As questões 5 e 6 dizem respeito à *Confidencialidade*. Uma entidade que ataca o canal de comunicação poderia interceptar a mensagem e ler o seu conteúdo, o que pode acarretar na publicação de informações sigilosas.

Todas as questões acima descritas, bem como a descrição da maneira como foram resolvidas nesta proposta são detalhados na Seção 4.2.3.

2.2.4 Considerações sobre Modelos de Sistema

A descrição de um sistema distribuído através de seu modelo é muito importante para que seu entendimento seja mais preciso. Através dessa descrição, é possível explicitar e detalhar precisamente o ambiente, os requisitos básicos e as hipóteses que o sistema necessita para que sua

execução seja correta. A descrição do modelo de sistema de SMIT é apresentada na Seção 4.2, detalhando o ambiente em que SMIT é executado, as formas de interação entre os processos, as falhas em seus componentes que são toleradas, bem como técnicas utilizadas para a garantia da segurança das informações.

2.3 Tecnologia de Virtualização

Embora diversas definições tenham sido apresentadas com o passar dos anos, a definição de **máquina virtual** como "*uma cópia isolada e eficiente de uma máquina real*" [Popek e Goldberg 1974] tem sido uma das mais utilizadas. Outra definição para um sistema de máquinas virtuais é "*um sistema de computação no qual as instruções emitidas por um programa podem ser diferentes daquelas realmente executadas pelo hardware*" [Parmelee et al. 1972].

Embora seja um conceito antigo, introduzido em meados da década de 1960 [Goldberg 1974, Chisnall 2007], a virtualização voltou à tona como um recurso importante para a computação no final da década de 1990, após o lançamento de tecnologias de virtualização capazes de funcionar sob plataforma *x86* [Rosenblum 2004]. Apesar de inicialmente concebidos visando o particionamento lógico de *mainframes* em vários sistemas virtuais, atualmente, sistemas virtualizados são bastante aplicados até mesmo em computadores pessoais. Em ambos os casos, o conceito foi aplicado com um objetivo comum: diminuir a subutilização de recursos de hardware, o que era comum nos antigos *mainframes* e passou a se tornar corriqueiro no final dos anos 1990 também em computadores pessoais, devido ao grande aumento do poder computacional fornecido por estes últimos. Assim, dada a grande capacidade computacional de computadores pessoais e o suporte a virtualização inserido nos processadores mais recentes, o mercado de virtualização de sistemas voltou a ser bastante atrativo [Rosenblum 2004]. Além do interesse em um melhor aproveitamento dos recursos oferecidos por um computador pessoal, o uso de virtualização tem sido bastante aplicado à área de gerenciamento de servidores de rede [Padala et al. 2007] e à área de segurança de sistemas computacionais [Laureano, Maziero e Jamhour 2004, Garfinkel e Rosenblum 2003].

Apesar de as definições acima apresentadas serem referentes a um tipo específico de máquinas virtuais, a utilização do termo causa certa ambigüidade, pois existem duas categorias de máquinas virtuais. É importante perceber a diferença entre as duas abordagens e evidenciar qual delas está sendo tratada. As duas categorias foram definidas em [Smith e Nair 2005] como:

- **Máquina Virtual de Processo**: é uma plataforma virtual que executa um processo individual. Tal máquina virtual é instanciada unicamente para suporte ao processo, sendo criada e terminada juntamente ao processo. Exemplos: *Java Virtual Machine* [Lindholm e Yellin 1999], *Common Language Runtime* (*Framework* .NET) [Thai e Lam 2002].

- **Máquina Virtual de Sistema**: provê um ambiente completo e persistente que suporta um sistema operacional completo e seus processos. Fornece ao sistema convidado um conjunto de recursos que compõem o hardware virtual. Exemplos: KVM [Qumranet 2006], Xen [Barham et al. 2003], Virtualbox [Microsystems 2009].

No presente documento, sempre que nos referirmos a máquinas virtuais e virtualização, estaremos nos referindo a máquinas virtuais (VMs) de sistema, a não ser que tenha sido explicitamente declarado o contrário.

Através da tecnologia de virtualização, uma única máquina física (por exemplo, um *mainframe* ou mesmo um computador pessoal) pode se apresentar aos usuários como várias Máquinas Virtuais (VM) separadas. A camada que fornece tal ilusão é chamada de Monitor de Máquinas Virtuais (VMM - *Virtual Machine Monitor*), a qual é, tradicionalmente, uma camada situada sobre o hardware, a qual exporta uma abstração de máquina virtual bastante semelhante ao hardware, de modo que um software escrito para tal hardware possa ser executado, sem problemas, sobre a máquina virtual [Rosenblum e Garfinkel 2005]. Os sistemas operacionais que executam sobre uma máquina virtual são tradicionalmente chamados de **convidados**, enquanto que o sistema que hospeda o VMM é conhecido como **anfitrião**. As próximas subseções apresentam alguns detalhes relevantes sobre máquinas virtuais e sobre sua importância para o trabalho aqui apresentado.

2.3.1 Propriedades dos VMMs

Os VMMs são os sistemas responsáveis pelo gerenciamento da execução das máquinas virtuais. Dada sua importância para a correta execução de máquinas virtuais, em [Popek e Goldberg 1974] foram descritas três propriedades fundamentais, as quais um sistema de controle de VMs deve prover, de forma a ser considerado um Monitor de Máquinas Virtuais (VMM):

- **Eficiência**: toda e qualquer instrução inofensiva deve ser executada diretamente pelo hardware, sem intervenção do VMM.

- **Controle de Recursos**: o VMM possui total controle sobre os recursos a serem oferecidos para as máquinas virtuais.
- **Equivalência**: qualquer programa K, executando com um VMM, deve apresentar comportamento idêntico à sua execução quando não há a presença do VMM.

Além destas, outras propriedades desejáveis para os VMMs foram descritas posteriormente [Garfinkel e Rosenblum 2003]:

- **Isolamento**: um software, executado em uma VM, não consegue acessar ou modificar o VMM ou outra VM.
- **Inspeção**: todo o estado das VMs está disponível ao VMM.
- **Interposição**: o VMM deve intervir em determinadas operações realizadas por máquinas virtuais, como a execução de instruções privilegiadas.

2.3.2 Tipos de VMMs

Existem várias formas de se categorizar os Monitores de Máquinas Virtuais. Uma das formas mais populares é dividi-los em duas categorias [King, Dunlap e Chen 2003]:

- **Tipo I**: VMMs que são implementados/executados diretamente sobre o hardware (Figura 2.2(a)).
- **Tipo II**: VMMs que são executados sobre um sistema operacional anfitrião comum (Figura 2.2(b)).

VMMs do Tipo I apresentam como grande vantagem sobre VMMs do Tipo II o desempenho, pois os sistemas operacionais anfitriões atuais não oferecem interfaces para o hardware suficientemente poderosas para suportar os padrões de acesso intensivo exigidos pelos VMMs [King, Dunlap e Chen 2003]. Em contrapartida, a utilização de VMMs do Tipo II apresenta algumas vantagens significativas. O fato de o VMM ser executado como um processo normal dentro do sistema operacional anfitrião, torna possível utilizar todo o poder de tal sistema para realização de tarefas de depuração e monitoração do VMM e das VMs executando sobre ele. Além de VMMs dos Tipos I e II, existem sistemas que empregam uma abordagem híbrida dos dois tipos, utilizando o sistema operacional anfitrião apenas para a realização de operações de entrada e saída, sendo o resto das operações realizadas diretamente sobre o hardware [King, Dunlap e Chen 2003].

(a) VMM Tipo I

(b) VMM Tipo II

Figura 2.2: Tipos de VMMs

2.3.3 Considerações sobre Virtualização

Nas duas últimas décadas, especialmente na última, sistemas virtualizados têm sido utilizados massivamente nas mais diversas áreas de aplicação. Muitas áreas de aplicação também têm surgido nos últimos anos, como por exemplo, o tema principal deste trabalho, que é o uso de virtualização para prover tolerância a faltas, realizando replicação virtual de serviços. Essa área de aplicação tem como objetivo principal diminuir o custo financeiro associado à replicação de serviços em ambientes distribuídos. Assim, é possível replicar um serviço através de vários sistemas operacionais isolados, sem que seja necessária a aquisição de várias máquinas físicas. Além da diminuição considerável no custo de implantação de um sistema replicado (eliminando boa parte das despesas associadas à aquisição de equipamentos), fatores como diminuição no consumo de energia também tornam a utilização de replicação através de máquinas virtuais um fator econômico bastante atrativo. Além dos fatores econômicos, o uso de sistemas virtualizados facilita o gerenciamento de sistemas replicados, visto que o VMM e o sistema anfitrião oferecem diversos recursos para monitoração e controle das atividades da máquinas virtuais.

2.4 Considerações sobre o Capítulo

Este capítulo apresentou os principais conceitos relacionados ao trabalho aqui apresentado. Tais conceitos foram estudados visando consolidar a base teórica necessária para o desenvolvimento da proposta apresentada neste documento. Através dos detalhes estudados sobre *Confiança no Funcionamento*, apresentados na Seção 2.1, foi possível um entendimento mais amplo sobre os problemas relacionados a esta área de estudo, bem como técnicas utilizadas para solucioná-los, os quais formam a base conceitual para qualquer trabalho relacionado à tolerância a faltas.

Na Seção 2.2 foram apresentados detalhes sobre *modelos de sistema* de sistemas distribuídos, os quais são fundamentais para a proposição de sistemas tolerantes a faltas, pois permitem o detalhamento dos principais aspectos relacionados ao sistema, como o tipo de faltas que será tolerado, detalhes sobre o canal de comunicação, comportamento de processos, etc. Para finalizar o conjunto de conceitos básicos relacionados a este trabalho, a Seção 2.3 apresentou conceitos relacionados a *virtualização de sistemas*, a qual é parte importante deste trabalho, visto que a arquitetura aqui proposta se utiliza de características específicas de ambientes virtualizados para o estabelecimento de uma arquitetura tolerante a intrusões.

Capítulo 3

Trabalhos Relacionados

A crescente presença de sistemas computacionais em nossa sociedade traz consigo uma crescente dependência e confiança da sociedade sobre o bom funcionamento desses. Muitos serviços de suporte às atividades básicas do ser humano vem se utilizando de computadores para realização de tarefas, e a ocorrência de uma falta nesses pode tomar proporções catastróficas. Dada a grande conectividade existente, tais sistemas também se tornam expostos à ataques que visam corromper a correção do serviço oferecido. Tais ataques são originados por entidades maliciosas, que possuem a intenção de causar danos. Estudos foram apresentados visando garantir a correção de sistemas computacionais mesmo sob a presença de faltas e entidades maliciosas. Um dos principais trabalhos a tratar o problema de entidades maliciosas interferindo no bom funcionamento de sistemas computacionais é o seminal Problema dos Generais Bizantinos [Lamport, Shostak e Pease 1982], que apresenta uma abstração para o problema de consenso em sistemas distribuídos cujos membros visam executar uma ação em comum.

Desde a formulação inicial do Problema dos Generais Bizantinos [Lamport, Shostak e Pease 1982], vários estudos sobre sistemas distribuídos tolerantes a faltas bizantinas foram apresentados. A proposição de alternativas ao problema do consenso em sistemas distribuídos [Fischer, Lynch e Paterson 1985] foi o alvo da investigação de grande parte desses estudos [Bracha e Toueg 1985, Doudou 1997, Baldoni et al. 2003, Doudou, Garbinato e Guerraoui 2002, Kihlstrom, Moser e Melliar-Smith 2003, Correia et al. 2005, Neves, Correia e Verissimo 2005]. Apesar de se tratar de um problema bastante estudado, apenas nos últimos dez anos é que foram propostas soluções com viabilidade prática [Castro e Liskov 1999, Yin et al. 2003, Correia, Neves e Verissimo 2004, Kotla et al. 2007, Chun et al. 2007, Luiz et al. 2007, Veronese et al. 2008]. Tais propostas possuem como objetivo principal prover arquiteturas e protocolos para estabelecimento de consenso entre membros de um sistema distribuído baseados na abordagem Replicação Máquina de Estados [Schneider 1990]. Além disso, muitos estudos buscaram circunscrever impossibilidades teóricas, como por exemplo a condição FLP [Fischer, Lynch e Paterson 1985], segundo a qual, em um sistema distribuído completamente assíncrono, é impossível que se obtenha o consenso caso um dos membros do sistema possa falhar. Além disso, estudos mais recentes buscaram

circunscrever a necessidade de, no mínimo, $3f + 1$ membros para tolerar f membros bizantinos [Lamport, Shostak e Pease 1982].

De modo a tornar possíveis tais resultados, os estudos realizaram modificações no modelo de sistema, de modo a relaxar algumas hipóteses. Para tornar possível o consenso, muitos trabalhos modificaram a hipótese de assincronia do sistema. Para isso, os modelos de sistema passaram a apresentar um comportamento parcialmente síncrono, onde se assume que as mensagens enviadas serão entregues em algum momento (da expressão inglesa *eventually*) [Castro e Liskov 1999, Kotla et al. 2007]. Assim, são feitas hipóteses de que mensagens que trafegam pela rede terão sua entrega completada em algum momento, fazendo com que os algoritmos de consenso possam confiar em tal hipótese para que possam prover a propriedade de *liveness*. Além disso, de modo a reduzir o número total de réplicas e simplificar o protocolo de consenso, algumas abordagens propuseram o uso de componentes seguros [Correia et al. 2002, Correia, Neves e Verissimo 2004] ou entidades confiáveis [Reiser e Kapitza 2007], através das quais o algoritmo de consenso é executado, formando assim um modelo híbrido de falhas, onde os componentes seguros não estão sujeitos a todos os tipos de falhas, enquanto que os componentes restantes estão sujeitos até mesmo a falhas bizantinas. Através do uso de tais componentes, conseguiu-se reduzir o número total de réplicas necessárias para tolerar f membros faltosos de $3f + 1$ para $2f + 1$.

O presente capítulo faz uma revisão, descrevendo os principais trabalhos relacionados à proposta apresentada nesse documento. Inicialmente, serão descritas duas das mais relevantes propostas para Replicação Máquina de Estados Tolerantes a Faltas Bizantinas, mesma abordagem utilizada no presente trabalho. Em seguida, serão apresentados trabalhos que, assim como o trabalho apresentado neste documento, se utilizam da idéia de modelo híbrido de falhas, que se baseia em assumir que determinados componentes de um sistema estão sujeitos somente a um conjunto restrito de falhas, enquanto que outros componentes estão sujeitos a falhas arbitrárias.

3.1 Técnicas de RME Tolerante a Faltas Bizantinas

A abordagem Replicação Máquina de Estados [Schneider 1990] é largamente utilizada como um meio de prover tolerância a faltas em sistemas computacionais. Através dessa abordagem, é possível modelar serviços como máquinas de estado determinísticas, tornando-os tolerantes a faltas. Dentre os principais trabalhos relacionados ao assunto estão o PBFT [Castro e Liskov 1999], trabalho seminal que impulsionou uma série de trabalhos que apresentam abordagens práticas para BFT, e o

Zyzzyva [Kotla et al. 2007], que apresentou uma nova abordagem que utiliza a noção de execução especulativa, passando parte da responsabilidade pela corretude do sistema para os clientes. Tais trabalhos serão descritos nas próximas seções.

3.1.1 *Practical Byzantine Fault Tolerance*

O PBFT (*Practical Byzantine Fault Tolerance*) [Castro e Liskov 1999] foi um dos primeiros trabalhos a focar os esforços na proposta de um protocolo para tolerância a faltas bizantinas com viabilidade prática. O PBFT segue a abordagem Replicação Máquina de Estados [Schneider 1990], sendo assim composto por um conjunto de máquinas que executam um serviço determinístico replicado, bem como os protocolos PBFT, que, juntos, têm por objetivo formar um sistema robusto e seguro. Além do protocolo de consenso, que assegura que todas as réplicas mantenham um estado consistente, são necessários vários protocolos auxiliares para manter a corretude do sistema. Compõem esse conjunto os seguintes protocolos: consenso, mudança de visão, *checkpointing*, *garbage collection*, bem como o lado do cliente, os quais serão descritos posteriormente.

Voltado para a execução de serviços determinísticos em ambientes distribuídos, o PBFT apresenta uma série de premissas que devem ser cumpridas para que o protocolo permaneça correto. O modelo é baseado em um sistema distribuído assíncrono, onde os membros são conectados através de uma rede, que pode perder, atrasar, duplicar e entregar as mensagens fora de ordem. Os membros se comunicam unicamente através de passagem de mensagens por esta rede. Conforme o título já deixa claro, o PBFT utiliza um modelo de faltas bizantino, assumindo que os nós sofrem faltas independentemente uns dos outros. O conjunto de servidores, que executam o protocolo PBFT e o serviço replicado, é composto por, no mínimo, $3f + 1$ máquinas, sendo que destas, apenas f apresentam comportamento bizantino. Não há limitações quanto ao número de clientes bizantinos.

Apesar de o modelo de sistema ser assíncrono, o PBFT necessita que o sistema apresente certo grau de sincronia para que possa garantir a propriedade de *liveness*, dada a impossibilidade de resolução do problema de consenso em um sistema distribuído totalmente assíncrono [Fischer, Lynch e Paterson 1985]. Assim, os algoritmos que compõem o PBFT se baseiam na hipótese de que o remetente de uma mensagem realiza retransmissões e que os atrasos de rede não crescem indefinidamente. Em sua primeira versão, proposta em 1999 [Castro e Liskov 1999], o PBFT faz uso de técnicas de criptografia assimétrica e funções de *hash* assumidamente resistentes à colisões. Em um aprimoramento posterior [Castro

e Liskov 2002], foi proposto um novo modelo e algoritmo, nos quais são utilizados *Message Authentication Codes* (MACs) para garantia da integridade e autenticidade de mensagens, eliminando a necessidade do uso de técnicas de criptografia assimétrica.

O conjunto de servidores possui dois tipos de réplicas: a primária, que é responsável por receber as requisições dos clientes e determinar a seqüência de execução destas nas réplicas restantes, e as *backups*, que são as réplicas restantes, que recebem a ordem da primária e a seguem, visando manter o serviço correto. Desse modo, os clientes submetem requisições apenas à réplica primária, a qual determina uma seqüência de execução para as requisições e a repassa para as réplicas *backup*, as quais entram em consenso sobre a seqüência proposta, executam as requisições e enviam as respostas aos clientes. Os clientes, por sua vez, aguardam por $f + 1$ respostas iguais para cada uma de suas requisições para considerá-las corretas.

Para evitar que uma réplica primária bizantina comprometa a corretude do sistema, o algoritmo utilizado é formado por uma sucessão de configurações, chamadas de visões [Babaoğlu, Davoli e Montresor 1997]. A cada visão, uma réplica é a primária se, e somente se, $p = v \ mod \ \|R\|$, onde p é a identidade da réplica em questão, v é o número da visão e R é o grupo composto por todas as réplicas. As visões são numeradas de forma consecutiva e uma mudança de visão ocorre (ou seja, v é incrementado) quando as réplicas *backup* suspeitam que a primária se comporta de forma incorreta. O protocolo de mudança de visão será descrito posteriormente.

Conforme mostra a Figura 3.1, o algoritmo para ordenação das requisições recebidas dos clientes é composto por três fases no lado dos servidores e duas fases no lado cliente. Inicialmente, na fase *REQUEST*, o cliente emite uma requisição para a réplica primária (identidade conhecida pelo cliente através da expressão $p = v \ mod \ \|R\|$). A partir desse momento, o cliente inicia um temporizador, utilizado para determinar quando uma retransmissão deverá ser realizada. Após recebida a mensagem, a primária inicia um protocolo de três fases para fazer a difusão atômica da mensagem bem como o número de seqüência associado a esta. Na fase *PRE-PREPARE*, a réplica primária atribui um número de seqüência para a mensagem recebida e a difunde, juntamente com o número de seqüência, para todas as réplicas *backup*. Tais réplicas aceitarão as mensagens dessa fase somente se essas mensagens estiverem devidamente assinadas pelo cliente, se a mensagem pertencer à visão v e se não tiver aceitado outra mensagem para a mesma visão com o mesmo número de seqüência.

Ao aceitar a mensagem de *PRE-PREPARE* recebida do primário,

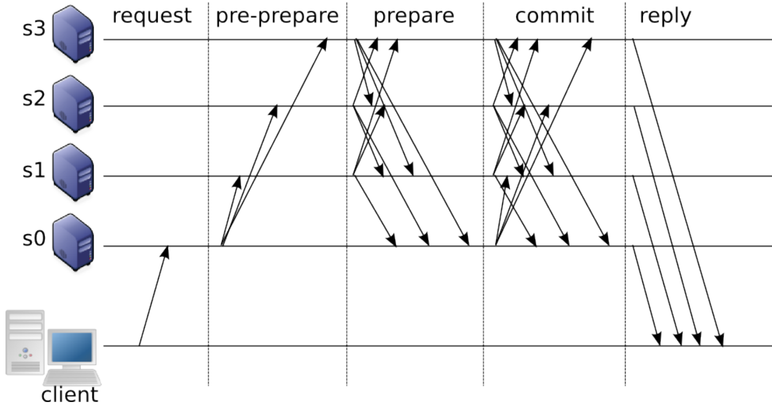


Figura 3.1: Diagrama de funcionamento do PBFT

a réplica entra na fase *PREPARE*. Nessa fase, cada réplica *backup* envia uma mensagem para todas as réplicas em R, que contém a sua identificação, número da visão atual, resumo criptográfico da mensagem e o número de seqüência proposto pela réplica primária. Nessa mesma fase, todas as réplicas recebem as mensagens enviadas pelas outras. Ao receber $2f$ mensagens iguais vindas de diferentes réplicas, uma réplica pode se considerar pronta para executar tal requisição naquela visão. A última fase, *COMMIT*, é iniciada por cada réplica após receber as $2f$ mensagens da fase anterior. Essa fase é necessária para garantir a ordenação das mensagens mesmo na ocorrência de uma mudança de visão. Ao receber $2f + 1$ mensagens corretas de *COMMIT*, as réplicas podem executar a mensagem enviada pelo cliente tão logo as mensagens anteriores (cujo número de seqüência é inferior ao número atribuído pelo primário para a mensagem atual) tenham sido processadas. Concluindo a execução, é iniciada a fase *REPLY*, na qual as réplicas enviam a requisição processada diretamente ao cliente, o qual, por sua vez, aguarda por $f + 1$ respostas com mesmo conteúdo de resposta e relativas à mesma requisição enviada (o que é verificado através de um campo *timestamp* que é inserido na requisição e o qual é anexado à mensagem de resposta pelas réplicas).

Se o temporizador referido anteriormente expirar antes de o cliente receber $f+1$ respostas corretas, o cliente então reenvia a requisição, porém, desta vez, para todas as réplicas. Tendo recebido a retransmissão, uma réplica qualquer apenas reenvia a resposta ao cliente, caso esta já tenha processado a mesma requisição anteriormente. Caso contrário,

a réplica repassa a mensagem ao primário e aguarda pela execução do protocolo de três fases para tal mensagem. Caso isso não ocorra, as réplicas *backup* acabarão por suspeitar da primária e iniciar uma mudança de visão, conforme será descrito na seqüência.

Para impedir que uma réplica primária corrompida impeça que o protocolo faça progresso, o PBFT utiliza um mecanismo de mudança de visão, no qual ocorre a troca da réplica primária. Ao suspeitar que a réplica primária está corrompida, cada réplica *backup* envia uma mensagem *VIEW-CHANGE* para todas as réplicas, contendo o identificador da próxima visão $(v + 1)$, número de seqüência n do último *checkpoint* estável, conjunto de $2f + 1$ mensagens válidas de *checkpoint* para provar a validade de n, um conjunto com todas as mensagens que, antes do início da mudança de visão, estavam com a fase *PREPARE* completa e que possuem um número de seqüência maior que n. Ao receber $2f$ mensagens *VIEW-CHANGE*, o novo primário, calculado através de $p = (v + 1) \ mod \ \|R\|$, emite uma mensagem *NEW-VIEW* para as réplicas restantes, composta pelo número da nova visão, o conjunto de mensagens de *VIEW-CHANGE* recebidas (para provar a veracidade da mudança de visão) e uma mensagem *PRE-PREPARE* para cada mensagem cujo processamento foi interrompido pela mudança de visão. Ao receber uma mensagem de *NEW-VIEW*, as réplicas realizam a troca de visão, desde que tal mensagem esteja correta.

