UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE AUTOMAÇÃO E SISTEMAS

Luciano Barreto

ITVM: Uma Abordagem para Tolerância a Intrusão Utilizando Máquinas Virtuais

Florianópolis
2012

UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE AUTOMAÇÃO E SISTEMAS

Luciano Barreto

ITVM: Uma Abordagem para Tolerância a Intrusão Utilizando Máquinas Virtuais

Dissertação submetida ao Programa de Pós-Graduação em Engenharia de Automação e Sistemas da Universidade Federal de Santa Catarina para a obtenção do Grau de Mestre em Engenharia de Automação e Sistemas.
Orientador: Prof. Dr. Joni da Silva Fraga

Florianópolis
2012

Catalogação na fonte pela Biblioteca Universitária
da
Universidade Federal de Santa Catarina

B273i Barreto, Luciano
ITVM [dissertação] : uma abordagem para tolerância a intrusão utilizando máquinas virtuais / Luciano Barreto ; orientador, Joni da Silva Fraga. - Florianópolis, SC, 2012.
136 p.: il., grafs., tabs.

Dissertação (mestrado) - Universidade Federal de Santa