Dada a robustez e complexidade do PBFT, a execução dos protocolos é um tanto quanto custosa. Para resolver esse problema, foram propostas diversas otimizações e melhorias para a versão inicial do protocolo:

- **Substituição de criptografia assimétrica por simétrica:** testes práticos realizados na época pelos autores do PBFT mostraram que operações de criptografia assimétrica apresentam alto custo computacional. Sabendo que o protocolo PBFT utiliza intensamente operações criptográficas, uma segunda versão do trabalho foi proposta [Castro e Liskov 2002]. Nessa nova versão, as assinaturas das mensagens foram trocadas por MACs (*Message Authentication Code*) e por *authenticators*. Esses últimos são nada mais que vetores de MACs, que possuem uma posição para cada réplica envolvida na comunicação.

- **Resumo criptográfico como resposta:** a largura de banda utilizada na versão padrão do protocolo é bastante grande, visto que para cada requisição submetida pelo cliente são necessárias, no caso ótimo, $f + 1$ mensagens contendo a resposta. Para serviços que necessitam

grandes quantidades de dados como resposta de suas operações (por exemplo, servidor de arquivos), a largura de banda utilizada é muito grande. Visando reverter tal consumo excessivo de banda, uma otimização proposta consiste em o cliente eleger uma das réplicas para lhe enviar o conteúdo inteiro da resposta e as réplicas restantes enviarem apenas os resumos. Assim, no melhor dos casos, apenas uma mensagem inteira será enviada e resumos criptográficos de tamanho fixo e reduzido serão enviados pelas réplicas restantes. Caso o cliente verifique que os resumos criptográficos não são correspondentes à mensagem inteira recebida, este solicita, através de uma mensagem, que todas as réplicas lhe enviem a mensagem completa.

- **Execução por tentativa:** essa otimização visa diminuir o número de passos de comunicação para execução de uma requisição de cinco para quatro. As fases *COMMIT* e *REPLY* são mescladas em uma só, na qual as réplicas realizam o envio da resposta ao cliente. As mensagens de *COMMIT* podem ser anexadas nas futuras mensagens de *PRE-PREPARE* e *PREPARE*. Caso ocorra uma troca de visão antes que uma mensagem seja efetivada pelas mensagens *COMMIT*, é necessário que tais mensagens (executadas por tentativas) sejam anuladas e que as réplicas revertam seu estado para o *checkpoint* estável contido na mensagem de *NEW-VIEW*.

- **Operações somente-leitura:** requisições de somente-leitura são executadas tão logo recebidas, visto que não afetam o estado do serviço. Para evitar que leituras sejam feitas sobre dados que ainda podem ser revertidos, é necessário que a réplica aguarde que todas as mensagens relacionadas ao conteúdo da leitura tenham sido efetivadas na fase *COMMIT*.

- **Execução em lote:** ao invés de executar todo o protocolo de três fases para cada mensagem recebida do cliente, as réplicas agrupam as requisições em grupos de b mensagens. Assim, uma mensagem recebida do cliente é inserida em um grupo de mensagens para que as operações necessárias para ordenação das mensagens sejam feitas sobre o conjunto todo, diminuindo bastante o número de trocas de mensagens envolvidas para execução de b mensagens.

- **Envio separado da requisição:** na abordagem padrão do protocolo, o primário recebe a requisição do cliente, anexa o conteúdo desta na mensagem de *PRE-PREPARE* e envia para as réplicas *backup*. Para mensagens grandes, essa abordagem representa desperdício de banda de rede, visto que uma mesma mensagem trafega

duas vezes pela rede. Assim, uma otimização foi proposta, segundo a qual, mensagens maiores que determinado tamanho são enviadas pelo cliente para todas as réplicas, as quais verificam a autenticidade destas e adicionam em seus *buffers*. Assim, na fase de *PRE-PREPARE*, a réplica primária adiciona apenas o resumo criptográfico da mensagem na mensagem *PRE-PREPARE*.

O PBFT foi o primeiro trabalho proposto a apresentar viabilidade prática, o que é comprovado através das avaliações de resultado apresentadas [Castro e Liskov 1999, Castro e Liskov 2002], onde a implementação do protocolo juntamente com as otimizações propostas obtém resultados com pequenas perdas de performance quando comparados à execução singular dos serviços. Apesar da quantidade de passos de comunicação entre réplicas necessários por requisição e da quantidade total de réplicas necessárias serem considerados elevados com relação aos trabalhos mais recentes, o PBFT é considerado o estado da arte na área de tolerância a faltas bizantinas através da abordagem replicação máquina de estados.

3.1.2 Zyzzyva - Speculative Byzantine Fault Tolerance

Zyzzyva [Kotla et al. 2007] é um estudo que apresenta um protocolo para tolerância a faltas bizantinas que possui como objetivo oferecer um protocolo BFT para execução de serviços replicados que diminua a complexidade existente nos protocolos previamente propostos, como o PBFT [Castro e Liskov 1999]. Diferentemente das propostas anteriores, o Zyzzyva tornou-se importante por apresentar uma abordagem na qual as réplicas de serviço executam as requisições recebidas dos clientes de forma especulativa, ou seja, uma requisição é processada e enviada ao cliente tão logo uma réplica primária determine a ordem para ela, sem a execução de um protocolo de acordo entre todas as réplicas. Assim, ao contrário dos trabalhos anteriores, nos quais as réplicas somente executam a requisição quando existe a certeza de que todas as réplicas corretas também a processarão na mesma seqüência, no Zyzzyva, a tarefa de garantir que as requisições sejam executadas de forma consistente fica a cargo dos clientes, os quais devem detectar possíveis inconsistências e informar as réplicas sobre isso. Para tanto, as réplicas anexam um histórico de suas execuções para que os clientes possam verificar se as réplicas possuem a mesma ordenação de requisições.

O Zyzzyva é executado por $3f + 1$ réplicas, sendo uma delas a primária. Uma execução do protocolo ocorre da seguinte forma: o cliente envia a requisição apenas para a réplica primária, a qual repassa a requisição para as outras réplicas, que imediatamente executam esta e enviam

a resposta para o cliente. Uma requisição pode ser completada de duas formas:

1. Cliente recebe $3f + 1$ respostas mutuamente consistentes: cliente considera a resposta como correta e opera sobre esta.

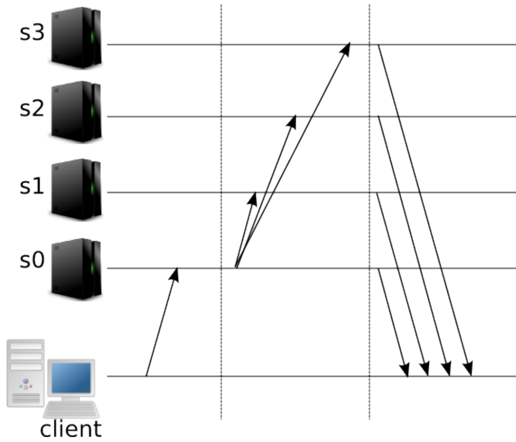
2. Cliente recebe entre $2f + 1$ e $3f$ respostas mutuamente consistentes: o cliente reúne $2f + 1$ respostas e as distribui como um certificado de *commit* para as réplicas. Então, ele aguarda por $2f + 1$ confirmações de recebimento do certificado e considera a requisição como completa, seguindo sua execução normal.

O Zyzzyva é composto por três sub-protocolos:

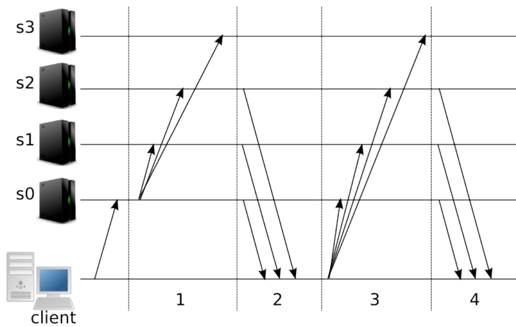
- ***Checkpointing***: limita o estado que deve ser armazenado pelas réplicas.
- **Acordo**: realiza a ordenação das requisições para execução das réplicas.
- **Mudança de Visão**: coordena a eleição de um novo primário quando o atual for considerado faltoso.

A cada *CP_INTERVAL* requisições, as réplicas constroem um *checkpoint* [Kotla et al. 2007]. Quando uma réplica correta gera uma tentativa de *checkpoint*, ela envia uma mensagem *CHECKPOINT* assinada para todas as réplicas, contendo o maior número de seqüência das requisições contidas no *checkpoint*, um resumo criptográfico de tal *checkpoint* e do estado da aplicação executada como serviço. Ao receber $f + 1$ mensagens de *CHECKPOINT* mutuamente consistentes e corretamente assinadas por diferentes réplicas (possivelmente incluindo a sua própria mensagem), uma réplica correta considera estáveis o *checkpoint* e o estado correspondente da aplicação. A geração dos *checkpoints* possui como principal utilidade prática o truncamento do histórico, de modo a diminuir o espaço em memória que o protocolo Zyzzyva necessita. Assim, ao tornar estável um *checkpoint*, uma réplica correta remove as entradas anteriores ao *checkpoint* do seu histórico. Além disso, uma réplica bloqueia o processamento de novas requisições após processar $2 \times CP_INTERVAL$ requisições desde o último *checkpoint* estável.

Conforme nos mostra a Figura 3.2, as réplicas que compõem o Zyzzyva executam as requisições tão logo as recebam do primário, sem realização de troca de mensagens entre réplicas para determinar a ordem para a requisição. Ao receber uma mensagem corretamente assinada do tipo *REQUEST* do cliente, a réplica primária transmite uma mensagem



(a) Execução correta do Zyzzyva



(b) Execução do Zyzzyva com uma réplica faltosa

Figura 3.2: Execução do protocolo de acordo do Zyzzyva

assinada do tipo *ORDER-REQ* para as demais réplicas, contendo um número de seqüência n dentro da visão v, o resumo da mensagem do cliente e o resumo do histórico. Ao receber uma mensagem *ORDER-REQ* da réplica primária, uma réplica *backup* i verifica se os dados contidos na mensagem são corretos, ou seja, se a mensagem vinda do cliente segue as especificações para mensagens, se o resumo enviado corresponde à mensagem, se o resumo do histórico enviado pelo primário é condizente com o histórico de i, e se $n = max_n + 1$, sendo max_n o maior número de seqüência no histórico de i. Caso todas as condições acima sejam satisfeitas, i aceita a mensagem, acrescenta esta ao seu histórico, executa

a mensagem do cliente e envia para este uma mensagem do tipo *SPEC-RESPONSE*, contendo a resposta especulativa para sua requisição, bem como o resumo do seu histórico e a mensagem *ORDER-REQ* recebida do primário.

Assim, o cliente c que emitiu a requisição para as réplicas recebe as respostas especulativas destas. Caso receba $3f + 1$ mensagens mutuamente consistentes vindas de réplicas diferentes e corretamente assinadas, c considera a requisição completa e segue sua execução normalmente, como mostra a Figura 3.2(a). Caso o cliente c receba entre $2f + 1$ e $3f$ mensagens mutuamente consistentes, c envia uma mensagem *COMMIT* para todas as réplicas contendo uma lista de $2f + 1$ réplicas, as mensagens *SPEC-RESPONSE* recebidas das $2f + 1$ réplicas e suas assinaturas. Ao receber a mensagem *COMMIT* do cliente, cada réplica verifica se seu estado local está consistente com o estado das outras réplicas (verificando os dados contidos na mensagem *COMMIT*), atualiza seu estado relacionado ao maior certificado de *commit* para o certificado da mensagem recebida (caso o maior número de seqüência do novo certificado seja maior que o anterior) e envia uma mensagem de *LOCAL-COMMIT* ao cliente. O cliente, por sua vez, ao receber $2f + 1$ mensagens de *LOCAL-COMMIT*, considera a mensagem completa e segue a sua execução.

Caso não receba $2f + 1$ mensagens mutuamente consistentes para uma requisição, c retransmite a mensagem para todas as réplicas. As réplicas *backup*, ao receber retransmissão, enviam uma mensagem *CONFIRM-REQ* para a réplica primária e iniciam um temporizador. Caso o temporizador expire antes de a réplica receber uma mensagem *ORDER-REQ* do primário contendo uma proposta de ordem para a mensagem recebida do cliente, a réplica *backup* inicia uma mudança de visão, conforme será descrito a seguir.

Caso o cliente c receba mensagens de *SPEC-REPONSE* que contenham uma mensagem válida de *ORDER-REQ* para a mesma requisição, com mesmo número de visão, mas com número de seqüência ou histórico diferentes, c envia uma prova de mau-comportamento da réplica primária para todas as réplicas, as quais iniciam uma mudança de visão.

O protocolo de mudança de visão é disparado quando uma réplica suspeita de que a réplica primária é faltosa, visando realizar a troca da réplica primária. Quando isso ocorre, a réplica envia uma mensagem do tipo *I-HATE-THE-PRIMARY* para todas as réplicas. Ao receber $f + 1$ mensagens do mesmo tipo, a réplica envia uma mensagem do tipo *VIEW-CHANGE* para todas as réplicas, contendo uma prova de que $f + 1$ réplicas desconfiam da corretude da réplica primária. Ao receber uma mensagem válida de *VIEW-CHANGE*, uma réplica correta efetiva a mudança de vi-

são.

Assim como no PBFT [Castro e Liskov 1999], algumas otimizações foram propostas e implementadas para o Zyzzyva:

- Substituição de assinatura por MACs.
- Separação do Acordo e da Execução [Yin et al. 2003].
- Requisições em lote.
- Otimização em requisições somente-leitura.
- Apenas uma réplica envia resposta completa, outras enviam resumo.

O Zyzzyva é um estudo bastante relevante, pois apresenta uma técnica para tolerância a faltas bizantinas baseada na idéia de Replicação Máquina de Estados associada a recente idéia de execução especulativa [Nightingale, Chen e Flinn 2005]. Através dessa idéia, é possível que uma requisição seja efetivada por parte do cliente após apenas uma fase de comunicação entre as réplicas, sendo este o melhor dos casos, onde todas as réplicas se comportam corretamente. A não-necessidade de custosas fases de comunicação entre réplicas para obtenção do acordo impacta bastante nos resultados de latência e *throughput* [Kotla et al. 2007]. Além disso, esse trabalho inova no sentido de repassar parte da responsabilidade pelo acordo na execução de requisições para os clientes, que, a cada mensagem recebida, verificam a consistência das execuções realizadas pelas réplicas. Os resultados obtidos nas avaliações práticas mostram que a proposta apresenta uma melhoria grande na performance quando comparado aos trabalhos anteriores, o que aumenta a sua viabilidade prática.

3.2 Modelos Híbridos

Tradicionalmente, os modelos de falhas de sistemas distribuídos são baseados na hipótese de que todos os componentes do sistema são sujeitos aos mesmos tipos de falhas. Em [Correia et al. 2002] foi apresentada uma abordagem inovadora, a qual apresenta um modelo híbrido de falhas, onde cada membro de um sistema distribuído possui um componente local seguro, sujeito apenas a falhas de *crash*, interconectado aos componentes locais de outros membros. Através desse componente são realizadas operações para o estabelecimento de acordo e consenso nesse sistema. Após isso, surgiram outros trabalhos a se utilizar da noção de modelo híbrido de falhas, através do relacionamento da tolerância a faltas bizantinas com o conceito e tecnologias de virtualização, onde se considera a existência de um componente confiável para dar suporte a execução das réplicas de serviço, as quais estão sujeitas a sofrer falhas arbitrárias.

A seguir, serão descritos os principais trabalhos relacionados a modelos híbridos de falhas, como o TTCB [Correia et al. 2002], primeiro trabalho a propor tal abordagem, o VM-FIT [Reiser e Kapitza 2007], que é o primeiro trabalho a aplicar as vantagens do uso de virtualização para proposição de protocolos BFT mais simples, bem como o LBFT (*Lightweight Byzantine Fault Tolerance* [Chun, Maniatis e Shenker 2008]), que realiza uma importante e imprescindível discussão sobre os aspectos que devem ser levados em consideração quando da implementação de protocolos BFT sobre ambientes virtualizados.

3.2.1 *Trusted Timely Computing Base*

Protocolos para difusão confiável de mensagens são, tradicionalmente, baseados em premissas realizadas sobre o ambiente no qual estes serão executados. Tais premissas vão desde os tipos de faltas toleradas nos componentes do sistema até as propriedades de sincronia na comunicação e no comportamento dos processos. O *Trusted Timely Computing Base* (TTCB) [Correia et al. 2002] apresenta uma abordagem inovadora, pois seu modelo de sistema apresenta um *modelo híbrido de falhas*, que consiste em realizar diferentes hipóteses sobre as falhas às quais diferentes componentes estão sujeitos. Assim, é possível circunscrever algumas impossibilidades existentes em sistemas que apresentam um modelo de falhas homogêneo entre seus componentes, como por exemplo a condição FLP [Fischer, Lynch e Paterson 1985] e a necessidade de $3f+1$ membros para tolerar faltas bizantinas [Lamport, Shostak e Pease 1982].

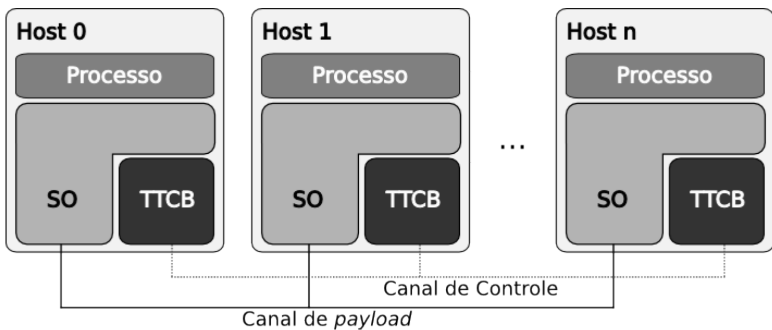


Figura 3.3: Arquitetura da TTCB

A proposta apresentada em [Correia et al. 2002] é baseada na existência de um núcleo distribuído que está sujeito apenas a falhas de *crash*, o

TTCB. Esse componente é apresentado na Figura 3.3. O TTCB é formado por componentes locais aos *hosts* que compõem o sistema distribuído. Os TTCBs locais são interconectados por um canal de comunicação privado e apresentam comportamento síncrono, dessa forma é possível determinar quando um *host* sofre uma falha através de um detector. O canal de *payload* interconecta os *hosts* e é um canal de comunicação assíncrono. A camada de serviço que executa sobre o *host* possuidor de um TTCB local pode estar sujeito a qualquer tipo de faltas, inclusive bizantinas. Assim, é composto um modelo híbrido, onde o componente que auxilia na obtenção de consenso entre os membros está sujeito apenas a falhas de *crash*, enquanto que a aplicação está sujeita a falhas de origem arbitrária. A idéia fundamental por trás da utilização de um modelo híbrido de falhas é a possibilidade de executar as partes cruciais de um algoritmo distribuído em um componente confiável. Assim, essa abordagem permite uma simplificação do protocolo de consenso, através da utilização da TTCB. O TTCB oferece dois serviços básicos:

- **Serviço de Autenticação Local**: fornece mecanismos para que os processos possam se comunicar de forma segura com o TTCB. Para isso, oferece chaves criptográficas para serem utilizadas na comunicação entre o processo e o TTCB.
- **Serviço de Acordo de Bloco Confiável**: oferece um serviço de acordo que entrega o valor obtido como acordo entre os valores propostos pelos processos que utilizam a TTCB. Esse serviço possui três primitivas: *TTCB_propose* (utilizada por um processo para propor um valor), *TTCB_decide* (utilizada para um processo para obter o valor decidido) e *decision* (uma função que define como o valor de acordo será calculado).

O protocolo de difusão confiável proposto no trabalho é composto por duas fases. Na primeira fase, o *remetente* da mensagem envia tal mensagem para todos os membros do grupo (tanto o próprio *remetente* quanto os membros restantes, os *destinatários*) e, através do TTCB, todos os membros transmitem, de forma segura, um *hash* da mensagem, visando garantir a autenticidade e a integridade desta. Então, através do serviço de acordo oferecido pelo TTCB, os processos aguardam pelo valor de decisão. Se o serviço de acordo indicar que todos os processos propuseram o mesmo valor, então estes podem terminar a execução. Caso um ou mais processos não propuserem o *hash* correto da mensagem até o momento em que o serviço de acordo foi executado, a segunda fase é iniciada. Na segunda fase, os processos portadores da mensagem retransmitem esta para os processos que não confirmaram seu recebimento até

que sejam recebidas essas confirmações de recebimento ou até que número de transmissões ultrapasse o grau de omissão definido.

Assim, a utilização de um modelo híbrido de falhas, através da criação de um componente que está sujeito apenas a falhas de *crash*, é possível implementar um protocolo de difusão confiável e até mesmo um protocolo de consenso [Correia et al. 2005].

3.2.2 VM-FIT

O VM-FIT (*Virtual Machine - Fault and Intrusion Tolerance*) foi um dos primeiros trabalhos a utilizar técnicas de virtualização para prover tolerância a intrusões do qual se tem notícia. Nesse caso, utilizar técnicas de virtualização significa tomar proveito de características arquiteturais específicas de sistemas virtualizados, de modo a facilitar a construção de um sistema tolerante a intrusões.

Em sua primeira versão, chamada de RESH [Reiser et al. 2006] (*Redundant Execution on Single Host*), o trabalho apresentou uma abordagem baseada em uma única máquina física oferecendo um serviço replicado através de máquinas virtuais. O uso de máquinas virtuais replicando um serviço não é o grande diferencial deste trabalho, visto que estudos anteriores, como o seminal PBFT [Castro e Liskov 1999], Zyzzyva [Kotla et al. 2007] e muitos outros, podem facilmente ser implantados em ambientes virtualizados, utilizando várias máquinas virtuais executando o protocolo. O ponto principal e diferencial desse trabalho é a utilização de uma máquina virtual minimalista realizando a interceptação de mensagens enviadas por clientes e definindo a seqüência na qual as mensagens serão executadas, sendo assim o responsável pelo consenso sobre a ordem de execução das requisições.

A proposta desse trabalho é baseada na arquitetura do VMM Xen [Barham et al. 2003]. Assim, a máquina virtual que define a seqüência das requisições é chamada de *Domínio NV* (*Network/Voting*). Tal domínio apenas provê *drivers* de rede, pilha de protocolos de rede e o componente de votação. Em se tratando de um domínio minimalista, provas de correção do código podem ser obtidas mais facilmente, podendo-se assim tratar tal sistema como confiável. Além do *Domínio NV*, o trabalho possui o *Domínio 0*, que contém os *drivers* de dispositivos reais, com os quais os sistemas convidados irão interagir. Além desses dois domínios, existe o VMM Xen em si, considerado uma entidade confiável. As réplicas de serviço são executadas como sistemas convidados nessa arquitetura e podem apresentar comportamento bizantino, sendo necessárias $2f + 1$ réplicas para tolerar f bizantinas.

Conforme pode ser visualizado na Figura 3.4, a replicação do ser-

viço é transparente para o cliente, que envia requisições apenas para a máquina hospedeira, como se fosse esta a responsável pela execução do serviço. Porém, o domínio NV intercepta as mensagens enviadas pelos clientes e as repassa para os sistemas convidados atribuindo uma seqüência para a execução destas. Após o recebimento de uma mensagem, os sistemas convidados executam a requisição e enviam a resposta ao domínio NV, o qual realiza uma votação para determinar a maioria de respostas iguais e retorna o valor obtido na maioria para o cliente.

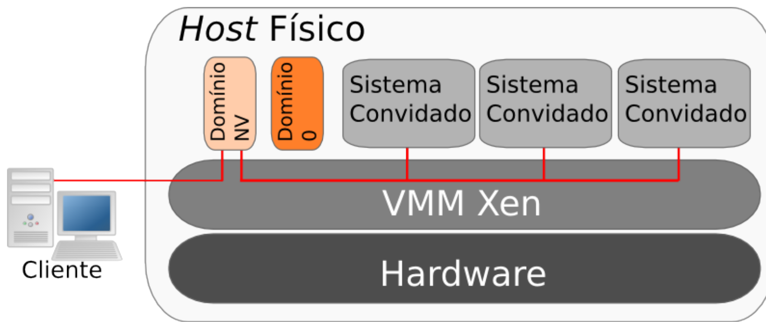


Figura 3.4: Arquitetura RESH

Porém, a abordagem RESH possui a limitação de não tolerar faltas de *crash* no sistema hospedeiro. Para contornar tal situação, foi proposta uma extensão da arquitetura RESH, chamada de REMH (*Redundant Execution on Multiple Hosts*), a qual realiza a replicação do sistema hospedeiro de modo a tolerar faltas de *crash*. Nessa abordagem, as réplicas de serviço são divididas em $2f + 1$ máquinas virtuais, cada uma executando sobre uma máquina física, conforme pode ser visto na Figura 3.5.

Como mostra a Figura 3.5, as requisições emitidas por clientes são interceptadas transparentemente por uma entidade confiável (Domínio NV) em alguma das máquinas físicas. Tal entidade, ao interceptar uma mensagem do cliente, utiliza um sistema de comunicação de grupo para difundir a mensagem interceptada para as réplicas de serviço (residentes em outras máquinas físicas) restantes. Após receber tal mensagem e processá-la, as réplicas de serviço retornam a resposta para o Domínio NV da máquina física onde foi interceptada a mensagem. Então, tal domínio realiza a votação para obter a maioria $(f + 1)$ de mensagens com conteúdo idêntico, definindo e enviando ao cliente a resposta correta para a requisição emitida.

Como referido anteriormente, RESH não é capaz de tolerar uma

falta de crash no hospedeiro físico, pois é implementado em uma única máquina física. A abordagem REMH, por sua vez, resolve tal fragilidade, realizando a replicação física do sistema hospedeiro. Assim, REMH é capaz de tolerar até f *crashes* em um total de $2f + 1$ sistemas físicos.

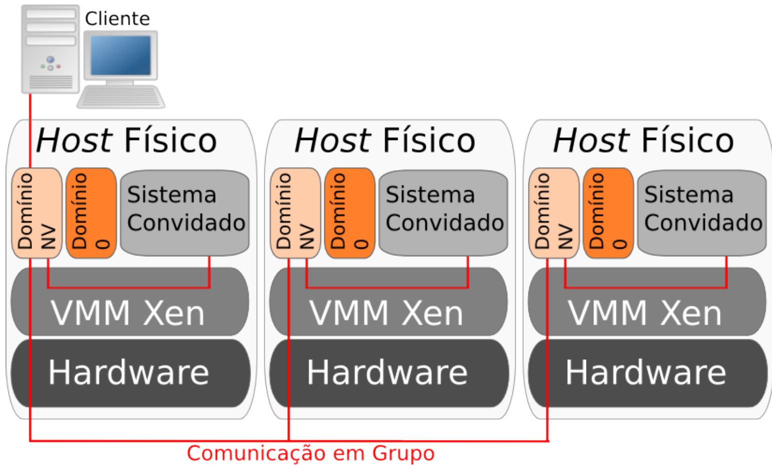


Figura 3.5: Arquitetura REMH

Assim sendo, a arquitetura VM-FIT apresenta duas abordagens, ambas necessitando $2f + 1$ réplicas de serviço para tolerar f réplicas bizantinas. A abordagem RESH necessita de apenas uma máquina física executando um sistema de máquinas virtuais composto por $2f + 1$ réplicas de serviço, o VMM, e um domínio específico que implementa a camada de rede e de votação. Já a abordagem REMH, necessita de $2f + 1$ máquinas físicas, cada uma delas executando a réplica de sistema sobre uma máquina virtual e, assim como em RESH, o domínio de rede e votação (NV), sendo que todos os domínios NV são conectados entre si e implementam também um sistema de comunicação em grupo de modo a repassar as mensagens interceptadas para as máquinas físicas restantes.

3.2.2.1 VM-FIT e Recuperação Pró-ativa

Sistemas BFT possuem na sua descrição do modelo de sistema, a restrição de tolerar no máximo f réplicas faltosas de um total que varia ($3f+1$, $2f+1$, etc). Porém, garantir que apenas um número limitado f de réplicas apresentem comportamento bizantino durante todo o período de vida do sistema é uma tarefa complicada. Para isso, trabalhos anteriores

[Castro e Liskov 2002] propuseram uma abordagem que utiliza recuperação pró-ativa para rejuvenescer os sistemas. Dessa forma, recuperações são disparadas periodicamente de modo que sistemas corrompidos por atacantes sejam recuperados e iniciados a partir de uma base de código segura. Um dos grandes desafios para sistemas que utilizam recuperação pró-ativa é diminuir o tempo necessário para a recuperação de um sistema, visto que durante tal período de tempo, o sistema terá sua resistência possivelmente diminuída, pois enquanto uma réplica estiver sendo recuperada, o número total de réplicas do sistema fica diminuído.

Também pertencente a arquitetura VM-FIT, foi proposta uma abordagem de recuperação pró-ativa baseada em virtualização [Reiser e Kapitza 2007]. Para diminuir o período de tempo em que uma réplica permanece indisponível devido à sua recuperação, nessa abordagem, a tarefa de recuperação dos sistemas fica a cargo do VMM (*hypervisor*). Assim, é possível que o VMM inicie um novo sistema convidado enquanto que o sistema convidado atual permanece sendo executado. Para sistemas *stateless*, a transição do sistema anterior para o sistema recém iniciado é praticamente instantânea. Já em sistemas *statefull*, é necessário que ocorra a transferência de estado atual das réplicas em execução para as réplicas a serem iniciadas. Para garantir que o estado a ser repassado para as novas réplicas seja consistente, são necessárias $f + 1$ mensagens de *CHECKPOINT* iguais, que representam o estado a ser transferido. De modo a diminuir o tempo que uma réplica permanece indisponível devido à recuperação, a transferência de estado pode ser feita através de um simples remapeamento de blocos de memória, caso a réplica recuperada apresentasse comportamento correto.

O VM-FIT, estudo composto pelas abordagens RESH e REMH, é um estudo de grande importância na área de tolerância a intrusões, devido ao fato de ser um dos primeiros trabalhos a tomar proveito de características específicas de sistemas virtualizados visando oferecer uma arquitetura que possibilite a simplificação do desenvolvimento de um protocolo de consenso. Através de sua arquitetura, foi possível a proposta de um protocolo de consenso simples, baseado em um coordenador confiável, fato que se tornou prático devido à possibilidade de utilização de uma camada mínima de software para tal tarefa.

3.2.3 LBFT

Diferentemente dos trabalhos referidos anteriormente neste capítulo, o trabalho a ser descrito nesta seção não apresenta implementação e resultados [Chun, Maniatis e Shenker 2008]. Focando principalmente em uma discussão teórica sobre as possibilidades e desafios que emergem

quando se trata do uso de virtualização para protocolos BFT, o estudo faz um apanhado sobre formas de se tomar proveito da virtualização para que se ofereça um ambiente de computação tolerante a faltas bizantinas.

Os autores tratam apenas de sistemas replicados sobre máquinas virtuais executando sobre uma mesma máquina física. Assim sendo, um dos desafios considerados mais importantes é a garantia da independência de faltas entre os sistemas virtuais, ou seja, que esses sistemas não sofram as mesmas falhas, causadas pelas mesmas faltas. Para tanto, é necessário o isolamento entre as máquinas virtuais e diversidade de *software* entre as réplicas. O isolamento pode ser garantido por VMMs que utilizam as recentes tecnologias oferecidos por processadores Intel (*Trusted Execution Platform* [Intel 2008]) e AMD (*Secure Virtual Machine* [Devices 2007]). Para prover a diversidade, os autores propõem que cada sistema operacional a ser executado sobre uma máquina virtual, bem como a camada de replicação e aplicação, sejam diversificados automática e sistematicamente. Uma das formas propostas para realização dessa tarefa é a utilização da randomização binária e de sistema operacional [Chew e Song 2002, Xu, Kalbarczyk e Iyer 2003].

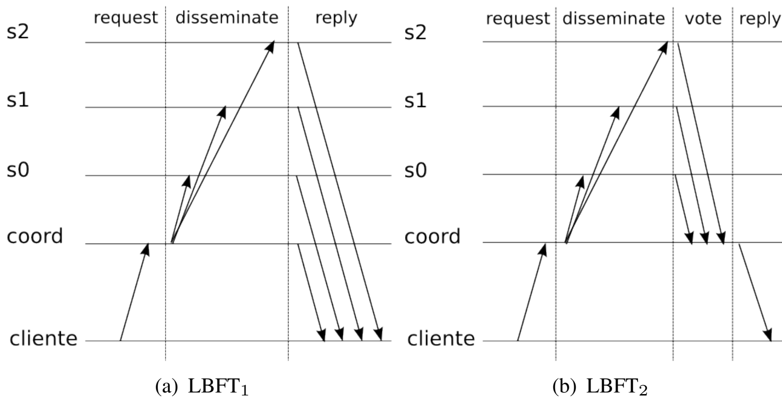
Com relação às oportunidades para criação de protocolos tolerantes a faltas bizantinas, os autores decompuseram o problema em três categorias: as características de confiabilidade do processo coordenador, as premissas de sincronia para comunicação entre réplicas e o nível de transparência da replicação para os clientes.

A primeira discussão foca nos aspectos de manter o processo coordenador (responsável por determinar a ordenação de processamento das requisições) dentro da base de computação confiável (*Trusted Computing Base* - TCB) ou não. De acordo com os autores, o sistema coordenador poderia ser suficientemente simples, visto que é um sistema específico para determinar a ordenação das mensagens e realizar a votação sobre estas. Tal simplicidade justificaria sua inclusão na TCB, visto que um coordenador confiável possibilita a criação de protocolos muito mais simples.

O segundo aspecto discutido pelos autores é a sincronia. Caso se consiga implementar um VMM que trate as comunicações entre as VMs como síncronas, os protocolos de replicação se tornam mais simples, visto que a ambigüidade entre um processo lento ou um processo maliciosamente silencioso deixaria de existir [Chun, Maniatis e Shenker 2008].

O último aspecto discutido é relacionado à transparência de replicação. Um protocolo BFT com replicação transparente ao cliente necessita que o processo coordenador, além de definir a ordenação das requisições, realize também a votação sobre as mensagens de resposta geradas

pelas réplicas de serviço para que retorne apenas a mensagem correta para o cliente. Já em um ambiente sem transparência, as réplicas de serviço enviam a resposta diretamente ao cliente, de modo que este realize a votação para obter a mensagem correta. Enquanto que a transparência de replicação reduz o consumo de largura de banda, traz consigo a desvantagem de agregar mais funcionalidades ao coordenador, tornando seu código maior e mais complexo. O aumento no tamanho e na complexidade do código acaba por tornar o software mais propenso a apresentar falhas [Nagappan, Ball e Zeller 2006].



(a) LBFT_1 (b) LBFT_2

Figura 3.6: Diagramas de execução dos protocolos LBFT_1 e LBFT_2

Dadas essas discussões, os autores do trabalho propuseram duas abordagens para o desenvolvimento de protocolos BFT, através de combinações das características descritas acima. A Figura 3.6(a) nos mostra o diagrama de execução da primeira abordagem, chamada de LBFT_1 e a Figura 3.6(b) mostra o diagrama de execução da segunda abordagem, chamada de LBFT_2.

LBFT_1 Essa proposta assume um coordenador confiável, assincronia e replicação explícita ao cliente. São necessárias $2f + 1$ réplicas de serviço somadas ao coordenador confiável, para prover *safety* e *liveness* em um ambiente com até f réplicas faltosas. O cliente c envia uma mensagem do tipo *REQUEST* apenas ao coordenador p, que, por sua vez, difunde uma mensagem do tipo *DIST* contendo a requisição do cliente e o número de seqüência para todas as réplicas de serviço. Cada réplica executa a requisição na ordem definida e envia a resposta diretamente para o cliente

c. Quando c recebe $f+1$ mensagens mutuamente consistentes, ele aceita a resposta. As operações criptográficas são realizadas apenas entre réplicas de serviço e clientes, visto que ambos têm conhecimento da existência de ambos.

LBFT$_2$ Nessa variação, a principal diferença é a replicação transparente ao cliente. Para isso, é necessário que o processo coordenador realize a verificação da assinatura da mensagem enviada pelo cliente (visto que cliente e réplicas de serviço não se conhecem) antes de difundi-la para as réplicas de serviço, as quais executam as requisições enviadas pelo coordenador sem verificar a assinatura do cliente e enviam a resposta da requisição para o coordenador, o qual realiza a votação sobre as mensagens e retorna apenas a mensagem correta ao cliente.

O intuito dos autores com a proposição das abordagens acima descritas foi apenas mostrar as possibilidades para criação de protocolos BFT tomando proveito de virtualização. Tais propostas não foram implementadas, mas sim discutidas, de forma a cobrir os aspectos mais relevantes para a proposição de arquiteturas BFT baseadas em virtualização. O artigo *Diverse Replication for Single-Machine Byzantine-Fault Tolerance* [Chun, Maniatis e Shenker 2008] realiza um importante estudo, fazendo um apanhado geral de questões que devem ser levadas em consideração quando da proposição/implementação de sistemas BFT baseados em máquinas virtuais executando sobre uma única máquina física.

3.3 Conclusões do Capítulo

Este capítulo apresentou os principais trabalhos na área de tolerância a faltas bizantinas utilizando a abordagem Replicação Máquina de Estados, os quais, de alguma forma se relacionam ao presente estudo. O primeiro trabalho apresentado, o PBFT [Castro e Liskov 1999], foi incluído por sua fundamental importância na área de tolerância a faltas bizantinas, por apresentar a primeira abordagem com viabilidade prática, comprovada através de resultados. A partir desses trabalhos, vários outros estudos surgiram [Yin et al. 2003, Kotla et al. 2007, Chun et al. 2007, Luiz et al. 2007, Veronese et al. 2008] apresentando soluções práticas para arquiteturas BFT baseadas na abordagem RME, buscando diminuir o custo e propor soluções alternativas às idéias apresentadas no PBFT, que é um protocolo bastante complexo.

O segundo trabalho descrito, o Zyzzyva [Kotla et al. 2007], foi considerado por apresentar uma abordagem nova para BFT, atingindo resultados de latência e *throughput* ainda melhores do que os obtidos no PBFT. Tais resultados são obtidos pois o Zyzzyva utiliza uma idéia de necessidade de comunicação entre réplicas para obter consenso sobre a re-

quisição a ser executada apenas quando se detecta alguma inconsistência no sistema, sendo assim um protocolo bastante otimizado para a execução do melhor caso, ou seja, quando as réplicas apresentam comportamento correto.

O TTCB [Correia et al. 2002] é um trabalho de grande importância com relação a este por introduzir a idéia de utilização de modelos híbridos para a especificação e implementação de sistemas tolerantes a faltas. Através de sua idéia de modelo híbrido de falhas, é possível assumir que determinados componentes são sujeitos a determinados tipos de falhas, enquanto que outros componentes são sujeitos apenas a outro conjunto de falhas. Tal abordagem foi utilizada neste trabalho para a especificação de uma extensão distribuída de nossa proposta.

O VM-FIT [Reiser et al. 2006, Reiser e Kapitza 2007, Reiser e Kapitza 2007] é apresentado devido ao fato de ser o primeiro trabalho conhecido a tomar proveito de facilidades providas pelo uso de virtualização para a simplificação de protocolos BFT. Assim como o presente estudo, o VM-FIT busca a diminuição da complexidade dos protocolos que compõem a arquitetura, de modo a transferir parte dessa complexidade para a implementação da arquitetura. Assim, através de facilidades presentes apenas em ambientes virtualizados, como gerenciamento centralizado de tarefas através de componentes minimalistas implementados no VMM, possibilidade de remapeamento de blocos para rápida transferência de estado, dentre outras, o VM-FIT se destaca por ser uma arquitetura totalmente focada em ambientes virtualizados.

Para finalizar, foi apresentado o trabalho *Diverse Replication for Single-Machine Byzantine Fault Tolerance* [Chun, Maniatis e Shenker 2008], que, apesar de não apresentar implementação e resultados práticos, realiza um abrangente estudo sobre aspectos relevantes na construção de propostas para BFT que utilizem virtualização. Além disso, o trabalho apresenta sugestões de características da virtualização a serem exploradas em trabalhos da área.

Com relação aos trabalhos relacionados que utilizam ambientes virtualizados para a execução de serviços tolerantes a faltas bizantinas/intrusões (VM-FIT [Reiser e Kapitza 2007] e LBFT [Chun, Maniatis e Shenker 2008]), foi realizado um levantamento dos principais atributos desses trabalhos, de modo a possibilitar uma comparação destes com o nosso trabalho. Foram considerados os seguintes aspectos em nossa comparação, a qual será retomada no Capítulo 4:

- **Passos de comunicação**: a quantidade de passos de comunicação necessária entre os servidores para efetivar uma requisição do cliente.

- **Total de VMs**: a quantidade mínima de máquinas virtuais necessárias para a execução da proposta.
- **Papel do componente confiável utilizado**: se o componente exerce papel ativo ou passivo nos protocolos de execução de operações.
- **Coordenador confiável**: se a proposta necessita da presença de um coordenador confiável (entidade ativa) nos protocolos de execução de operações.
- **Independência de VMM**: se a proposta apresentada é genérica ou se é voltada a um VMM especificamente.
- **Interoperabilidade de SO**: se existe a possibilidade de utilização de diferentes SOs tanto nos sistemas convidados (réplicas de serviço) quanto no sistema anfitrião.
- **Tolerante a *Crash***: se o estudo apresenta uma solução para tolerar faltas de *crash* no sistema anfitrião.

Tabela 3.1: Comparação entre os trabalhos relacionados

Característica	**VM-FIT**	**LBFT**
Passos de comunicação	2	2
Total de VMs	$2f+2$	$2f+2$
Componente	Ativo	Ativo
Coordenador confiável	Sim	Sim
Independência de VMM	Não	Sim
Interoperabilidade de SO	Não	Sim
Tolerante a *Crash*	Sim	Não

A Tabela 3.1 apresenta um resumo dos aspectos considerados para os trabalhos relacionados. Com relação ao número de passos de comunicação necessários, ambos os trabalhos necessitam de 2 passos para efetivar uma operação, visto que em ambos o coordenador confiável do algoritmo determina a ordem e repassa a mensagem para as outras réplicas. Em ambos os algoritmos são necessários $2f + 2$ entidades envolvidas no processo de execução, visto que o componente confiável em ambos exerce papel ativo. Com relação a independência de VMM, o VM-FIT é uma proposta específica para o VMM Xen, baseando sua proposta na arquitetura desse VMM. O LBFT, por sua vez, é uma proposta apenas

teórica, por isso pode ser considerado independente de VMM. O mesmo vale para a interoperabilidade de SO no LBFT. Já no VM-FIT, a interoperabilidade fica restrita aos ambientes suportados pelo VMM Xen. Através de sua versão distribuída, chamada de REMH, o VM-FIT se torna tolerante a faltas de *crash* ocorridas no sistema anfitrião, o que não ocorre com o LBFT, que apresenta um estudo baseado em uma máquina física apenas.

Capítulo 4

SMIT - Uma abordagem para TI usando Memória Compartilhada

O presente capítulo aborda especificamente nossa proposta de trabalho, descrevendo detalhes desde a motivação até os aspectos mais específicos da proposta. Nossa proposta foi batizada de SMIT, acrônimo para o título, em língua inglesa, *Shared Memory Intrusion Tolerance*, ou Tolerância a Intrusões Baseada em Memória Compartilhada, em língua portuguesa. Anteriormente chamado de VMBFT [Stumm Júnior et al. 2009] (*Virtual Machine based Byzantine Fault Tolerance*), nosso trabalho apresenta a especificação de uma arquitetura para replicação de serviços através de máquinas virtuais que se comunicam unicamente através de uma área de memória compartilhada de forma a prover uma arquitetura tolerante a faltas bizantinas baseada no modelo Replicação Máquina de Estados [Schneider 1990].

Assim sendo, em nosso trabalho, especificamos uma arquitetura e propusemos um conjunto de protocolos que, juntos, oferecem um ambiente para execução de serviços BFT. Mais especificamente, nossa proposta oferece mecanismos para que réplicas de serviço atinjam o consenso sobre a ordem de execução de requisições emitidas por clientes, de modo que todas as réplicas executem as requisições dos clientes em ordem total, conforme previsto para corretude do modelo Replicação Máquina de Estados [Schneider 1990]. Assim como em trabalhos anteriores [Reiser et al. 2006, Reiser e Kapitza 2007, Reiser e Kapitza 2007, Chun, Maniatis e Shenker 2008], nossa proposta toma proveito de características específicas e exclusivas de ambientes virtualizados de modo a simplificar o conjunto de protocolos que garante que a execução de requisições emitidas por clientes sejam executadas corretamente, assegurando o cumprimento das propriedades de Acordo e Ordem Total.

A Seção 4.1 apresenta uma descrição geral sobre o trabalho aqui apresentado. Em seguida, na Seção 4.2, é apresentado o modelo de sistema adotado em nosso trabalho, de acordo com os aspectos descritos na Seção 2.2, detalhando as hipóteses realizadas sobre os componentes de SMIT, de forma a assegurar que este ofereça uma execução correta do serviço a ser oferecido. Dada a sua importância para nosso trabalho, a Seção 4.3 apresenta os detalhes relativos à *Postbox*, que é o componente de memória compartilhada pelo qual as réplicas de serviço se comunicam. A Seção 4.4 apresenta as propriedades básicas que SMIT deve assegurar.

O protocolo de consenso para execução sobre SMIT é apresentado na Seção 4.5 e, na seção seguinte (Seção 4.6) são apresentadas as provas de correção de SMIT. Após isso, é apresentada uma abordagem distribuída de SMIT, denominada DSMIT (*Distributed* SMIT), a qual busca circunscrever a maior fraqueza de SMIT, que é a inabilidade de tolerar faltas de *crash* no sistema físico.

4.1 Motivação e Descrição Geral

Nossa proposta possui como objetivo principal a especificação de uma arquitetura para tolerância a intrusões, baseada em virtualização e que tome proveito de recursos oferecidos por sistemas virtualizados para possibilitar a criação de protocolos mais simples para execução de serviços replicados. A simplicidade é um fator muito importante, pois protocolos complexos possuem projetos e implementações complexas, que podem levar mais facilmente à ocorrência de faltas.

Agregada à simplificação dos protocolos, a diminuição do número de passos de comunicação para efetivação de uma requisição também é alvo de nossos esforços. Por exemplo, o protocolo de difusão atômica do PBFT [Castro e Liskov 1999] é composto por três custosas fases de comunicação entre réplicas, sendo que em duas destas fases existe difusão de mensagens de todos para todos. Tal custo é mais impactante se levarmos em consideração as operações criptográficas que são necessárias para verificar a integridade e a autenticidade de cada mensagem.

Além dos objetivos acima referidos, buscamos propor uma arquitetura com diminuição nos custos de replicação, que são bastante altos em propostas anteriores para BFT [Castro e Liskov 1999, Kotla et al. 2007]. Obviamente, a utilização de máquinas virtuais ao invés de máquinas físicas já proporciona tal redução. Porém, nosso trabalho também busca reduzir o número total de réplicas necessárias para tolerar f réplicas bizantinas, diminuindo de $3f + 1$ para $2f + 1$.

Nossa proposta se baseia na abordagem *Replicação Máquina de Estados* [Schneider 1990] (também descrita na Seção 2.1.4.1), onde as máquinas de estado replicadas são sistemas operacionais executando sobre VMs. De acordo com a Figura 4.1, um único sistema hospedeiro (*SO Anfitrião*) suporta a execução do VMM (*Monitor de Máquinas Virtuais*), bem como todo o conjunto de máquinas virtuais ($VM_0, VM_1, ..., VM_{N-1}$). Cada VM oferecida pelo VMM suporta a execução de um sistema operacional, que executa a camada de replicação de nossa proposta (protocolo de consenso), bem como a aplicação a ser utilizada como serviço.

Ainda sobre a Figura 4.1, é possível perceber que o VMM oferece às máquinas virtuais um componente chamado de *postbox*, que nada mais

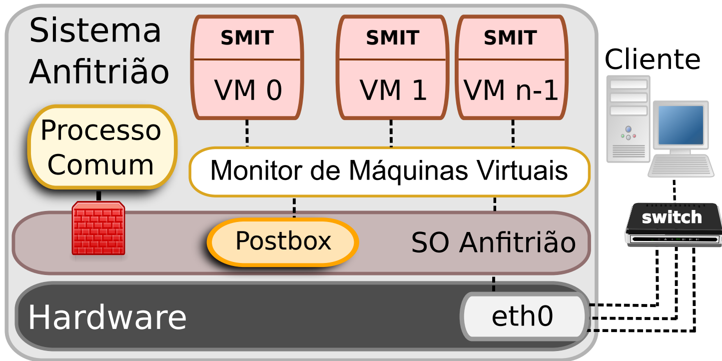


Figura 4.1: Visão Geral da Arquitetura SMIT

é do que uma área de memória compartilhada entre as VMs, cujas propriedades e detalhes serão descritos na Seção 4.3. Tal componente será utilizado como único meio de comunicação entre as réplicas de serviço. A comunicação entre as réplicas de serviço é necessária para que estas obtenham consenso sobre a seqüência na qual as requisições recebidas dos clientes serão executadas.

Os termos *faltas bizantinas* e *intrusões* por vezes se confundem no decorrer do texto. Isso se deve ao fato de SMIT oferecer tolerância a intrusões através de técnicas de tolerância a faltas bizantinas. Acreditamos que o termo *Tolerância a Intrusões* seja mais adequado para a arquitetura como um todo, visto que a utilização de redundância de serviço em máquinas virtuais aumenta a resistência de sistemas baseados em uma única máquina física, principalmente em se tratando de faltas maliciosas. Como a abordagem centralizada de SMIT não tolera faltas de *crash* no sistema anfitrião, o termo *Tolerância a Intrusões* se torna mais adequado que *Tolerância a Faltas Bizantinas*.

4.2 Modelo de Sistema

Esta Seção apresenta o modelo de sistema utilizado em nossa proposta. Assim como apresentado na Seção 2.2, a descrição do modelo de

sistema será feita de acordo com os aspectos dos modelos de interação, de falhas e de segurança [Coulouris, Dollimore e Kindberg 2005].

4.2.1 Modelo de Interação

De acordo com os aspectos descritos na Seção 2.2.1, SMIT utiliza um modelo composto por dois grupos de processos que interagem através de comunicação pela rede: os *Servidores* e os *Clientes*. O conjunto de servidores, $S = \{s_0, s_1, ..., s_{n-1}\}$, é formado por n processos executando sobre máquinas virtuais, as quais executam sobre uma única máquina física. Os clientes, aqui representados pelo conjunto de tamanho arbitrário $C = \{c_0, c_1, ...\}$, são processos que enviam requisições para os servidores através da rede.

Para que os servidores componentes de S façam progresso mantendo seus estados internos do serviço mutuamente consistentes, é necessário Determinismo de Réplica [Schneider 1990], que define que réplicas que iniciam sua execução em um mesmo estado inicial e executam a mesma seqüência de operações, devem terminar suas execuções em um mesmo estado final. Isso é necessário para que o estado do serviço oferecido ao cliente permaneça consistente através das réplicas.

Dada a estreita relação de nosso trabalho com a virtualização de sistemas (descrita em detalhes na Seção 2.3), os processos de nosso modelo podem também ser descritos de acordo com conceitos de máquinas virtuais, descrevendo os processos de acordo com o papel que eles exercem no ambiente virtualizado. Sendo assim, nosso modelo é composto por três tipos de processos:

- **Entidades externas ao ambiente virtualizado**: os *clientes*, que são sistemas que executam sobre suas próprias máquinas físicas, não sendo integrantes do sistema virtualizado que compreende SMIT.

- **Sistemas Convidados**: as réplicas de serviço executando sobre máquinas virtuais que executam sobre um único sistema físico, o *Anfitrião*.

- **Sistema Anfitrião**: pode ser representado por um VMM ou por um sistema operacional sobre o qual um VMM está sendo executado. O sistema anfitrião hospeda todos os servidores contidos no grupo S.

Com relação a interação entre os processos, assumimos um sistema assíncrono onde os clientes e servidores interagem através de uma rede, a qual pode perder, atrasar, reordenar e corromper os pacotes. Apesar

de seguir o modelo assíncrono, para garantia de propriedade de *liveness* (Seção 4.4), é necessário certo grau de sincronia, onde assumimos que uma mensagem será, em algum momento, entregue ao seu destinatário. A interação entre as réplicas de serviço (componentes de S) se dá única e exclusivamente através de uma abstração de memória compartilhada a ser fornecida pelo sistema anfitrião, a qual é denominada como *Postbox*.

Uma descrição formal sobre acesso à memória compartilhada foi introduzida em [Fernandez, Jimenez e Raynal 2007], que define hipóteses para que uma entidade consiga acessar tal componente. A memória é descrita através de registradores compartilhados com noção de pertinência. A hipótese AWB_1 requer que, em algum momento, um processo não se comporte de forma totalmente assíncrona, possibilitando assim o acesso à memória compartilhada dentro de um tempo limitado (não necessariamente conhecido). Tal propriedade foi definida como [Fernandez, Jimenez e Raynal 2007]:

AWB_1: *Existe um tempo τ_{01}, um limite Δ, e um processo correto p_l, tais que, após τ_{01}, quaisquer dois acessos consecutivos emitidos por p_l para seus registradores na memória compartilhada são completados em no máximo Δ unidades de tempo.*

Isso significa que, após determinado tempo finito, um processo passa a se comportar de forma *parcialmente síncrona* [Dwork, Lynch e Stockmeyer 1988], isto é, os limites de tempo para execução de passos existem, porém não são previamente conhecidos, ou então, os limites de tempo são conhecidos, porém somente são cumpridos após um Tempo de Estabilização Global (*Global Stabilization Time* - GST). Dessa forma, no contexto do trabalho aqui apresentado, os acessos à memória compartilhada por parte das réplicas se dão de acordo com a hipótese AWB_1 [Fernandez, Jimenez e Raynal 2007]. Maiores detalhes relativos à *Postbox* são fornecidos na Seção 4.3.

Os processos que compõem SMIT possuem relógios locais não-sincronizados uns com os outros. Tais relógios são utilizados apenas para marcação das mensagens submetidas pelos clientes, de modo que ao criar uma mensagem de requisição, o cliente realiza uma leitura de seu relógio local e marca a mensagem com o valor obtido, identificando unicamente tal mensagem dentro do conjunto de requisições que ele emite. Através dessa marcação, é possível aos servidores verificar, para cada cliente, se as mensagens enviadas por tal cliente estão sendo recebidas e processadas na ordem cronológica do cliente, e também é possível a garantia da semântica de execução *at most once*, relativa às mensagens submetidas pelos clientes.

4.2.2 Modelo de Falhas

De acordo com as definições apresentadas na Seção 2.2.2, no contexto de SMIT os processos componentes do grupo S podem sofrer falhas bizantinas (arbitrárias), as quais, de acordo com as relações apresentadas na Seção 2.1.1.4, são reconhecidas como faltas bizantinas em uma visão de SMIT como um todo, observando cada processo como um componente deste. A resistência oferecida por SMIT assegura que até $f = [\frac{N-1}{2}]$ processos podem apresentar comportamento faltoso, ou de acordo com a outra notação, SMIT necessita de $N \geq 2f + 1$ processos em S no total. Os clientes, membros do grupo C, podem apresentar comportamento bizantino e não há limites quanto ao número de clientes bizantinos tolerado, uma vez que clientes faltosos não comprometem a execução correta de SMIT.

Assumimos em nosso modelo a utilização de *Diversidade de Software* [Avizienis et al. 2004]. Assim, as réplicas de serviço são construídas sobre sistemas operacionais distintos, de forma que tais sistemas não compartilhem *bugs* de software. Caso apresentem os mesmos *bugs* e sofram as mesmas faltas, manter a premissa de que apenas f membros são maliciosos em um total de $2f + 1$ réplicas se torna mais complicado, visto que basta descobrir uma vulnerabilidade e a maneira de explorá-la em uma das réplicas para que se consiga fazer o mesmo com as réplicas restantes.

Assumimos que o sistema anfitrião, bem como o VMM, pode apresentar vulnerabilidades, porém estas não podem ser exploradas. Possíveis atacantes seriam máquinas virtuais corrompidas, ou mesmo entidades externas acessando diretamente o hospedeiro. Para evitar o primeiro tipo de atacante, confiamos que o VMM forneça isolamento adequado entre as VMs e o hospedeiro. O segundo tipo de atacante pode ser evitado bloqueando acesso direto pela rede ao sistema hospedeiro e a seus processos comuns (como mostra a Figura 4.1), o que pode ser obtido utilizando filtro de pacotes, bloqueando o acesso direto ao sistema hospedeiro através de regras específicas. Em hospedeiros GNU/Linux, isso pode ser obtido através da ferramenta de filtragem de pacotes Iptables [Andreasson 2003]. Dessa forma, os clientes se comunicam diretamente com as VMs (réplicas de serviço), através de interfaces de rede específicas destas. Diferentemente da presente proposta, outros trabalhos baseados em tolerância a faltas bizantinas através de máquinas virtuais assumem a existência de um anfitrião confiável [Reiser et al. 2006, Reiser e Kapitza 2007, Chun, Maniatis e Shenker 2008].

O isolamento provido pelo VMM somado a utilização de diversidade de software na construção das réplicas de serviço auxiliam na obtenção de independência de faltas em SMIT. Dessa forma, uma falta sofrida

em uma das réplicas de serviço não implica em uma falta em outra réplica de serviço.

Sistemas replicados virtualmente em uma única máquina física, como SMIT, possuem a limitação de serem vulneráveis a faltas de *crash* no sistema físico. Ao ocorrer um *crash* no sistema anfitrião, todas as réplicas de serviço do grupo S (executando sobre máquinas virtuais) também sofrem o *crash*. Para contornar tal limitação, apresentamos uma abordagem com replicação tolerante a faltas de *crash* para o sistema anfitrião, chamada de DSMIT (*Distributed* SMIT), a qual utiliza um modelo híbrido de falhas [Correia et al. 2002].

Através da utilização de um modelo híbrido de falhas [Correia et al. 2002], é possível a realização de hipóteses de falhas diferentes para diferentes componentes de um sistema. Em nossa abordagem distribuída (DSMIT), o sistema anfitrião é replicado fisicamente, sendo assim feita a replicação de todo o conjunto S de réplicas de serviço pertencentes a SMIT. Assim como em SMIT, às réplicas de serviço (executando sobre máquinas virtuais) é permitido falhar de forma arbitrária. Diferentemente de SMIT, onde ao sistema anfitrião não é permitida a falha de *crash*, em DSMIT, através da replicação física do anfitrião, assumimos que as falhas possíveis neste sistema estão restritas às falhas de *crash*.

Na Seção 4.7, mostraremos que a utilização do modelo híbrido de falhas nos permite aplicar diferentes abordagens de replicação em diferentes níveis do sistema. Assim, pudemos aplicar técnicas de replicação tolerantes a faltas de *crash*, como a abordagem primário-*backup*, para a replicação do sistema anfitrião. Maiores detalhes sobre DSMIT são apresentados na Seção 4.7.

4.2.3 Modelo de Segurança

Dadas as preocupações com segurança descritas na Seção 2.2.3, todas as mensagens trocadas por clientes e réplicas em SMIT são assinadas utilizando técnicas de criptografia assimétrica [Rivest, Shamir e Adleman 1978]. Assumimos que um atacante não possui poder computacional para quebrar as técnicas criptográficas utilizadas. Para mensagens trocadas via memória compartilhada entre as réplicas, a assinatura não é necessária, visto que o VMM realiza esse gerenciamento, e tem controle total sobre as VMs.

Com relação à questão de *Autenticidade* das mensagens, em SMIT aplicamos tais técnicas de criptografia assimétrica, onde o remetente assina a mensagem utilizando sua chave privada e o destinatário verifica a autenticidade desta através da chave pública do remetente. Para cada par comunicante $\{m_0, m_1\}$, onde m_0 e m_1 são membros do conjunto de

clientes e do conjunto de réplicas, respectivamente, m_0 deve conhecer a chave pública de m_1 e vice-versa. Caso contrário, ambos não serão capazes de checar a integridade e autenticidade das mensagens trocadas entre eles.

Assim como a autenticidade, a *Integridade* das mensagens em SMIT é garantida através de assinatura usando criptografia assimétrica. Caso o conteúdo da mensagem tenha sido modificado durante sua transmissão, a assinatura contida na mensagem e a verificação realizada pelo destinatário não irão corresponder uma a outra, sendo possível assim detectar a alteração.

A questão do *Controle de Acesso* é dependente da aplicação que for utilizada em conjunto com nossa proposta. Porém, em SMIT podem ser utilizados os mecanismos de criptografia assimétrica acima juntamente com uma tabela de privilégios de usuários para resolver essa questão.

Assim como nas outras questões, a questão de *Confidencialidade* pode ser resolvida em SMIT pela utilização de criptografia assimétrica, fazendo com que o remetente criptografe a mensagem utilizando a chave pública do destinatário antes de enviá-la, de modo que somente o destinatário, detentor da chave privada correspondente, poderá decifrá-la e ler o conteúdo da mensagem.

4.3 Componente *Postbox*

A *postbox* é o principal componente de nosso trabalho, visto que é através dela que as réplicas de serviço se comunicam visando obter consenso. Como já fora descrito anteriormente, a *postbox* é uma abstração de memória compartilhada, oferecida pelo VMM para acesso pelas máquinas virtuais. Sendo a *postbox* um componente único e de acesso a todas as máquinas virtuais, ao escrever um valor neste componente, *uma réplica está difundindo atomicamente tal valor para todas as réplicas corretas*. Para isso, são necessárias algumas hipóteses e construções, como será descrito a seguir.

4.3.1 Funcionamento da *Postbox*

A *postbox* é utilizada pelas réplicas de serviço (VMs) para troca de mensagens entre si. Assim, todas as réplicas devem possuir acesso de leitura e escrita (em modo *append*) a tal componente. A *postbox* possui uma interface simples, composta por dois métodos:

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

O método ***append*** deve gravar, de forma atômica, um valor na *postbox*, juntamente com um identificador da réplica que o gravou (o VMM gerencia isso, então não há falsificação sobre o remetente da mensagem). O valor é armazenado em uma posição imediatamente após o valor gravado na última operação *append* que foi realizada. O valor de retorno é do tipo booleano, indicando se a operação foi realizada com sucesso ou não. O método ***read*** retorna o valor imediatamente posterior ao valor retornado na última operação *read* executada pela réplica que invocou a operação. Resumindo, os valores são lidos e escritos por cada réplica em modo FIFO (*First in, First Out*).

Um detalhe crucial para nossa abordagem é que a *postbox* deve operar em modo *append-only*, ou seja, uma vez lá armazenado, um valor não pode ser alterado. Essa restrição evita a situação na qual uma réplica maliciosa escreve um valor, aguarda até que uma ou mais das réplicas realizem a leitura, e então altera tal valor, de modo que as réplicas que ainda estão por realizar a leitura correspondente leiam valores diferentes dos lidos pelas primeiras. Tal situação poderia gerar uma inconsistência em todo o serviço replicado.

Dado esse funcionamento, podemos garantir que todas as réplicas corretas no conjunto S lêem todos os valores na mesma ordem em que foram escritos na *postbox*. Ou seja, as operações de leitura realizadas pelo conjunto de réplicas corretas ocorrem em Ordem Total. Assim, para qualquer conjunto de operações W_i contendo *k* operações de escrita, que vão da *n-ésima* até a *(n+k)-ésima* posição da *postbox*, todas as réplicas corretas que realizarem um conjunto de *k* operações de leitura a partir da posição *n* possuirão um conjunto de leituras R_i, com conteúdo idêntico a W_i. Ou seja, todas as réplicas corretas possuem uma visão homogênea do conteúdo da *postbox*.

A Figura 4.2 apresenta um exemplo de utilização da *postbox*. Conforme pode ser visto na figura, considere que as escritas e leituras na *postbox* ocorrem seguindo a ordem crescente do campo *posição*. Dessa forma, analisando a figura, tivemos a ocorrência do seguinte cenário: a VM 0 escreveu a mensagem $m_{0,0}$, em seguida, a VM 2 escreveu a mensagem $m_{1,0}$ e a VM 1 escreveu a mensagem $m_{1,1}$, após isso a VM 0 escreveu a mensagem $m_{0,1}$. De acordo com as propriedades e hipóteses descritas anteriormente, todas as réplicas corretas que executarem as operações de leitura sobre a *postbox* obterão a seguinte seqüência de valores $m_{0,0}, m_{1,0}, m_{1,1}, m_{0,1}$. A operação de leitura posterior irá bloquear, visto que não há valores escritos a partir da posição 4.

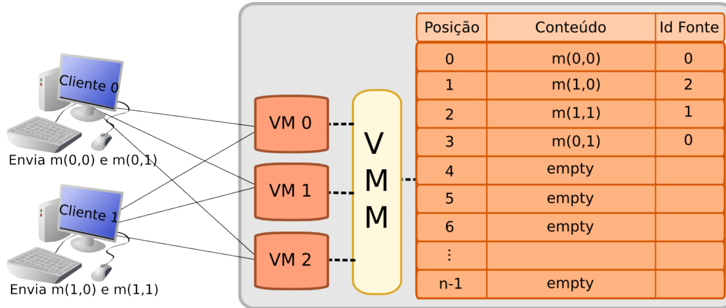


Figura 4.2: Exemplo de utilização da *Postbox*

4.3.2 Detalhes de Construção da *Postbox*

A *postbox* é um componente que deve ser fornecido pelo VMM para os sistemas convidados. O VMM deve fornecer mecanismos básicos de controle de acesso sobre a *postbox*, bem como garantir as escritas em modo *append-only* por parte das máquinas virtuais. Além disso, para garantir que as VMs obtenham acesso à *postbox* tão logo necessário, o VMM deve empregar algoritmos de escalonamento de acesso à *postbox* que sejam suficientemente justos, de modo que uma VM não seja impedida indefinidamente de realizar operações no componente devido a utilização deste por outras VMs, cumprindo a propriedade AWB_1, descrita na Seção 4.2.1.

Sendo implementado em memória, a *postbox* é um componente de capacidade finita. Dessa forma, para preservar a corretude do protocolo, é importante que seja implementada uma política de coleta de lixo, que faça a remoção de valores da *postbox* que não são mais necessários pelas VMs. É importante perceber que caso tal política não seja implementada, ou seja mal parametrizada com relação à capacidade de memória do sistema hospedeiro, o protocolo pode vir a sofrer uma parada em sua execução, visto que ocorrerá um *overflow* do principal componente da arquitetura, a *postbox*. O protocolo de coleta de lixo será apresentado em detalhes no Capítulo 5, visto que seu funcionamento é dependente da implementação da *postbox*.

É de fundamental importância para a corretude do sistema que esse componente seja implementado contemplando todos os detalhes e restrições descritas nessa seção. Caso contrário, uma falha no componente *postbox* poderia causar a falha de toda a arquitetura. Sendo provida pelo

VMM, assumimos que, caso a *postbox* seja implementada seguindo corretamente a especificação aqui apresentada, o VMM deverá garantir o funcionamento correto desse componente.

4.4 Propriedades de SMIT

De modo geral, sistemas distribuídos são descritos de forma que sejam capazes de garantir as propriedades de *safety* e *liveness*. Através dessas propriedades, é possível provar a corretude desses sistemas. De acordo com Lamport [Lamport 1977], garantir *safety* significa garantir que "*coisas ruins*" não irão acontecer durante o período de execução do sistema, enquanto que garantir *liveness* significa assegurar que "*coisas boas*" irão acontecer, em algum momento. Adaptando tais propriedades à nossa proposta, é necessário que SMIT assegure ambas as propriedades abaixo, conforme sua descrição:

- ***Safety***: todas as réplicas corretas de SMIT executam as *mesmas requisições na mesma seqüência*.
- ***Liveness***: requisições submetidas por clientes corretos a SMIT terão sua execução completada *em algum momento*, ou seja, não importa o que aconteça, *o protocolo sempre faz progresso*.

Na abordagem Replicação Máquina de Estados, para que a propriedade *safety* seja garantida, é necessário que todas as requisições enviadas por clientes corretos sejam recebidas (*Acordo*) e executadas na mesma ordem (*Ordem Total*) por todas as réplicas corretas [Schneider 1990]. Além disso, o serviço provido pelas réplicas corretas deve apresentar comportamento determinístico.

Após a apresentação do protocolo de consenso, na Seção 4.6 serão apresentadas provas de correção deste, demonstrando que as propriedades acima referidas são satisfeitas em nossa proposta.

4.5 Protocolo

Para executar sobre a arquitetura descrita nas seções anteriores, propusemos um protocolo de consenso para que seja possível a execução de serviços Tolerantes a Intrusões sobre esta. A presente seção apresenta os detalhes desse protocolo. Sucintamente, uma execução correta do nosso protocolo funciona da seguinte maneira:

1. Cliente c_j envia uma requisição req para as réplicas do conjunto S;
2. As réplicas do grupo S recebem req e alguma réplica s_i é a primeira a propor a execução de req, escrevendo uma proposta de execução para req, a qual inclui a mensagem inteira, na *postbox*;

3. Todas as réplicas corretas de S lêem a proposta feita por s_i na *postbox*, executam a mensagem na ordem proposta e enviam a resposta diretamente para c_j;

4. c_j aguarda até que receba $f + 1$ respostas mutuamente consistentes para req.

A Figura 4.3 apresenta o diagrama de execução de uma requisição na arquitetura SMIT, utilizando o protocolo apresentado na presente seção. Conforme já fora descrito acima, na primeira fase, chamada de *REQUEST*, o cliente difunde a requisição para as réplicas, as quais passam para a fase seguinte. Nessa fase, chamada de *CONSENSUS*, todas as réplicas escrevem na *postbox* as propostas de execução para a mensagem recebida do cliente. Na mesma fase, as réplicas obtêm, através de leituras na *postbox* a próxima mensagem a ser processada. Para garantir que as réplicas obtenham consenso sobre a mensagem a ser executada, todas as réplicas corretas aplicam uma mesma função determinística sobre os valores lidos da *postbox*, componente sobre o qual todas as réplicas possuem uma visão homogênea. A função determinística utilizada em nossa proposta é "*executar a próxima mensagem lida, ainda não processada, corretamente assinada e com timestamp crescente*". Ou seja, uma requisição req é processada na ordem definida pela primeira réplica a propor sua execução na *postbox*. As propostas das outras réplicas são ignoradas. Como todas as réplicas corretas possuem visão homogênea da *postbox*, garantimos que estas entrarão em consenso sobre a ordem de execução da mensagem. Após o consenso obtido, as réplicas executam a requisição e enviam a resposta diretamente para o cliente, o que compõe a fase *REPLY*.

Diferentemente de abordagens anteriores [Castro e Liskov 1999, Yin et al. 2003, Kotla et al. 2007, Luiz et al. 2007], nossa proposta não faz uso de réplica primária fixa para proposição da ordem de execução para as requisições. Ao invés disso, utilizamos uma abordagem na qual a primeira proposta válida para uma requisição na *postbox* é considerada a proposta a ser seguida, não importando qual réplica a propôs. Tal abordagem somente é possível pela utilização do componente *postbox*, que é um componente único e que fornece uma visão homogênea de seu conteúdo para as réplicas corretas. Abordagens baseadas na idéia de réplica primária para definição de ordem de execução necessitam de mecanismos para detecção de primário faltoso e de troca de primário, o que gera uma complicação adicional ao protocolo. Utilizando uma abordagem descentralizada com relação a proposição de ordens de execução, temos a vantagem de não depender diretamente da corretude de uma réplica, especificamente, para garantir a corretude do protocolo, pois todas

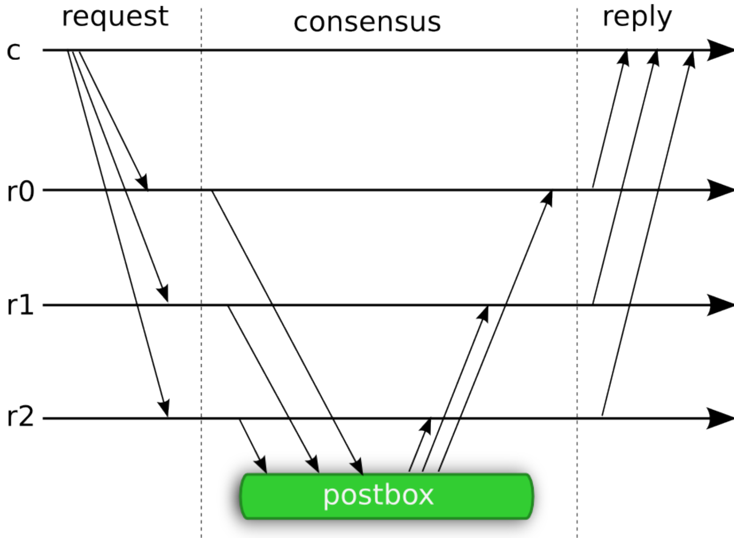


Figura 4.3: Diagrama de execução do SMIT

as réplicas do conjunto S podem propor valores para execução, sendo a primeira mensagem correta escrita considerada válida.

A seguir, as etapas brevemente descritas na enumeração e na Figura 4.3, serão abordadas em detalhes, de modo a explicitar todos os aspectos relacionados a execução de requisições na arquitetura proposta.

4.5.1 Tipos de Mensagens

Os membros envolvidos (clientes e réplicas) no processo de execução de requisições se comunicam, seja através da *postbox* ou de passagem de mensagens. A comunicação entre tais membros é necessária para as seguintes situações:

1. **Submissão de requisições**: cliente envia mensagem para as réplicas (mensagens do tipo *REQ*).

2. **Estabelecimento de consenso entre as réplicas**: réplicas gravam mensagens na *postbox* (mensagens do tipo *PROP*).

3. **Envio da resposta para o cliente**: réplicas enviam mensagens para o cliente (mensagens do tipo *REPLY*).

As mensagens do tipo *REQ*, enviadas quando da ocorrência da situação 1, são compostas pelos seguintes campos:

- *data*: refere-se ao *payload* da mensagem. Esse campo é dependente da aplicação.
- *id*: identificador do cliente remetente da mensagem.
- *t*: *timestamp* do momento do envio da requisição. Necessário para garantia da semântica *at most once*.
- *sgn*: assinatura do remetente da mensagem, gerada utilizando o *payload* (campo *data*), o campo *id*, o campo *t* e a chave privada do cliente.

As mensagens do tipo *PROP* (situação 2), trocadas somente entre as réplicas através da *postbox*, são compostas pelos seguintes campos:

- *req*: requisição recebida do cliente (mensagem completa).
- *id*: identificador da réplica que escreveu a mensagem na *postbox*.

Por fim, as mensagens do tipo *REPLY* (situação 3) são compostas pelos seguintes campos:

- *rep*: conteúdo da resposta para a requisição recebida.
- *id*: identificador da réplica remetente.
- *t*: *timestamp* de origem da mensagem processada.
- *sgn*: assinatura da réplica, gerada utilizando o campo *rep* e a chave privada da réplica.

4.5.2 O Lado Cliente

O Algoritmo 4.1 apresenta o lado cliente do protocolo. Como mostra a linha 1, uma mensagem do tipo *REQ* é enviada para o conjunto de réplicas S. Após o envio da requisição, o cliente aguarda até que receba $f+1$ respostas *mutuamente consistentes* (conforme explicado abaixo). O recebimento de $f+1$ mensagens mutuamente consistentes assegura que *ao menos uma* das $f+1$ mensagens tem como procedência uma réplica correta (visto que apenas f réplicas podem ser faltosas), o que significa que todas as $f+1$ mensagens possuem conteúdo válido.

Para que $f+1$ mensagens sejam mutuamente consistentes e sejam aceitas por um cliente como resposta para uma requisição req, elas devem satisfazer as seguintes condições:

Algoritmo 4.1 Envio de requisição do cliente para o grupo de réplicas S

```
1: multicast_send(⟨REQ, data, id, t, sgn⟩, S)
2: repeat
3:    buffer ← buffer ∪ recv()
4: until f + 1 matching replies ∈ buffer
```

1. Campo *rep* (conteúdo da resposta) idêntico em todas as mensagens.
2. Campo *t* (*timestamp* da requisição) idêntico em todas as mensagens.
3. Campo *t* idêntico ao *timestamp* de req.
4. Mensagens corretamente assinadas por seus remetentes.
5. Todas as $f + 1$ mensagens remetidas por diferentes réplicas.

4.5.3 O Lado das Réplicas

O lado das réplicas do protocolo é composto por dois fluxos básicos de execução, os quais são executados de forma concorrente em cada uma das réplicas. Um deles, descrito no Algoritmo 4.2 é responsável por receber as requisições do cliente e propor a execução de tais requisições através da *postbox*. O outro fluxo, descrito no Algoritmo 4.3, é responsável por ler mensagens da *postbox* e executá-las.

O Algoritmo 4.2 permanece sempre aguardando por mensagens dos clientes através da rede (linha 2). Ao receber uma requisição *req*, a réplica verifica se o *timestamp* de tal mensagem é maior do que o *timestamp* da última mensagem processada remetida pelo mesmo cliente (linha 3). Caso não seja, a réplica verifica se a resposta para tal requisição se encontra em sua *cache* de respostas a requisições (linha 8) e reenvia a resposta para o cliente (linha 9). Tal situação pode ocorrer quando o cliente retransmite uma requisição, possivelmente devido ao fato de o envio da resposta por uma ou mais réplicas ter sido atrasado ou mesmo perdido pela rede. Para isso, as réplicas sempre armazenam a resposta para a última requisição submetida por cada cliente. Maiores detalhes sobre essa *cache* são apresentados na seqüência do texto.

Caso o *timestamp* seja maior, a réplica realiza a verificação de integridade e autenticidade de *req* (linha 4). Tal verificação é realizada utilizando a assinatura incorporada na mensagem (campo *sgn*), o *payload* da mensagem (campo *data*) e a chave pública do cliente, de conhecimento prévio de todas as réplicas. Caso a mensagem seja válida e ainda não tenha sido processada, a réplica escreve uma mensagem do tipo *PROP* na

postbox (linha 5), propondo assim uma ordem para execução desta mensagem.

Algoritmo 4.2 Algoritmo de Recebimento de Requisições

```
1:  loop
2:      req ← receive()
3:      if req.t > last_ts[req.id] then
4:          if signed(req) then
5:              postbox.append(⟨PROP, req, id⟩)
6:          end if
7:      else if req.t = last_ts[req.id] then
8:          rep ← last_processed[req.id]
9:          send(⟨REPLY, rep, id, req.t, sgn⟩, rep.id)
10:     end if
11: end loop
```

O Algoritmo 4.2 mantém no estado interno das réplicas uma *cache* composta por duas tabelas usadas para garantia da semântica de execução *at most once*. Visto que ambas são tabelas indexadas pelo código identificador do cliente, ambas armazenam apenas uma entrada por cliente. As variáveis armazenadas são as seguintes:

- *last_ts[]*: tabela indexada pelo identificador de cliente, contendo, em cada posição i, o *timestamp* da última requisição vinda do cliente i processada pela réplica.
- *last_processed[]*: tabela indexada pelo identificador de cliente, contendo, em cada posição i, a resposta para a última requisição vinda do cliente i processada pela réplica.

É necessário um mecanismo de coleta de lixo sobre essas tabelas, para evitar que uma quantidade grande de clientes páre o protocolo devido à falta de memória nas réplicas para armazenar os valores relativos a todos os clientes. Um mecanismo de temporização sobre os elementos de tais tabelas soluciona o problema, sendo realizada a remoção dos elementos nela escritos, cuja operação de escrita tenha sido realizada há um período de tempo maior que o período determinado para manutenção dos valores na *cache*. Dessa forma, é estabelecido um período de tempo máximo, durante o qual os valores serão mantidos na *cache* de requisições processadas.

O Algoritmo 4.3 permanece constantemente checando a *postbox* por novas propostas de execução (linha 2). Se a operação de leitura re-

tornar uma mensagem de proposta *prop* (linha 3), a réplica verifica se a requisição *req* contida em *prop* possui uma assinatura válida (linha 4), de acordo com a chave pública do cliente *prop.req.id*, e se o *timestamp* (*prop.req.t*) da mesma apresenta ordem crescente com relação ao *timestamp* das mensagens anteriores do mesmo cliente. Se ambas as condições forem verdadeiras, a réplica modifica seu estado interno (ou não, no caso de uma requisição de somente-leitura) executando a requisição *prop.req* (linha 5), adiciona a resposta para a mensagem na sua *cache* que armazena a última requisição processada (linha 6), grava o valor do timestamp *prop.req.t* da última requisição processada (linha 7) e envia uma mensagem do tipo *REPLY* (*rep*), contendo a resposta assinada para a requisição do cliente (linha 8).

Algoritmo 4.3 Leitura da *Postbox* e execução das requisições

```
 1: loop
 2:   prop ← postbox.read()
 3:   if prop = ⟨PROP, req, id⟩ then
 4:     if prop.req.t > last_ts[prop.req.id] ∧ signed(prop.req) then
 5:       rep ← execute(prop.req.data)
 6:       last_processed[prop.req.id] ← rep
 7:       last_ts[p.req.id] ← p.req.t
 8:       send(⟨REPLY, rep, p.req.t, id, sgn⟩, p.req.addr)
 9:     end if
10:   end if
11: end loop
```

4.6 Correção de SMIT

Conforme descrito na Seção 4.4, SMIT deve satisfazer as propriedades de *safety* e *liveness*.

Na abordagem Replicação Máquina de Estados, para que a propriedade *liveness* seja garantida, é necessário que todas as requisições enviadas por clientes corretos sejam recebidas (*Acordo*) e executadas na mesma ordem (*Ordem Total*) por todas as réplicas corretas [Schneider 1990]. Além disso, o serviço provido pelas réplicas corretas deve apresentar comportamento determinístico.

Comecemos relembrando algumas hipóteses assumidas anteriormente, as quais serão úteis para a elaboração das provas de correção.

1. *Postbox append-only*: uma vez que um valor é escrito na *postbox*, ele não pode ser alterado.

2. O conjunto de réplicas de serviço corretas possui uma visão homogênea do conteúdo da *postbox*.
3. Existem, no mínimo, *2f+1* réplicas de serviço em um *host*, das quais apenas *f* podem apresentar comportamento bizantino.
4. O cliente retransmite as mensagens para as réplicas enquanto não receber uma resposta. Para isso, inicia um *timeout* a cada envio.
5. Os tempos de atraso da rede não crescem indefinidamente.
6. O escalonamento de escritas e leituras na *postbox* deve ser justo o suficiente para que todos possam executar suas operações nela assim que possível, cumprindo a propriedade AWB_1.
7. Uma réplica maliciosa não pode falsificar mensagens na *postbox*.
8. A função de execução do serviço nas réplicas é determinística.

Lema 1. *Uma requisição req, enviada por um cliente correto, será, em algum momento, entregue, executada e respondida pelas réplicas corretas.*

Prova (esboço). Através das hipóteses 4 e 5, garantimos que, em algum momento, uma réplica correta s_i receberá req. Assim, ao receber tal mensagem, s_i irá gravar uma proposta de execução para req na *postbox*, propagando assim req para todas as réplicas corretas restantes, visto que a proposta contém o conteúdo da requisição por inteiro. Assim, mesmo que uma réplica correta s_j ainda não tenha recebido req do cliente, s_j será capaz de obter req através da *postbox*, podendo assim executar e responder a requisição, ou seja, basta que uma réplica correta receba req para que todas as réplicas corretas também tenham acesso a req. □

Lema 2. *Se uma réplica correta s_i processa uma requisição req, então todas as réplicas corretas também processarão req na mesma seqüência que s_i.*

Prova (esboço). Uma réplica correta executa, obrigatoriamente, os algoritmos que compõem SMIT, tal como descrito em sua especificação. Assim, de acordo com o Algoritmo 4.3 (leitura de ordens), uma réplica correta somente executa uma requisição após ter realizado a leitura de uma proposta de execução para tal mensagem da *postbox*. De acordo com as hipóteses 1 e 2, todas as réplicas lêem exatamente os mesmos valores da *postbox* e na mesma ordem. Acrescente-se a isso a hipótese 6, que exige que todas as réplicas terão acesso a *postbox* assim que for necessário.

Assim, para que s_i processe req, é necessário que ele tenha lido req da *postbox*. Se s_i leu req da *postbox*, então todas as réplicas corretas restantes também o farão, pois executam exatamente o mesmo protocolo e tem acesso a mesma *postbox*, obtendo a mesma requisição req e a processando na mesma seqüência que s_i a processou. □

Lema 3. *SMIT garante que uma requisição req, recebida por uma réplica correta, terá mesmo resultado em ao menos $f + 1$ réplicas.*

Prova (esboço). De acordo com a hipótese 3, no mínimo $f + 1$ réplicas são corretas. Assim, conforme provado no Lema 1 e acrescentando o que foi assumido na hipótese 8, garantimos que no mínimo $f + 1$ réplicas irão executar req deterministicamente. Tendo elas executado a mesma seqüência de requisições anteriormente a req, a execução da requisição irá produzir o mesmo resultado para as $f + 1$ réplicas. □

Levando em consideração os Lemas expostos acima, podemos provar ambas as propriedades de *Safety* e *Liveness*.

Teorema 1. *Os comandos enviados ao SMIT por clientes corretos são todos executados pelas réplicas corretas em ordem total.*

Prova (esboço). Conforme o Lema 1, todas as réplicas corretas recebem as requisições enviadas pelos clientes corretos (acordo no recebimento) e, de acordo com o Lema 2, todas as réplicas corretas irão executar as mesmas requisições em uma mesma seqüência (ordem total). Dessa forma, é garantido que todas as réplicas corretas recebem e executam todas as requisições corretas em uma mesma seqüência, mantendo consistente o estado do serviço. □

Teorema 2. *O protocolo SMIT sempre faz progresso.*

Prova (esboço). Este teorema será provado por contradição. Considere, sem perda de generalidade, que c_1 e c_2 são clientes que emitem requisições para o grupo de réplicas de SMIT. Suponha que ambos permanecem bloqueados para sempre, aguardando por respostas para suas requisições. Será mostrado que a ocorrência de tal situação é impossível. De acordo com o Algoritmo 4.2, ao receber uma requisição correta (assinada e não executada anteriormente) do cliente, uma réplica correta escreve uma proposta de execução para tal mensagem na *postbox*. Sabendo que no mínimo $f + 1$ réplicas são corretas e conforme o Lema 1, podemos garantir, inspecionando o Algoritmo 4.2, que, no mínimo, $f + 1$ réplicas corretas receberão as requisições dos clientes e escreverão propostas para estas na

postbox, tornando tais requisições acessíveis para todas as réplicas. Assim, a execução do Algoritmo 4.3 pelas réplicas corretas mostra que estas irão ler as propostas da *postbox* e executar as mensagens correspondentes, desde que estas sejam corretas e que ainda não tenham sido executadas. Conforme provado no Teorema 1, todas as réplicas corretas recebem todas as requisições de clientes corretos na mesma seqüência. Isso, associado ao Lema 3, que diz que uma requisição terá no mínimo $f + 1$ respostas com mesmo resultado, garante que os clientes podem fazer progresso em suas execuções, visto que estes aguardam por apenas $f + 1$ mensagens com respostas iguais para uma mesma requisição. Tal cenário contradiz a situação inicial onde os clientes permanecem bloqueados. Mesmo que um dos clientes seja malicioso e tente parar o protocolo enviando mensagens inconsistentes para as réplicas, isso não afeta o progresso do protocolo para o cliente correto, visto que toda mensagem recebida por uma réplica correta será propagada, através da *postbox* para as outras réplicas, e, caso tal mensagem não seja corretamente assinada, esta será descartada. □

4.7 *Distributed* SMIT

Sistemas replicados virtualmente em uma única máquina física possuem a limitação de serem vulneráveis a faltas de *crash* no sistema físico. Em SMIT, ao ocorrer um *crash* no anfitrião, todas as réplicas de serviço (executando sobre máquinas virtuais) também sofrem o *crash*. Para contornar tal limitação, apresentamos uma abordagem com replicação tolerante a faltas de *crash* para o sistema físico (anfitrião), chamada de DS-MIT (*Distributed* SMIT). Para tanto, modificamos nosso modelo de sistema, replicando o sistema anfitrião fisicamente e passando a utilizar um modelo híbrido de falhas [Correia et al. 2002]. Nesse modelo, as réplicas de serviço (VMs) estão sujeitos a falhas bizantinas enquanto que os sistemas físicos (anfitriões) estão sujeitos apenas a falhas de *crash*, diferindo assim de SMIT, onde ao anfitrião não era permitido qualquer tipo de falha, pois se tratava de um ponto singular de falha.

Além da idéia de replicação física de anfitrião, introduzimos em nosso modelo a noção de *grupos de réplicas*. Um grupo de réplicas é um conjunto formado por todas as réplicas de serviço residentes em uma mesma máquina física. Para cada anfitrião que executa SMIT (permanecendo com as mesmas premissas feitas anteriormente), existe um grupo de $2f + 1$ réplicas de serviço executando sobre ele. Cada anfitrião executa uma instância de SMIT e os anfitriões são conectados entre si através de um canal de comunicação confiável privado síncrono, por onde farão a sincronização de seus estados (*postboxes*) locais. É necessário que os clientes possuam as chaves públicas das réplicas de serviço do grupo pri-

mário e do grupo *backup*, para que possam se comunicar com estas.

A abordagem DSMIT requer, no mínimo, $f_c + 1$ máquinas físicas para tolerar f_c faltas de *crash* em nível de anfitrião. Cada máquina física deve ter pelo menos $2f_b + 1$ máquinas virtuais executando SMIT juntamente com o serviço, sendo f_b o número máximo de membros bizantinos tolerados por máquina física. De forma geral, DSMIT requer N máquinas virtuais no total, sendo que:

$$N = (2f_b + 1) \times (f_c + 1)$$

Através das modificações acima referidas, foi possível a aplicação de uma técnica de replicação tolerante a faltas de *crash* no nível do sistema anfitrião para reverter a limitação de SMIT não tolerar faltas de *crash* na máquina física (sistema anfitrião). A técnica tolerante a faltas de *crash* deve ser aplicada de modo a considerar a *postbox* e seu conteúdo como o serviço a ser replicado. Visando manter a simplicidade de SMIT, a estratégia de replicação escolhida foi a abordagem primário-*backup* (em nível de anfitrião), capaz de tolerar a falta de *crash* de uma máquina física, apresentada em [Budhiraja et al. 1993]. Outras abordagens para tolerância a faltas de *crash* poderiam ter sido igualmente aplicadas na construção de DSMIT. Apesar disso, optamos por uma implementação específica da abordagem primário-*backup*, com a única intenção de provar a viabilidade prática da aplicação de uma técnica tolerante a *crash* em nossa proposta.

Uma visão geral da arquitetura distribuída é ilustrada na Figura 4.4. A estratégia de replicação, bem como seu funcionamento e detalhes são discutidos na Seção 4.7.1.

4.7.1 Abordagem Primário-*Backup*

Conforme comentado anteriormente, optamos pela abordagem primário *backup* para demonstrar a viabilidade prática da aplicação de uma técnica de tolerância a faltas de *crash* em SMIT. Na abordagem primário-*backup* para DSMIT, as réplicas de serviço residentes na máquina primária formam o grupo de réplicas primário, que são ativas, recebendo e respondendo à requisições dos clientes. As réplicas de serviço residentes na máquina *backup* apenas executam as requisições recebidas através de sua *postbox* local, que é sincronizada à *postbox* do grupo de réplicas primário por um processo no anfitrião. Estas réplicas não recebem requisições diretamente do cliente e tampouco enviam respostas a eles. A execução das requisições lidas da *postbox* pelo grupo de réplicas *backup* tem o único intuito de manter seu estado consistente com as réplicas de serviço de outras máquinas físicas, de forma que estas possam assumir o controle das operações caso a máquina primária sofra um *crash*.

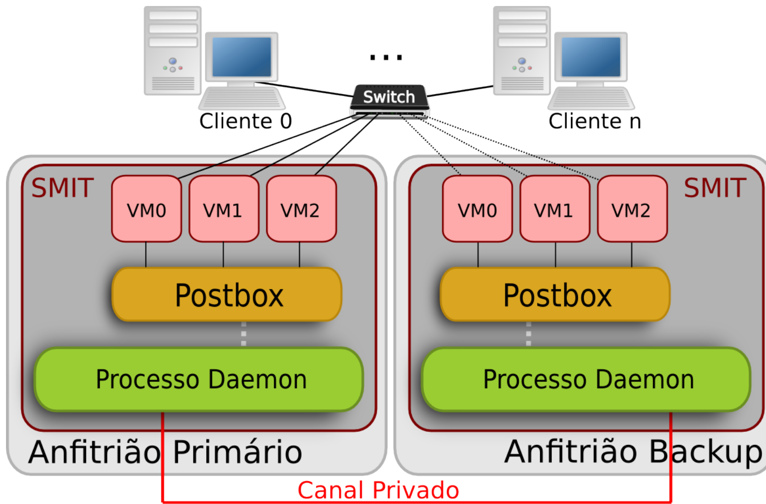


Figura 4.4: Arquitetura SMIT Distribuída (DSMIT)

Assim, para cada valor *v* escrito na *postbox* do anfitrião primário, a máquina primária deve transferir *v* para a *postbox* da máquina *backup* e aguardar por uma confirmação de recebimento. Somente após o recebimento da confirmação é que o valor passa a estar pronto para ser lido pelas máquinas virtuais da máquina primária (um mecanismo de detecção de *crash* da máquina *backup* é usado para evitar que a primária aguarde indefinidamente que uma confirmação seja enviada por uma *backup* faltosa). Assim, garantimos que a máquina *backup* estará com a sua *postbox* atualizada, mesmo que a máquina primária sofra um *crash* entre o recebimento da requisição e o envio da resposta desta para o cliente. Dessa forma, o serviço permanece disponível mesmo que a máquina primária falhe, diferentemente de SMIT.

No caso normal do protocolo, o cliente envia uma requisição somente para as réplicas do grupo primário. Tais réplicas escrevem propostas na *postbox*, seguindo o protocolo SMIT. Cada valor escrito é interceptado por um processo *daemon* executando no sistema anfitrião. Antes de efetivar a escrita do valor na *postbox* local, o *daemon* envia o valor a ser escrito, através do canal privado, para o processo *daemon* residente na máquina *backup* e aguarda por confirmação de recebimento. Somente após o recebimento da confirmação é que a escrita do valor na *postbox* do primário é efetivada. As réplicas de serviço residentes na máquina

backup fazem a leitura de sua *postbox* local e executam as requisições ali propostas sem enviá-las ao cliente. Dessa forma, garantimos a consistência através das *postboxes*, de modo que, se a máquina primária sofrer um *crash*, a máquina *backup* detecta a falha (detalhes na Seção 4.7.1.1) e assume o controle, mantendo o serviço correto e disponível aos clientes.

Caso a máquina primária sofra uma falta de *crash* em um momento após enviar a requisição req_i para a máquina *backup* e antes de enviar ao cliente a resposta para req_i, tal cliente, após ser informado de que há um novo primário, irá reenviar a requisição req_i para o novo grupo de réplicas primário.

4.7.1.1 Detecção de *Crash* do Primário

Para determinar quando a máquina primária sofre um *crash*, é necessário um mecanismo de detecção por parte da réplica *backup*. A implementação de tal mecanismo depende de premissas de sincronia no canal de comunicação que conecta as máquinas físicas.

O *daemon* da máquina primária envia periodicamente uma mensagem de notificação (vazia) para o *daemon backup*, de modo que este último possa saber que o primeiro permanece vivo. A cada τ segundos, a primária deve enviar uma mensagem para sua *backup*. Assim, o tempo entre duas mensagens de notificação é $\tau + \delta$, sendo δ a latência para que uma mensagem vazia trafegue de uma máquina para outra [Budhiraja et al. 1993]. Se uma mensagem de notificação não for recebida pelo *backup* após $\tau + \delta$ segundos, o *daemon backup* notifica suas máquinas virtuais locais para assumirem o posto das máquinas virtuais da máquina primária. Cada máquina virtual local ao *backup* então envia uma mensagem $\langle CHANGE - PRIMARY \rangle$ aos clientes, os quais então trocam o grupo de réplicas para o qual submetem suas requisições. Os clientes somente realizam tal mudança ao receberem $f + 1$ mensagens $\langle CHANGE - PRIMARY \rangle$ assinadas e oriundas de diferentes réplicas pertencentes ao mesmo grupo. Isso evita réplicas maliciosas forçando a troca de grupo de réplicas por parte do cliente quando isso não é necessário. Devido à possibilidade de ocorrerem atrasos e perdas na entrega das mensagens $\langle CHANGE - PRIMARY \rangle$ para os clientes, retransmissões são realizadas enquanto não houver confirmação de entrega.

4.8 Comparações com Trabalhos Relacionados

A Tabela 4.1 sumariza os principais aspectos a serem considerados na comparação entre a abordagem SMIT (em conjunto com DSMIT) e os trabalhos mais estreitamente relacionados, de acordo com os atributos considerados na Tabela 3.1. De acordo com a Tabela 4.1, SMIT possui

uma quantidade menor de passos de comunicação do que as outras abordagens. Quanto ao número total de VMs, SMIT apresenta uma redução, visto que em nossa abordagem o componente através do qual o consenso é obtido (a *postbox*) não exerce papel ativo no sistema. Outra característica de SMIT é que este não apresenta a figura de coordenador confiável, pois a ordem de execução das requisições é proposta pelas réplicas de serviço, sem a presença de um coordenador fixo, pois a proposta a ser considerada é a primeira proposta encontrada na *postbox*. Se tratando de uma proposta genérica, SMIT é independente de VMM e SO, bastando que seja possível implementar o modelo de sistema descrito neste capítulo nas arquiteturas de VMM e SO desejadas. O último aspecto comparado é a tolerância a faltas de *crash* no sistema anfitrião. A extensão de SMIT, chamada DS-MIT, insere suporte a faltas de *crash* no anfitrião, de forma que o serviço não se torne indisponível caso um anfitrião sofra um *crash*.

Característica	**VM-FIT**	**LBFT**	**SMIT**
Passos de comunicação	2	2	1
Total de VMs	$2f+2$	$2f+2$	$2f+1$
Componente	Ativo	Ativo	Passivo
Coordenador confiável	Sim	Sim	Não
Independência de VMM	Não	Sim	Sim
Interoperabilidade de SO	Não	Sim	Sim
Tolerante a *Crash*	Sim	Não	Sim

Tabela 4.1: Comparação entre SMIT e os trabalhos relacionados

4.9 Considerações sobre o Capítulo

Este capítulo apresentou um modelo para Replicação Máquina de Estados Tolerante a Intrusões baseado em virtualização que visa fornecer uma infra-estrutura através da qual seja possível a construção de protocolos mais simples para execução de serviços replicados. SMIT, o modelo aqui apresentado, permite a diminuição do número total de réplicas necessário de $N \geq 3f+1$ para $N \geq 2f+1$. Outro objetivo atingido por SMIT é a diminuição do custo de replicação, através da utilização de máquinas virtuais e de características específicas desses ambientes. Dessa forma, SMIT permite aumentar a resistência oferecida por sistemas baseados em uma única máquina física.

Além de SMIT, o presente capítulo apresentou uma proposta que visa contornar a maior limitação deste, que é a incapacidade de lidar com

faltas no sistema anfitrião. Para isso, foi proposta DSMIT, a versão distribuída de SMIT. O próximo capítulo apresenta a implementação de um protótipo de SMIT e DSMIT, visando demonstrar a viabilidade prática da proposta através de experimentos e da avaliação dos resultados obtidos.

Capítulo 5

Protótipo e Resultados

O presente capítulo apresenta os experimentos realizados visando verificar a viabilidade prática do modelo e protocolos propostos nas seções anteriores. Dada a importância da *postbox* para nosso modelo, a Subseção 5.1 fará uma discussão sobre alguns detalhes importantes na implementação de tal componente. Após isso, será apresentada a implementação do protocolo, bem como os experimentos realizados, juntamente com a avaliação dos resultados obtidos com o protótipo implementado.

5.1 Implementação da *Postbox*

Como já fora descrito anteriormente, a *postbox* é o principal componente de nossa arquitetura. Para que a arquitetura como um todo se torne viável, é necessário que a *postbox* seja implementada de modo a cumprir as premissas e propriedades detalhadas na Seção 4.3. Analisamos e testamos algumas alternativas existentes para a implementação da *postbox* em um VMM. Utilizar um *daemon* no sistema anfitrião para a interceptação das chamadas de sistema invocadas pelos sistemas convidados e relacionadas ao envio de mensagens pela rede é uma alternativa que oferece como principal vantagem a transparência aos sistemas convidados, ou seja, tais sistemas sequer saberiam da existência de uma *postbox* por onde se comunicam. Outra alternativa seria a utilização do VMM KVM [Qumranet 2006] em conjunto com um módulo para este que oferece a possibilidade de compartilhamento de blocos da memória principal para escrita e leitura de dados. Porém, dada a falta de maturidade do código do módulo (o qual ainda se encontra em versão experimental) e a inexistência de mecanismos para garantia das propriedades que a *postbox* necessita, optamos por não utilizar essa abordagem.

Dessa forma, a implementação de nosso protótipo é baseada no VMM *Sun VirtualBox* [Microsystems 2009]. Tal escolha se deve ao fato de esse VMM oferecer um recurso chamado de *Shared Folders*, que provê a funcionalidade de compartilhamento de arquivos do sistema de arquivos do sistema hospedeiro para os sistemas convidados (VMs), se tornando assim adequada para a implementação da *postbox*. Tal recurso provê esse compartilhamento sem a necessidade de uso da rede para o acesso às pastas compartilhadas, sendo necessário apenas um serviço específico no sistema anfitrião e um *driver* para sistema de arquivos no sistema convidado. O comando abaixo adiciona ao sistema convidado, chamado

so0, um compartilhamento chamado *share*, o qual compartilha o diretório */mnt/share/* residente no sistema anfitrião.

```
VBoxManage sharedfolder add "so0" --name "share" --hostpath "/mnt/share/"
```

Para acessar tal compartilhamento, o sistema convidado *so0*, deve montar o sistema de arquivos oferecido pelo hospedeiro, aqui chamado de *share*, em sua árvore de diretórios. Um convidado executando *GNU/Linux*, pode fazê-lo com o seguinte comando:

```
mount -t vboxsf share /path/to/mount
```

Em um convidado executando o sistema operacional *Microsoft Windows*, no qual se deseja acessar o compartilhamento *share* através da unidade *x*, o comando necessário é o seguinte:

```
net use x: \\vboxsvr\share
```

Juntamente ao mecanismo de *Shared Folders*, o VirtualBox ainda fornece um mecanismo de controle de acesso aos arquivos compartilhados, permitindo, via comandos emitidos pelo VMM, determinar quais máquinas virtuais possuem direito de escrita e leitura sobre quais arquivos compartilhados. Porém, o controle de acesso oferecido não é suficiente para que possamos garantir a propriedade *append-only*, que é crucial para a corretude da *postbox*, visto que o Virtualbox apenas oferece mecanismos para definição do modo de compartilhamento do arquivo: somente-leitura ou leitura-escrita. Desse modo, não basta apenas compartilhar um arquivo do sistema hospedeiro para as máquinas virtuais, é necessário um mecanismo mais elaborado. Para resolver tal problema, propusemos, implementamos e avaliamos duas abordagens: *n-write/1-read* (*nw1r*) e *n-write/n-read* (*nwnr*), ambas baseadas no compartilhamento de múltiplos arquivos do hospedeiro, as quais serão descritas a seguir. Após a descrição das abordagens, na Seção 5.1.3 são descritos os protocolos de coleta de lixo para cada abordagem. Após isso, na Seção 5.1.4 serão apresentadas as implementações e análise de resultados para ambas as abordagens, de forma a avaliarmos a viabilidade prática de cada uma.

5.1.1 *n-write/1-read*

Nessa abordagem, implementamos a *postbox* como um conjunto de recursos, composto por um processo *daemon* executando no sistema anfitrião, e um conjunto de $N + 1$ arquivos compartilhados, sendo N o número de réplicas no conjunto S. Assim, para cada réplica de serviço s_i (VM), é compartilhado um arquivo wf_i entre o anfitrião e s_i (a qual possui acesso de leitura e escrita sobre tal arquivo). Além de um arquivo com direito de escrita para cada VM, existe também um arquivo rf compartilhado pelo anfitrião com todas as VMs, sobre o qual somente o processo

daemon possui acesso de escrita. O papel desempenhado pelo processo *daemon* é ler, constantemente, os valores escritos pelas VMs nos arquivos do conjunto $WF = \{wf_0, wf_1, ..., wf_{N-1}\}$ e escrever os valores lidos para o arquivo *rf*. Dessa forma, o processo *daemon* determina a ordem total nas leituras das propostas feitas pelas réplicas.

O Algoritmo 5.1 apresenta o funcionamento do processo *daemon* nessa abordagem. Como podemos ver na linha 2, o *daemon* permanece lendo a partir dos arquivos de escrita das réplicas. Ao encontrar valores escritos, a função *read* retorna um vetor com os valores obtidos dos arquivos das réplicas. De posse desses valores, o *daemon* os escreve no arquivo de somente-leitura das réplicas, *rf*, fazendo assim, com que as escritas realizadas pelas réplicas possam ser visualizadas em ordem total pelas mesmas. A Figura 5.1 ilustra essa abordagem.

Algoritmo 5.1 *Daemon* de Controle da *Postbox* - *nw1r*

```
1: while true do
2:     values ← read(WF)
3:     for all v in values do
4:         rf.append(v)
5:     end for
6: end while
```

5.1.2 *n-write/n-read*

A segunda abordagem visa prover maior isolamento entre as VMs, visto que na primeira delas, as réplicas compartilham a utilização de um arquivo de somente-leitura. Tal compartilhamento pode representar um aumento considerável na complexidade do protocolo de coleta de lixo, como será descrito na Seção 5.1.3. Para prover tal isolamento, além de um arquivo com permissão de escrita para cada VM (grupo WF), é necessário mais um arquivo com permissão de leitura-escrita para cada réplica (grupo $RF = \{rf_0, rf_1, ..., rf_{N-1}\}$). Assim, cada VM possui seu par de arquivos privado. De forma a prover a visão homogênea requerida no modelo de sistema, o processo *daemon* deve, ao realizar a leitura de um valor em um dos elementos de WF, escrever tal valor de forma atômica para todos os elementos de RF. Ou seja, um valor v, lido de um arquivo wf_i, deve ser escrito atomicamente pelo *daemon* em todos os elementos de RF, mantendo assim a ordem total nos arquivos compartilhados. O Algoritmo 5.2 apresenta o funcionamento da abordagem *nwnr*. Assim como no Algoritmo 5.1, a operação *read* (linha 2) retorna um vetor de valores. Após isso, cada valor retornado é escrito, de forma atômica, para

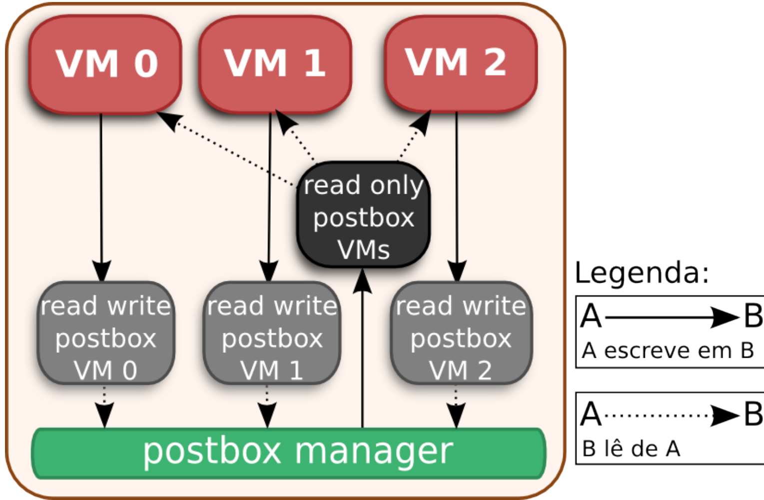


Figura 5.1: Abordagem *Multi-write/Single-read*

todos os arquivos pertencentes ao conjunto RF. Assim, uma réplica r_i pode realizar as leituras em seu arquivo individual rf_i, sendo que o conteúdo do arquivo rf_j de qualquer outra réplica r_j é idêntico ao conteúdo de rf_i. A Figura 5.2 ilustra essa abordagem.

Algoritmo 5.2 *Daemon* de Controle da *Postbox* - *nwnr*

```
1: while true do
2:     values ← read(WF)
3:     for all v in values do
4:         for all rf in RF do
5:             rf.append(v) {esse laço deve ser atômico}
6:         end for
7:     end for
8: end while
```

5.1.3 Protocolo de Coleta de Lixo

A *postbox* é um componente de memória finita. Assim sendo, é necessário um mecanismo para evitar que o tamanho limite desse componente seja ultrapassado, o que acarretaria na parada do protocolo como um todo para a máquina física. O mecanismo de coleta de lixo é necessário

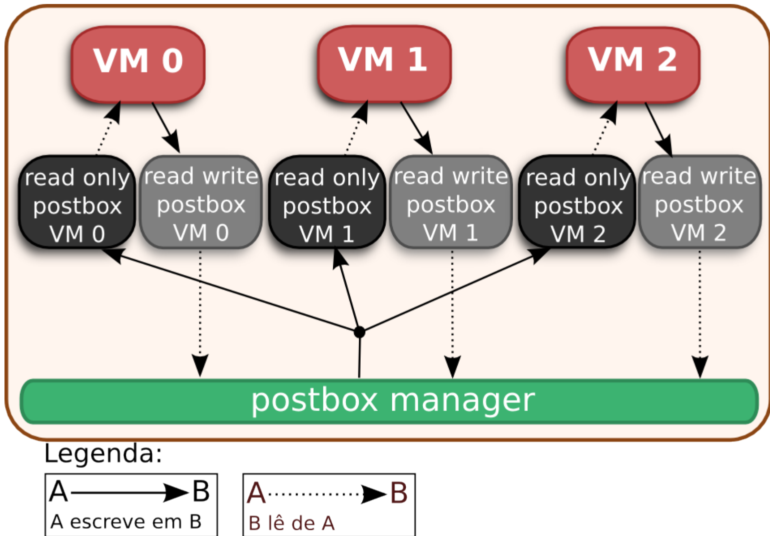


Figura 5.2: Abordagem *Multi-write/Multi-read*

para que seja possível remover da *postbox* as entradas que não são mais necessárias pelas réplicas, de modo a evitar que os arquivos compartilhados cresçam indefinidamente. Ambas as abordagens descritas acima para a implementação da *postbox* são bastante semelhantes, porém elas apresentam uma diferença fundamental: a abordagem *nw1r* não isola completamente os arquivos das VMs que representam a *postbox*, enquanto que a abordagem *nwnr* o faz. Esse detalhe possui um grande impacto na implementação prática, principalmente na construção do algoritmo de coleta de lixo, conforme será descrito nos parágrafos abaixo.

nw1r Na primeira abordagem, existe apenas um arquivo sobre o qual as réplicas de serviço (VMs) realizam suas operações de leitura. Dessa forma, apontar quando todas as réplicas já processaram determinada mensagem e não necessitam mais que tal valor permaneça armazenado no arquivo se torna uma tarefa complicada em um cenário bizantino. Neste caso, uma réplica maliciosa poderia simplesmente se omitir da tarefa de notificar a não-necessidade de um valor, podendo causar facilmente um *overflow* da *postbox*. Para resolver esse problema, uma hipótese temporal deve ser feita sobre o comportamento das réplicas, ou seja, se uma réplica, após determinado tempo pré-definido não apontar os valores que não lhe

são mais necessários, tal réplica passa a ser considerada faltosa. Nesse caso, as entradas da *postbox* apontadas como não mais necessárias pelas réplicas restantes e comuns a todas elas serão removidas da *postbox*. Ou seja, a cada intervalo de tempo, um conjunto de entradas será removida e considera-se que as réplicas corretas já terão processado estas. As réplicas que não manifestarem a não-necessidade destas entradas são consideradas faltosas. Porém, em um ambiente de execução assíncrono, a utilização de tal hipótese se torna um problema, visto que máquinas virtuais que sofram perda de desempenho, mesmo que temporárias, serão consideradas faltosas.

nwnr Na segunda abordagem, a responsabilidade da remoção de entradas não-necessárias fica a cargo das próprias VMs. Como cada réplica possui seu par de arquivos de leitura e escrita, basta que um protocolo simples de coleta de lixo seja implementado nos sistemas convidados. Não é necessária comunicação alguma com outras entidades, uma réplica de serviço pode fazer a limpeza de entradas em seus arquivos assim que julgá-las desnecessárias. Assim, basta que as VMs escrevam os valores nos arquivos de forma circular, sobrescrevendo valores desnecessários. Caso uma VM maliciosa se omita da limpeza de seus *buffers*, esta estará prejudicando apenas a sua própria execução, não afetando as réplicas corretas, pois cada VM possui seus arquivos de comunicação, que representam a *postbox*, isolados. Para que isso seja garantido, deve ser implementado um mecanismo que possibilite o controle do tamanho máximo que os arquivos compartilhados entre anfitrião e convidados podem atingir. Uma implementação já existente de tal mecanismo é o sistema de quotas de disco, disponível em sistemas Linux.

5.1.4 Implementação e Resultados

Ambas as abordagens, *nw1r* e *nwnr* foram implementadas e avaliadas através de experimentos, os quais serão descritos nessa seção. A implementação se deu seguindo exatamente as descrições feitas nas seções 5.1.1 e 5.1.2, utilizando o seguinte ambiente:

- **Sistema Hospedeiro**: processador Core 2 Quad, com 8 GB de memória principal, executando o sistema operacional Debian GNU/-Linux 5.0.
    - **VMM**: Sun VirtualBox 2.2.4.
    - **3 VMs**: cada uma com 1 GB de RAM, executando Ubuntu GNU/Linux 9.04 Server.

O experimento consistiu de uma VM escrevendo valores de tamanho variável na *postbox*, e outras VMs lendo tais valores, de modo a existir concorrência no acesso a *postbox* durante os testes. Durante a realização dessas operações, os tempos de acesso foram medidos, tornando possível o cálculo da média dos tempos de acesso nas operações na *postbox*. O tamanho dos valores varia de 32 *bytes* até 4096 *bytes*. O processo *daemon* de cada abordagem foi implementado na linguagem Python [Python Software Foundation 2010], utilizando a versão 2.6.2 do interpretador. O Código 5.1 apresenta uma versão simplificada do código-fonte da classe que compõe o processo *daemon*, chamado de *postbox_manager.py*. Ambas as abordagens podem ser implementadas utilizando-se a classe apresentada. Da forma que se encontra, o código-fonte representa a abordagem *nwnr*. Como se pode ver, as variáveis *RPATH* e *WPATH* armazenam os caminhos para os arquivos compartilhados com os sistemas virtuais, sendo um diretório para cada sistema convidado (*share0* para o primeiro convidado, *share1* para o segundo e *share2* para o terceiro). Para a abordagem *nwnr*, ambos os arquivos de cada convidado podem residir no mesmo diretório, visto que em ambos, o sistema convidado possui acesso de escrita. O Código 5.1 necessita de leves modificações para que se comporte seguindo a abordagem *nw1r*, sendo necessário apenas modificar a variável *WPATH*, que representa os arquivos sobre os quais as réplicas realizam a leitura de valores, representando o conteúdo da *postbox* em si. Ao invés de múltiplos arquivos, basta configurá-la para representar apenas um arquivo, que seja acessível em modo somente-leitura por todas as réplicas, representando assim o arquivo rf referido anteriormente na Seção 5.1.1.

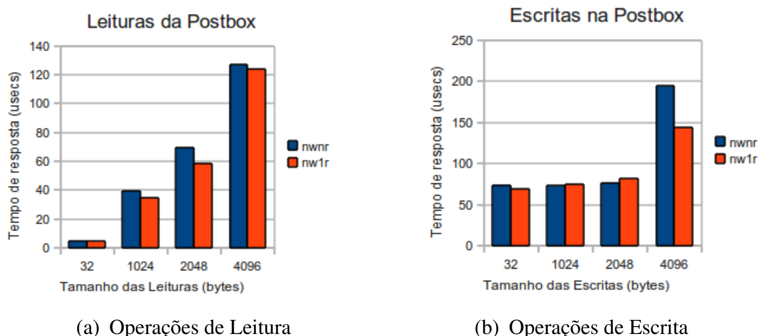
As Figuras 5.3(a) e 5.3(b) apresentam os resultados obtidos nas duas abordagens para operações de leitura e de escrita. Em ambos os gráficos, os resultados foram obtidos através da média aritmética de 10000 execuções para cada tamanho e estão apresentados em *microsegundos*. De modo geral, em ambas as operações, a abordagem *nw1r* leva vantagem, apresentando tempos de resposta menores que a abordagem *nwnr*. Os resultados obtidos em ambas as abordagens foram satisfatórios, visto que na maior média de tempos de resposta obtida (em leituras de mensagens de 4kB), o valor fica abaixo dos 200 microsegundos, sendo que esse valor é consideravelmente maior do que os resultados obtidos para outros tamanhos de mensagens.

Apesar de a abordagem *nw1r* obter uma leve vantagem nos resultados obtidos com os experimentos práticos, a sua implementação requer uma hipótese temporal complexa de ser feita, o que complica bastante sua viabilidade prática em um cenário real, onde o comportamento dos processos é assíncrono. Dessa forma, optamos por utilizar como *postbox*

Código 5.1: Código Python para escrita na *Postbox*

```
import os
import sys
from threading import Thread, Lock
from comm.postbox import postbox
from utils.my_logger import log

# conjuntos RF e WF
RPATH = ['/mnt/share0/wpbox','/mnt/share1/wpbox','/mnt/share2/wpbox']
WPATH = ['/mnt/share0/rpbox','/mnt/share1/rpbox','/mnt/share1/rpbox']

class postbox_manager():

  def __init__(self):
    log.info('started␣postbox␣manager')
    self.__lock = Lock()
    self.wpb = [postbox(p, 'r+') for p in WPATH]
    self.rpb = [self.pb_create(p) for p in RPATH if os.access(p, os.R_OK)]

  def pb_create(self, p):
    rpbox = postbox(p, 'r+')
    th = Thread(target = self.read_postbox, args = (rpbox, ))
    return rpbox, th

  def read_postbox(self, rpbox):
    while True:
      readed = rpbox.receive()
      log.debug('readed:␣%s' % ''.join(readed.split('\n')))
      self.__lock.acquire()
      try:
        for w in self.wpb:
          w.send(readed)
      finally:
        self.__lock.release()

  def start_readers(self):
    for pbox in self.rpb:
      pbox[1].start()

if __name__ == '__main__':
  p = postbox_manager()
  p.start_readers()
```

a implementação *nwnr*, que assegura todas as propriedades estabelecidas para a *postbox* na Seção 4.3. Além disso, essa abordagem apresenta um protocolo de coleta de lixo simplificado, sem a necessidade de definição de hipóteses temporais sobre o comportamento dos processos em execução. Dessa forma, na implementação apresentada nas próximas seções, a implementação da *postbox* utilizada foi a abordagem *nwnr*.



(a) Operações de Leitura (b) Operações de Escrita

Figura 5.3: Comparação de resultados entre abordagens *nw1r* e *nwnr*

5.2 Camada de Replicação

A camada responsável pela implementação do lado servidor do protocolo proposto na Seção 4.5 é a Camada de Replicação. Essa camada implementa o protocolo necessário para que as réplicas entrem em consenso sobre a ordem de execução das requisições emitidas pelos clientes. Além disso, as funções de acesso ao componente *postbox* serão descritas nessa seção, visto que fazem parte da camada de replicação, funcionando como base para a operação dos protocolos. O Código 5.2 apresenta uma versão simplificada da implementação Python dos Algoritmos 4.2 (método `reqs_listener()`) e 4.3 (método `read_postbox()`).

O Código 5.3 apresenta uma implementação simplificada da classe que representa a interface de acesso das réplicas ao componente *postbox*. Através de uma instância dessa classe, as réplicas de serviço realizam as operações de leitura e escrita sobre a *postbox*, necessárias para a comunicação entre réplicas para concretização do protocolo de consenso apresentado anteriormente para SMIT.

5.3 Cliente

O lado cliente do protocolo, o qual emite requisições para as réplicas e aguarda pelo recebimento de $f+1$ respostas mutuamente consistentes foi implementado em Python, seguindo a descrição do Algoritmo 4.1. O Código 5.4 apresenta uma versão simplificada do código Python do método utilizado pelo cliente para envio da requisição (`send_request()`).

Visando diminuir o custo para assinar as mensagens a serem enviadas, a assinatura é feita apenas sobre o resumo criptográfico da mensagem

Código 5.2: Código Python da implementação da Camada de Replicação

```
def reqs_listener(self):
    while True:
        serial_msg = self.receiver.receive()
        try:
            raw_req, cert = serial_msg.split(defs.MSG_SEP)
        except ValueError:
            self.log.critical('bad␣formatted␣message␣from␣client')
            continue
        else:
            req = unserial(raw_req)
            d = sha1(raw_req).hexdigest()
            if d not in self.processed and \
              self.clients[req[1]][3].Verify(d, cert):
                self.wPostbox.append(serial_msg)

def read_postbox(self):
    while True:
        readed = self.rPostbox.read()
        try:
            raw_msg, cert = readed.split(defs.MSG_SEP)
        except ValueError:
            self.log.critical('bad␣formatted␣message␣from␣postbox')
            continue
        else:
            # verifica assinatura e executa raw_msg, caso correta
            self.handle_proposal(raw_msg, cert)
```

Código 5.3: Interface de acesso à *Postbox* - lado servidor

```
class postbox:
    def __init__(self, path, mode):
        self.data = open(path, mode)

    def read(self):
        while True:
            line = self.data.readline()
            entries = line.split(PBOX_SEP)
            # caso leia uma entrada incompleta, volta atrás
            if len(entries) != 2:
                self.data.seek(-(len(line)), os.SEEK_CUR)
                continue
            if len(entries[0].strip()) > 0:
                return entries[0]

    def append(self, msg):
        self.data.write(msg + PBOX_SEP + '\n')
        self.data.flush()
```

(SHA-1), visto que essa possui um tamanho reduzido e fixo (40 *bytes*). O método `send_request()`, além do envio da requisição para as réplicas, faz uma chamada ao método `collect_replies()`, o qual é responsável por aguardar e acumular as mensagens recebidas das réplicas.

Como se pode ver no Código 5.4, o cliente permanece aguardando por mensagens até que receba $f + 1$ mensagens iguais relativas ao *timestamp* da requisição pela qual está esperando. A variável `self.max_faulty` é configurada com o valor máximo tolerado de réplicas maliciosas.

Código 5.4: Código Python da implementação do Cliente

```
def send_request(self, req):
    s_req = cPickle.dumps(req)  # serialização
    digest = hashlib.sha1(s_req).hexdigest()
    s_req = s_req + defs.MSG_SEP + self.signer.Sign(digest)
    self.mcast.send(s_req)
    reply = self.collect_replies(req[2])
    return reply

def collect_replies(self, ts):
    replies = {}
    msg_sha1 = None
    while True:
        # self.channels: lista de canais de comunicação
        irdy, ordy, err = select.select(self.channels, [], [], 0)
        for channel in irdy:
            serial_msg = self.REPLICAS[self.sock_owners[channel]][4].recv()
            try:
                # verifica integridade, autenticidade e timestamp
                reply = self.check_msg(serial_msg, ts)
            except InvalidTimestamp as e:
                self.log.warning('Old␣ts[ts:%s][old:%s]' % (e.ts, e.old))
                continue
            except BadSignature as e:
                self.log.warning('Bad␣signature␣from␣%s' % e.source)
                continue
            else:
                self.log.debug('Reply␣from␣%s␣is␣OK!' % reply[1])
                msg_id = reply[0] + str(reply[2])
                msg_sha1 = hashlib.sha1(msg_id).hexdigest()
                replies[hashlib.sha1(serial_msg).hexdigest()] = msg_sha1
                # se tiver f+1 mensagens iguais, retorna
                if replies.values().count(msg_sha1) > self.max_faulty:
                    return reply
```

5.4 Avaliação de Desempenho

A presente seção apresenta detalhes sobre o protótipo implementado de SMIT descrito nas seções anteriores. A arquitetura proposta neste trabalho foi implementada em Python, utilizando a versão 2.6.2 do interpretador. Para testá-la, construímos o seguinte ambiente de testes:

- **Sistema Hospedeiro Primário**: processador Core 2 Quad, com 8 GB de memória principal, executando o sistema operacional Debian GNU/Linux 5.0.
    - **VMM**: Sun VirtualBox 2.2.4.

  - ***Postbox***: abordagem *n-write/n-read*.
  - **3 VMs**: cada uma com 1 GB de RAM, executando Ubuntu GNU/Linux 9.04 Server.

- **Hospedeiro Backup (se aplica apenas ao cenário DSMIT)**: processador Core 2 Duo, com 1.5 GB de memória principal, executando Ubuntu 9.04 Desktop (demais detalhes são idênticos ao Hospedeiro Primário, exceto que as VMs foram configuradas com 256 MB de memória apenas).

- **Clientes**: executando em um processador Core 2 Duo, com 2 GB de RAM, executando o sistema operacional Ubuntu GNU/Linux 9.04 Desktop.

- **Canal de Comunicação**: *switch* de rede de 100 Mb/s.

- **Técnica Criptográfica**: RSA, com chaves de 1024 bits.

Nossa avaliação de performance consiste em uma análise do tempo de resposta, de acordo com vários cenários de comparação, uma análise do tempo de resposta variando o método criptográfico utilizado, bem como o tamanho da chave, e em uma análise do *throughput* obtido pelo sistema quando da existência de acessos simultâneos de clientes. Além disso, avaliamos o tempo de recuperação para que a máquina *backup* assuma o controle das operações em DSMIT, ou seja, verificamos o tempo que o sistema permanece fora de alcance.

5.4.1 Tempo de Resposta

Para avaliar o desempenho do protótipo, utilizamos *micro benchmarks*, metodologia de avaliação semelhante ao usado no PBFT [Castro e Liskov 2002]. Nesses *micro benchmarks*, foram executadas operações nulas nas réplicas de serviço, variando os tamanhos da requisição e da resposta de 0 kB a 4 kB. A nomenclatura das operações se dá da seguinte forma: *ab*, onde *a* é o tamanho da requisição enviada e *b* é o tamanho da resposta para a requisição, ambos representados em kB. Por exemplo, uma operação chamada *04* possui uma mensagem de 0 kB como requisição e obtém como resposta uma mensagem de 4 kB.

Assim, avaliamos a execução de nosso protótipo através de medidas do tempo de resposta para certos tipos de operação. O tempo de resposta foi medido pelos clientes, lendo seu relógio local imediatamente antes de enviar uma requisição e imediatamente após receber a resposta definitiva para tal requisição. Para cada tipo de operação testada (variando o tamanho da requisição e da resposta), realizamos 10000 repetições, de

forma que os valores apresentados na Figura 5.4 representam a média aritmética dos valores obtidos, representados em milisegundos. Criamos seis cenários diferentes para a avaliação de nosso protótipo:

1. **SMIT**: o protótipo, com 3 VMs executando o protocolo proposto.
2. **SMIT-f**: mesmo que *SMIT*, porém com uma réplica faltosa.
3. **SMIT-*socket***: SMIT, com postbox implementada através de um *daemon* no anfitrião, conectado aos convidados através de *sockets*.
4. **DSMIT**: a versão distribuída do protocolo, utilizando duas máquinas físicas com 3 VMs em cada uma destas.
5. ***Single***: o serviço não-replicado.
6. ***Single*-VM**: o serviço não-replicado, executando sobre uma máquina virtual.

Como podemos ver na Figura 5.4, nosso protótipo gera um aumento substancial no tempo de resposta das operações. Tais resultados já eram esperados, visto que nosso protótipo oferece segurança à execução de uma requisição, necessitando as operações necessárias para a execução segura em um ambiente replicado, diferentemente dos ambientes não replicados (*Single* e *Single-VM*). É importante perceber que o tamanho da resposta de uma operação possui efeito maior sobre o tempo de resposta do que o tamanho da requisição, pois o cliente precisa receber e votar sobre ao menos $f + 1$ respostas para completar a execução. Assim, operações que requerem grandes quantidades de dados como suas respostas levam a tempos de resposta maiores. Assim como no PBFT [Castro e Liskov 2002], isso pode ser otimizado fazendo com que a cada requisição enviada, o cliente eleja apenas uma das réplicas para lhe enviar a mensagem de resposta por inteiro, enquanto as outras lhe enviam apenas o resumo criptográfico destas. Caso os resumos recebidos não sejam idênticos ao resumo calculado sobre a mensagem inteira, o cliente solicita que todas as réplicas lhe enviem a mensagem inteira. Outro fato que merece atenção é o fato de o cenário *Single-VM* apresentar um aumento considerável no tempo de resposta, se comparado ao caso mais simples (*Single*), mostrando que o uso de um ambiente virtualizado acarreta em um maior *overhead* para completar uma operação.

Com o único intuito de avaliar o impacto que a nossa implementação da *postbox* em arquivos gera no tempo de resposta, implementamos um protótipo da *postbox* que consiste de um *daemon* residente no anfitrião, o qual mantém um estado interno e oferece *sockets* para que os

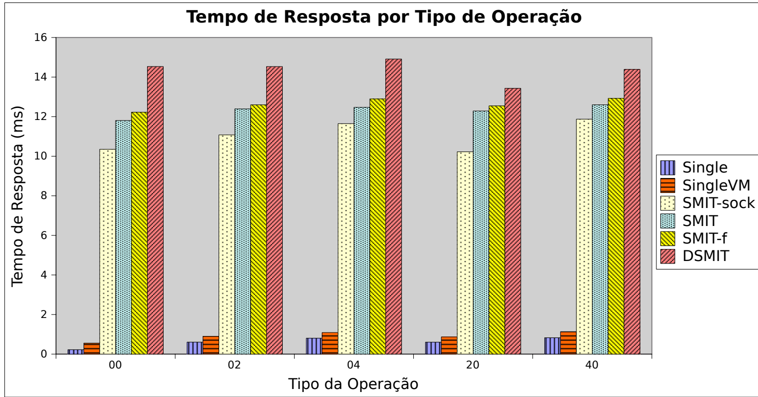


Figura 5.4: Tempo de Resposta por Operação

processos convidados leiam e escrevam valores. Tal cenário é apresentado na Figura 5.4 como SMIT-*socket* e, através dos resultados obtidos em sua execução, foi possível verificar que nossa implementação da *postbox* não apresenta perdas de desempenho tão grandes quando comparado a uma implementação com *sockets*, conforme nos mostra a Figura 5.4. É importante perceber que SMIT-*socket* necessita que os convidados sejam conectados através da rede ao sistema anfitrião. Assim sendo, um convidado malicioso poderia realizar ataques sobre o anfitrião através da rede. Para evitar que um convidado malicioso obtenha sucesso ao atacar o anfitrião através dos *sockets* utilizados em SMIT-*socket*, é possível aplicar um filtro de pacotes nas conexões entre anfitrião e convidados, permitindo que apenas pacotes de aplicação sejam transferidos por tal conexão. Na abordagem utilizando os arquivos compartilhados, não há conexão pela rede entre as réplicas de serviço e o sistema anfitrião.

O cenário SMIT-f apresenta os resultados da execução do protótipo com a presença de uma réplica faltosa. Como podemos ver na Figura 5.4, o tempo de resposta cresceu levemente no cenário em que há uma réplica faltosa. Tal aumento pode ser justificado pelo recebimento de respostas incorretas por parte do cliente. Enquanto que no cenário SMIT (onde todas as respostas são corretas) o cliente aguarda por apenas $f + 1$ mensagens para definir a resposta correta, em SMIT-f pode ser necessário que o cliente aguarde por mais de $f + 1$ mensagens, visto que no conjunto das primeiras $f + 1$ mensagens recebidas pode estar contida uma mensagem incorreta enviada pela réplica faltosa, o que não ocorre no cenário SMIT.

O cenário DSMIT apresenta os resultados obtidos na execução da abordagem distribuída de SMIT, DSMIT (apresentada na Seção 4.7). Esse cenário apresentou um tempo de resposta de, em média, 3 milisegundos a mais que o cenário SMIT. Tal acréscimo se justifica pela necessidade de, a cada requisição recebida, o *daemon* primário transferir os valores escritos na *postbox* para o *daemon backup* e ainda aguardar por uma confirmação de recebimento, conforme descrito na Seção 4.7.

5.4.1.1 Diversidade de Software

Também foram realizados experimentos para verificar a viabilidade prática de nossa implementação em conjunto com técnicas de diversidade de software. Para isso, implementamos os protocolos em três sistemas operacionais distintos:

- *Microsoft Windows XP Professional Service Pack* 2.
- *SunOS OpenSolaris 5.11*.
- *Ubuntu GNU/Linux 9.04 Server*.

Em cada um dos SOs listado acima, utilizamos uma versão específica do interpretador Python. Todos os SOs foram executados sobre o VMM VirtualBox, utilizando como *postbox* nossa versão baseada no mecanismo de pastas compartilhadas fornecido por este VMM. O código de implementação dos protocolos necessitou de algumas mudanças no processo de porte para os SOs acima listados. A mais significativa diz respeito a implementação da *postbox* para o SO Microsoft Windows, para a qual, após cada leitura realizada foi necessário fechar os descritores de arquivo relacionadas a *postbox* para que o conteúdo destes fosse atualizado.

Os resultados obtidos com os experimentos são apresentados na Figura 5.5. Como podemos ver, os tempos de respostas das operações cuja mensagem de requisição possui tamanho 0 obtiveram resultados muito semelhantes aos resultados obtidos na abordagem SMIT (Figura 5.4). As operações cujo tamanho da requisição é diferente de 0 apresentaram um crescimento no tempo de resposta em comparação com a abordagem sem diversidade. Isto resulta das modificações necessárias no código da *postbox*, bem como de possíveis diferenças nos drivers de acesso às pastas compartilhadas para os diferentes sistemas operacionais.

5.4.2 Mecanismos Criptográficos

Sabendo que as operações criptográficas realizadas são responsáveis por grande parte do processamento realizado tanto nos clientes

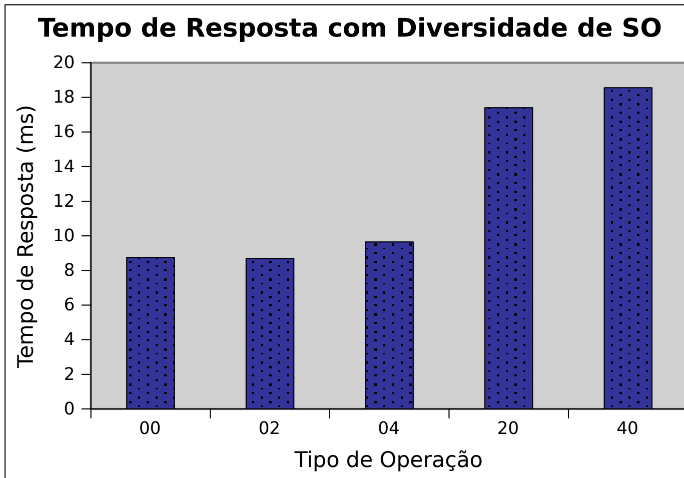


Figura 5.5: Tempo de Resposta por Operação - Diversidade de SO

quanto nas réplicas, elaboramos um experimento para verificar o impacto prático causado pelo uso de tais mecanismos. Assim, modificamos o protótipo para verificar o tempo de resposta obtido na realização das operações utilizando métodos criptográficos diferentes. A metodologia utilizada foi a mesma dos resultados apresentados na Figura 5.4, porém, foram criados diferentes cenários, utilizando diferentes técnicas criptográficas. Além da técnica RSA com chaves de 1024 *bits*, utilizada nos testes de SMIT da Figura 5.4, utilizamos RSA [Rivest, Shamir e Adleman 1978] com chaves de 512 e 768 *bits*, bem como AES [FIPS 2001] com chaves de 256 *bits* e DSA [PUB 1994] com chaves de 1024 *bits*.

A Figura 5.6 apresenta os resultados obtidos. Como era esperado, as execuções utilizando chaves menores, como AES 256 e RSA 512 obtiveram os melhores resultados. O mecanismo DSA com chaves de 1024 *bits* obteve o pior resultado. Apesar de os resultados do RSA 1024 serem piores que os resultados obtidos com chaves menores, utilizamos esse mecanismo como padrão em nosso protótipo, visto que chaves maiores oferecem maior resistência a ataques sobre o mecanismo criptográfico [Cavallar et al. 2000]. Além disso, a utilização da técnica RSA 1024 em nosso protótipo apresentou resultados consideravelmente melhores quando comparado a utilização da técnica DSA 1024, como mostra a Figura 5.6.

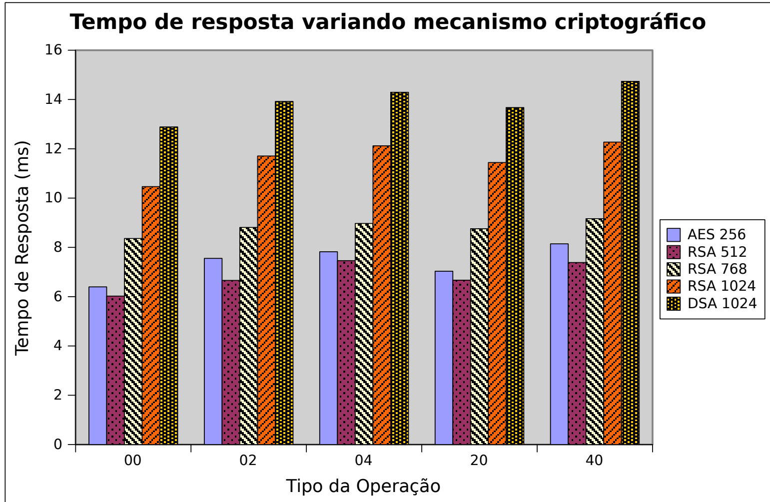


Figura 5.6: Tempo de resposta por operação variando mecanismo criptográfico

5.4.3 *Throughput*

Para avaliar o *throughput* de nosso protótipo, executamos operações 00 (onde tanto o argumento quanto o resultado possuem tamanho 0 kB), variando o número de clientes simultâneos solicitando operações, entre 0 e 15 clientes. A Figura 5.7 apresenta os resultados obtidos nesse experimento. Como podemos ver, o *throughput* cresce muito pouco de acordo com o aumento do número de clientes, estabilizando em cerca de 165 operações por segundo quando da presença de 9 ou mais clientes simultâneos. Tal estagnação pode ser explicada pela necessidade de, a cada operação a ser realizada, acessar um componente centralizado (a *postbox*), não sendo possível a realização de execuções paralelas, dada tal necessidade. Otimizações no acesso a tal componente podem gerar melhorias nos resultados no *throughput*.

5.4.4 Tempo de Recuperação - DSMIT

Para avaliar o impacto que uma falta de *crash* no *host* primário gera para os clientes, foram realizadas execuções do ambiente DSMIT, no qual o *host* primário sofre uma falta de *crash* em meio à execução de requisições dos clientes. O tempo médio obtido para recuperação foi $225,95$ ms, sendo $\tau + \delta = 200$ ms (conforme descrito na Seção 4.7.1.1). O tempo de

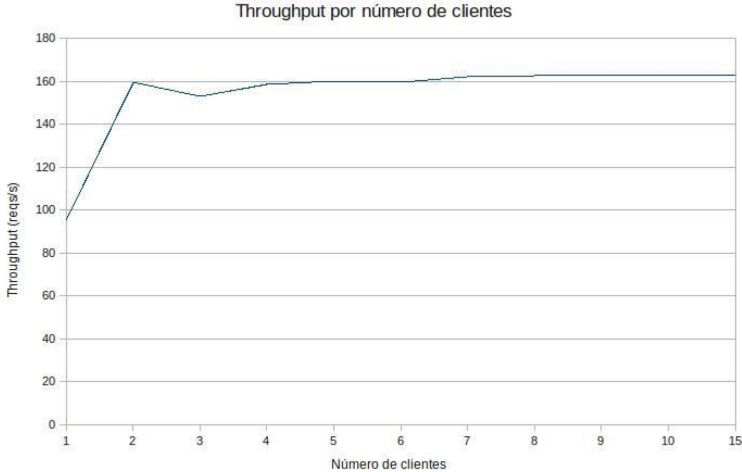


Figura 5.7: *Throughput* verificado através da variação na carga

recuperação representa o tempo necessário para que o hospedeiro *backup* assuma o controle e responda as requisições pendentes dos clientes. Tal valor foi medido pelo cliente, como o tempo de resposta para a requisição cuja execução ocorreu em meio a falha do *host* primário.

5.5 Considerações Finais

A presente seção descreveu em detalhes os protótipos de SMIT e DSMIT implementados. Através desses protótipos, pudemos verificar a viabilidade prática do modelo, bem como tomar decisões baseadas em resultados de experimentos práticos.

Apresentamos a implementação e avaliação de duas abordagens distintas para implementação da *postbox*, de forma que pudéssemos redirecionar nossos esforços para uma abordagem apenas. Através da análise dos resultados, pudemos escolher uma abordagem em particular (*nwnr*) para utilização na implementação de SMIT. Para tal escolha, nos baseamos não apenas nos resultados obtidos nos experimentos práticos, mas também na viabilidade de uma implementação prática para nossa proposta.

Após definida a abordagem utilizada para a *postbox*, pudemos nos concentrar na implementação e avaliação de SMIT. Tal avaliação se deu através da utilização de *micro-benchmarks*, com o intuito de verificar o

desempenho obtido por SMIT sem que os resultados de execução fossem influenciados pela carga gerada pela utilização de uma aplicação específica. Assim avaliamos o desempenho com relação ao tempo de resposta para a execução de uma operação, possibilitando a verificação do impacto prático causado pela replicação de um serviço qualquer através de nossa arquitetura. Além disso, realizamos experimentos para verificação do *throughput* máximo que nosso protótipo é capaz de manipular. Tal avaliação é muito importante, pois permitiu a mensuração do desempenho de SMIT em situações onde a carga de trabalho é aumentada.

Embora em geral os resultados obtidos tenham sido satisfatórios, eles nos mostraram a necessidade de otimizações na implementação, de modo a melhorar pontos específicos da execução, como o *throughput*, por exemplo. Para isso, seriam necessárias novas implementações de determinados componentes, como a *postbox*, possivelmente utilizando novas tecnologias de virtualização, os quais podem ser redirecionados para os esforços dos trabalhos futuros.

Capítulo 6

Conclusões e Perspectivas Futuras

O presente documento apresentou os detalhes da proposta de uma arquitetura para tolerância a intrusões baseada em máquinas virtuais e no compartilhamento de memória entre estas. Diferentemente de outros trabalhos anteriores [Castro e Liskov 2002, Yin et al. 2003, Kotla et al. 2007], nossa abordagem é voltada apenas para ambientes virtualizados, utilizando características específicas desses ambientes. O aproveitamento de tais características permitiu a proposta de um protocolo de consenso bizantino para a abordagem Replicação Máquina de Estados mais simplificado do que propostas anteriores [Castro e Liskov 2002, Yin et al. 2003, Kotla et al. 2007]. Consideramos tal simplicidade um fator fundamental para o desenvolvimento dessa proposta, visto que protocolos complexos levam a implementações complexas, aumentando a possibilidade de ocorrência de faltas de implementação. A idéia básica aqui seguida foi transferir parte da complexidade dos protocolos para a arquitetura.

Assim, propusemos uma arquitetura, fundamentada em um modelo de sistema apresentado nesse documento, capaz de replicar serviços e executar protocolos para que esses serviços sigam estritamente as propriedades definidas para corretude na abordagem RME. Tal arquitetura apresenta como principal vantagem o aumento da resistência a ataques em uma máquina física. Através da replicação em máquinas virtuais residentes em uma mesma máquina física, tornamos tal máquina tolerante a intrusões, no sentido de que a um atacante não bastará atacar apenas um sistema para obter o controle sobre o serviço.

O trabalho aqui apresentado atingiu todos os objetivos estipulados em sua proposta inicial (apresentados na Seção 1.4). Através da arquitetura tolerante a intrusões aqui apresentada, pudemos diminuir o número total de réplicas necessárias para tolerar f membros faltosos de $N \geq 3f + 1$ para $N \geq 2f + 1$. Além disso, o custo de replicação foi diminuído com relação a abordagens anteriores, através da replicação de sistemas através de máquinas virtuais.

Um levantamento bibliográfico foi realizado de modo a formar a base teórica necessária para a proposta de SMIT, sendo composto pelo estudo detalhado de conceitos básicos relacionados à Confiança no Funcionamento, Sistemas Distribuídos e Virtualização, os quais formam a base para o trabalho aqui proposto, sendo apresentados no Capítulo 2. Além de tais conceitos, o estudo de trabalhos relacionados, apresentado no Ca-

pítulo 3, também faz parte dos objetivos consolidados. Através desse estudo, foi possível um melhor entendimento de protocolos para tolerância a faltas bizantinas/intrusões, um refinamento de nossa proposta, bem como comparações de tais abordagens com o nosso trabalho.

Para tornar possível a execução correta de serviços em nossa arquitetura, propusemos um protocolo de consenso baseado em memória compartilhada entre VMs (apresentado no Capítulo 4). Através da utilização de tal componente, foi possível a simplificação do protocolo, diminuindo o número de passos de comunicação e trocas de mensagens entre réplicas para concretização do consenso. Visando garantir a corretude do protocolo proposto, elaboramos suas provas de correção, as quais foram apresentadas na Seção 4.6. Além do protocolo de consenso, a descrição de um protocolo auxiliar de coleta de lixo na memória compartilhada também pode ser incluído dentre os objetivos atingidos.

Além da proposta de SMIT, que consiste de uma abordagem para TI baseada em uma única máquina física, propusemos uma extensão a tal proposta, baseada na replicação de máquinas físicas, chamada de DSMIT. Tal proposta se fez necessária para contornar a limitação de SMIT não tolerar faltas ocorridas no sistema anfitrião. Assim como para a proposta de SMIT, o objetivo de simplificação dos protocolos foi mantido para DSMIT. Para isso, nos utilizamos de um modelo híbrido de falhas, o que possibilitou a utilização de diferentes componentes sujeitos a diferentes tipos de falhas.

Também como parte dos objetivos atingidos, implementamos e avaliamos o desempenho de um protótipo de SMIT, de forma a verificar a viabilidade prática do modelo apresentado. Através de diversos experimentos, foi possível embasar a tomada de decisões de projeto necessárias para o desenvolvimento. Além disso, pudemos verificar pontos fracos da implementação, identificando potenciais trabalhos futuros.

A utilização de um componente de memória compartilhada entre as réplicas de serviço, bem como um modelo híbrido de falhas, nos possibilitou a redução da complexidade para a elaboração de um protocolo de consenso. Comparado a abordagens anteriores, nossa proposta envolve um número reduzido de passos de comunicação entre as réplicas, mostrando assim que a utilização de componentes seguros para o desenvolvimento de serviços tolerantes a faltas bizantinas/intrusões é uma alternativa viável para a proposta de novos protocolos. A utilização de uma memória compartilhada entre as réplicas de serviço se mostrou uma alternativa bastante atrativa, por se utilizar de uma característica específica de ambientes virtualizados, os quais são muito utilizados na prática. Assim, esperamos que esse trabalho sirva para impulsionar novos estudos basea-

dos em tolerância a faltas através de ambientes virtualizados, bem como estimule novos estudos baseados na utilização de componentes seguros para simplificação de protocolos que garantam a corretude de sistemas.

Durante a elaboração deste trabalho, identificamos alguns pontos que podem representar potenciais trabalhos futuros, os quais são listados abaixo:

- Exploração de novas propostas para a abordagem distribuída de SMIT, baseadas em abordagens de replicação diferentes da técnica utilizada neste trabalho, a replicação primário-*backup*.
- Comparação dos resultados de nossa proposta com outras abordagens.
- Implementação da *postbox* utilizando outro VMM e memória principal, para verificar possíveis melhorias de desempenho em tal componente.

Além deste documento, o trabalho aqui apresentado gerou dois artigos aceitos para publicação em eventos:

- "*Desenvolvimento de Serviços Tolerantes a Intrusões Usando Máquinas Virtuais*" - XXXVI Seminário Integrado de Software e Hardware - SEMISH 2009 [Stumm Júnior et al. 2009].
- "*Intrusion Tolerant Services Through Virtualization: a Shared Memory Approach*" - *International Conference on Advanced Information Networking and Applications* - AINA 2010 [Stumm Júnior et al. 2010].

Referências Bibliográficas

[Andreasson 2003]ANDREASSON, O. *Iptables Tutorial*. 2003.

[Avižienis 1967]AVIŽIENIS, A. Design of fault-tolerant computers. In: ACM. *Proceedings of the Fall Joint Computer Conference*. [S.l.], 1967. p. 733–743.

[Avizienis e Kelly 1984]AVIZIENIS, A.; KELLY, J. P. J. Fault tolerance by design diversity: Concepts and experiments. *Computer*, v. 17, n. 8, p. 67–80, 1984.

[Avizienis et al. 2001]AVIZIENIS, A. et al. Fundamental concepts of dependability. *Technical Report Series - University Of Newcastle Upon Tyne Computing Science*, 2001.

[Avizienis et al. 2004]AVIZIENIS, A. et al. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on dependable and secure computing*, IEEE Computer Society, p. 11–33, 2004.

[Babaoğlu, Davoli e Montresor 1997]BABAOĞLU, Ö.; DAVOLI, R.; MONTRESOR, A. Group membership and view synchrony in partitionable asynchronous distributed systems: Specifications. *ACM SIGOPS Operating Systems Review*, ACM, v. 31, n. 2, p. 22, 1997.

[Baldoni et al. 2003]BALDONI, R. et al. Consensus in Byzantine asynchronous systems. *Journal of Discrete Algorithms*, Elsevier, v. 1, n. 2, p. 185–210, 2003.

[Barham et al. 2003]BARHAM, P. et al. Xen and the art of virtualization. In: ACM. *Proceedings of the nineteenth ACM symposium on Operating systems principles*. [S.l.], 2003. p. 177.

[Bessani et al. 2007]BESSANI, A. et al. Intrusion-tolerant protection for critical infrastructures. *Dep. of Informatics, Univ. of Lisbon, DI/FCUL TR*, p. 07–8, 2007.

[Bessani et al. 2008]BESSANI, A. et al. The CRUTIAL way of critical infrastructure protection. *IEEE Security and Privacy*, IEEE Educational Activities Department, v. 6, n. 6, p. 44–51, 2008.

[Bessani et al. 2007]BESSANI, A. N. et al. Intrusion-tolerant protection for critical infrastructures. *DI/FCUL TR*, p. 07–8, 2007.

[Bracha e Toueg 1985]BRACHA, G.; TOUEG, S. Asynchronous consensus and broadcast protocols. *Journal of the ACM (JACM)*, ACM New York, NY, USA, v. 32, n. 4, p. 824–840, 1985.

[Budhiraja et al. 1993]BUDHIRAJA, N. et al. The primary-backup approach. *Distributed systems*, v. 2, p. 199–216, 1993.

[Castro e Liskov 1999]CASTRO, M.; LISKOV, B. Practical byzantine fault tolerance. *Proceedings of Third Symposium on Operating Systems Design and Implementation*, v. 1, 1999.

[Castro e Liskov 2000]CASTRO, M.; LISKOV, B. Proactive recovery in a Byzantine-fault-tolerant system. In: *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*. [S.l.: s.n.], 2000.

[Castro e Liskov 2002]CASTRO, M.; LISKOV, B. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, ACM New York, NY, USA, v. 20, n. 4, p. 398–461, 2002.

[Cavallar et al. 2000]CAVALLAR, S. et al. Factorization of a 512-bit RSA modulus. *Lecture Notes in Computer Science*, Springer, p. 1–18, 2000.

[Chandra e Toueg 1996]CHANDRA, T.; TOUEG, S. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, ACM New York, NY, USA, v. 43, n. 2, p. 225–267, 1996.

[Chew e Song 2002]CHEW, M.; SONG, D. *Mitigating buffer overflows by operating system randomization*. [S.l.], 2002.

[Chisnall 2007]CHISNALL, D. The definitive guide to the Xen hypervisor. Prentice Hall Press Upper Saddle River, NJ, USA, 2007.

[Chun, Maniatis e Shenker 2008]CHUN, B. G.; MANIATIS, P.; SHENKER, S. Diverse replication for single-machine byzantine-fault tolerance. In: *Proceedings of the 2008 USENIX Annual Technical Conference*. [S.l.: s.n.], 2008.

[Chun et al. 2007]CHUN, B. G. et al. Attested append-only memory: making adversaries stick to their word. p. 189–204, 2007.

[Correia 2005]CORREIA, M. Serviços Distribuídos Tolerantes a Intrusões: resultados recentes e problemas abertos. *Proceedings of 8th Brazilian Symposium on Information and Computer System Security*, p. 113–162, 2005.

[Correia et al. 2002]CORREIA, M. et al. Efficient Byzantine-resilient reliable multicast on a hybrid failure model. In: *PROCEEDINGS OF THE SYMPOSIUM ON RELIABLE DISTRIBUTED SYSTEMS*. [S.l.: s.n.], 2002. p. 2–11.

[Correia et al. 2005]CORREIA, M. et al. Low complexity Byzantine-resilient consensus. *Distributed Computing*, Springer, v. 17, n. 3, p. 237–249, 2005.

[Correia et al. 2007]CORREIA, M. et al. Worm-it–a wormhole-based intrusion-tolerant group communication system. *The Journal of Systems & Software*, Elsevier, v. 80, n. 2, p. 178–197, 2007.

[Correia, Neves e Verissimo 2004]CORREIA, M.; NEVES, N. F.; VERISSIMO, P. How to tolerate half less one byzantine nodes in practical distributed systems. p. 174–183, 2004.

[Coulouris, Dollimore e Kindberg 2005]COULOURIS, G.; DOLLIMORE, J.; KINDBERG, T. *Distributed systems: concepts and design*. [S.l.]: Addison-Wesley Longman, 2005.

[Défago, Schiper e Sergent 1998]DÉFAGO, X.; SCHIPER, A.; SERGENT, N. Semi-passive replication. In: *17th IEEE Symposium on Reliable Distributed Systems, 1998. Proceedings*. [S.l.: s.n.], 1998. p. 43–50.

[Devices 2007]DEVICES, A. M. AMD64 architecture programmer's manual volume 2: System programming. *AMD Publication*, v. 24594, 2007.

[Doudou 1997]DOUDOU, A. Muteness detectors for consensus with Byzantine processes. In: *Proceedings of the 17th ACM Symposium on Principle of Distributed Computing*. [S.l.: s.n.], 1997.

[Doudou, Garbinato e Guerraoui 2002]DOUDOU, A.; GARBINATO, B.; GUERRAOUI, R. Encapsulating failure detection: From crash to byzantine failures. *Lecture notes in computer science*, Springer, p. 24–50, 2002.

[Dwork, Lynch e Stockmeyer 1988]DWORK, C.; LYNCH, N.; STOCKMEYER, L. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, ACM New York, NY, USA, v. 35, n. 2, p. 288–323, 1988.

[Fernandez, Jimenez e Raynal 2007]FERNANDEZ, A.; JIMENEZ, E.; RAYNAL, M. Electing an eventual leader in an asynchronous shared

memory system. In: *Dependable Systems and Networks, 2007. DSN'07. 37th Annual IEEE/IFIP International Conference on*. [S.l.: s.n.], 2007. p. 399–408.

[FIPS 2001]FIPS, P. 197: Advanced Encryption Standard. *National Inst. of Standards & Tech*, 2001.

[Fischer 1983]FISCHER, M. The consensus problem in unreliable distributed systems (a brief survey). In: SPRINGER. *Foundations of Computation Theory*. [S.l.], 1983. p. 127–140.

[Fischer, Lynch e Paterson 1985]FISCHER, M. J.; LYNCH, N. A.; PATERSON, M. S. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, ACM New York, NY, USA, v. 32, n. 2, p. 374–382, 1985.

[Fraga e Powell 1985]FRAGA, J. S.; POWELL, D. A fault- and intrusion-tolerant file system. In: *Proceedings of the 3rd International Conference on Computer Security*. [S.l.: s.n.], 1985. p. 203–218.

[Garfinkel e Rosenblum 2003]GARFINKEL, T.; ROSENBLUM, M. A virtual machine introspection based architecture for intrusion detection. In: *Proceedings of Network and Distributed Systems Security Symposium*. [S.l.: s.n.], 2003. v. 1, p. 253–285.

[Goldberg 1974]GOLDBERG, R. P. Survey of virtual machine research. *IEEE Computer*, v. 7, n. 6, p. 34–45, 1974.

[Guerraoui e Schiper 1996]GUERRAOUI, R.; SCHIPER, A. Fault-tolerance by replication in distributed systems. *Lecture Notes in Computer Science*, Springer, v. 1088, p. 38–57, 1996.

[Hadzilacos e Toueg 1994]HADZILACOS, V.; TOUEG, S. *A modular approach to fault-tolerant broadcasts and related problems*. [S.l.], 1994. 94–1425 p.

[Hiltunen, Schlichting e Ugarte 2003]HILTUNEN, M. A.; SCHLICHTING, R. D.; UGARTE, C. A. Building survivable services using redundancy and adaptation. *IEEE Transactions On Computers*, IEEE Computer Society, p. 181–194, 2003.

[Intel 2008]INTEL. Trusted execution technology software development guide. 2008.

[Kihlstrom, Moser e Melliar-Smith 2003]KIHLSTROM, K.; MOSER, L.; MELLIAR-SMITH, P. Byzantine fault detectors for solving consensus. *The Computer Journal*, Br Computer Soc, v. 46, n. 1, p. 16, 2003.

[King, Dunlap e Chen 2003]KING, S.; DUNLAP, G.; CHEN, P. Operating system support for virtual machines. In: *Proceedings of the 2003 Annual USENIX Technical Conference*. [S.l.: s.n.], 2003.

[Kotla et al. 2007]KOTLA, R. et al. Zyzzyva: speculative byzantine fault tolerance. In: ACM PRESS NEW YORK, NY, USA. *Proceedings of 21st ACM SIGOPS symposium on Operating systems principles*. [S.l.], 2007. p. 45–58.

[Lamport 1977]LAMPORT, L. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, IEEE Press, Piscataway, NJ, USA, v. 3, n. 2, p. 125–143, 1977. ISSN 0098-5589.

[Lamport 1978]LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Communications*, 1978.

[Lamport, Shostak e Pease 1982]LAMPORT, L.; SHOSTAK, R.; PEASE, M. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, ACM New York, NY, USA, v. 4, n. 3, p. 382–401, 1982.

[Laprie, Avizienis e Kopetz 1992]LAPRIE, J.; AVIZIENIS, A.; KOPETZ, H. *Dependability: Basic concepts and terminology*. [S.l.]: Springer-Verlag New York, Inc. Secaucus, NJ, USA, 1992.

[Laureano, Maziero e Jamhour 2004]LAUREANO, M.; MAZIERO, C.; JAMHOUR, E. Intrusion detection in virtual machine environments. In: *EUROMICRO Conference*. [S.l.: s.n.], 2004.

[Leung e White 1989]LEUNG, H.; WHITE, L. Insights into regression testing [software testing]. In: *Software Maintenance, 1989., Proceedings., Conference on*. [S.l.: s.n.], 1989. p. 60–69.

[Lindholm e Yellin 1999]LINDHOLM, T.; YELLIN, F. *Java(tm) Virtual Machine Specification*. [S.l.]: Addison-Wesley Professional, 1999.

[Luiz et al. 2007]LUIZ, A. F. et al. RePEATS - uma arquitetura para replicação tolerante a faltas bizantinas baseada em espaço de tuplas. *Proceedings of 26th Brazilian Symposium on Computer Networks and Distributed Systems*, 2007.

[Microsystems 2009]MICROSYSTEMS, I. S. Sun virtualbox user manual. http://www.virtualbox.org/manual/UserManual.html, 2009.

[Nagappan, Ball e Zeller 2006]NAGAPPAN, N.; BALL, T.; ZELLER, A. Mining Metrics to Predict Component Failures. In: ACM. *Proceedings of the 28th International Conference on Software Engineering*. [S.l.], 2006.

[Neves, Correia e Verissimo 2005]NEVES, N.; CORREIA, M.; VERISSIMO, P. Solving vector consensus with a wormhole. *IEEE Transactions on Parallel and Distributed Systems*, Institute of Electrical and Electronics Engineers, Inc, 445 Hoes Ln, Piscataway, NJ, 08854-1331, USA,, v. 16, n. 12, p. 1120–1131, 2005.

[Nightingale, Chen e Flinn 2005]NIGHTINGALE, E.; CHEN, P.; FLINN, J. Speculative execution in a distributed file system. *ACM SIGOPS Operating Systems Review*, ACM, v. 39, n. 5, p. 205, 2005.

[Padala et al. 2007]PADALA, P. et al. Performance evaluation of virtualization technologies for server consolidation. *HP Laboratories Technical Report*, 2007.

[Parmelee et al. 1972]PARMELEE, R. et al. Virtual storage and virtual machine concepts. *IBM Journal of Research and Development*, v. 11, n. 2, p. 99, 1972.

[Popek e Goldberg 1974]POPEK, G.; GOLDBERG, R. Formal requirements for virtualizable third generation architectures. ACM New York, NY, USA, 1974.

[PUB 1994]PUB, N. 186–Digital Signature Standard. *National Institute of Standards and Technology, US Department of Commerce*, 1994.

[Python Software Foundation 2010]Python Software Foundation. Python programming language. http://www.python.org/, 2010.

[Qumranet 2006]QUMRANET. KVM: Kernel-based virtualization driver. *White Paper*, 2006.

[Randell 1975]RANDELL, B. System structure for software fault tolerance. In: ACM NEW YORK, NY, USA. *Proceedings of the international conference on Reliable software table of contents*. [S.l.], 1975. p. 437–449.

[Reiser et al. 2006]REISER, H. P. et al. Hypervisor-based redundant execution on a single physical host. In: *Proceedings of the 6th European Dependable Computing Conference, Supplemental Volume - EDCC'06 (Oct 18-20, 2006, Coimbra, Portugal)*. [S.l.: s.n.], 2006. p. 67–68.

[Reiser e Kapitza 2007]REISER, H. P.; KAPITZA, R. Hypervisor-based efficient proactive recovery. In: *Proceedings of 26th IEEE International Symposium on Reliable Distributed Systems*. [S.l.: s.n.], 2007. p. 83–92.

[Reiser e Kapitza 2007]REISER, H. P.; KAPITZA, R. VM-FIT: Supporting intrusion tolerance with virtualisation technology. In: *Proceedings of the 1st Workshop on Recent Advances on Intrusion-Tolerant Systems*. [S.l.: s.n.], 2007. p. 18–22.

[Rivest, Shamir e Adleman 1978]RIVEST, R.; SHAMIR, A.; ADLEMAN, L. A method for obtaining digital signatures and public-key cryptosystems. ACM New York, NY, USA, 1978.

[Rosenblum 2004]ROSENBLUM, M. The reincarnation of virtual machines. *Queue*, ACM, v. 2, n. 5, p. 40, 2004.

[Rosenblum e Garfinkel 2005]ROSENBLUM, M.; GARFINKEL, T. Virtual machine monitors: current technology and future trends. *Computer*, v. 38, n. 5, p. 39–47, May 2005. ISSN 0018-9162.

[Schneider 1990]SCHNEIDER, F. B. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys (CSUR)*, ACM Press New York, NY, USA, v. 22, n. 4, p. 299–319, 1990.

[Shamir 1979]SHAMIR, A. How to share a secret. *Communications of the ACM*, ACM, v. 22, n. 11, p. 612–613, 1979.

[Smith e Nair 2005]SMITH, J.; NAIR, R. The architecture of virtual machines. *Computer*, IEEE Computer Society, p. 32–38, 2005.

[Sousa et al. 2007]SOUSA, P. et al. Resilient intrusion tolerance through proactive and reactive recovery. In: *Proceedings of the 13th IEEE Pacific Rim Dependable Computing Conference*. [S.l.: s.n.], 2007. p. 373–380.

[Stumm Júnior et al. 2009]Stumm Júnior, V. et al. Desenvolvimento de Serviços Tolerantes a Intrusões Usando Máquinas Virtuais. In: *Anais do XXXVI Seminário Integrado de Software e Hardware - SEMISH*. [S.l.: s.n.], 2009.

[Stumm Júnior et al. 2010]Stumm Júnior, V. et al. Intrusion Tolerant Services Through Virtualization: a Shared Memory Approach. In: *Proceedings of the 24th International Conference on Advanced Information Networking and Applications - AINA 2010*. [S.l.: s.n.], 2010.

[Thai e Lam 2002]THAI, T.; LAM, H. *.NET framework essentials*. [S.l.]: O'Reilly & Associates, Inc., 2002.

[Verissimo, Neves e Correia 2003]VERISSIMO, P.; NEVES, N.; CORREIA, M. Intrusion-tolerant architectures: Concepts and design. *Lecture Notes in Computer Science*, Springer, p. 3–36, 2003.

[Verissimo e Rodrigues 2001]VERISSIMO, P.; RODRIGUES, L. *Distributed systems for system architects*. [S.l.]: Kluwer Academic Publishers, 2001.

[Veronese et al. 2008]VERONESE, G. S. et al. *Minimal Byzantine Fault Tolerance*. [S.l.], December 2008. Disponível em: <http://www.di.fc.ul.pt/tech-reports/08-29.pdf>.

[Weber 2003]WEBER, T. Um roteiro para exploração dos conceitos básicos de tolerância a falhas. *Programa de Pós-Graduação em Computação-Instituto de Informática–UFRGS, Disponível em http://www. inf. ufrgs. br/˜ taisy/disciplinas/textos/, Acessado em 13 de abril de 2010.*, 2003.

[Xu, Kalbarczyk e Iyer 2003]XU, J.; KALBARCZYK, Z.; IYER, R. Transparent runtime randomization for security. In: *Proceedings Of The Symposium On Reliable Distributed Systems*. [S.l.: s.n.], 2003. p. 260–272.

[Yin et al. 2003]YIN, J. et al. Separating agreement from execution for byzantine fault tolerant services. *ACM SIGOPS Operating Systems Review*, ACM New York, NY, USA, v. 37, n. 5, p. 253–267, 2003.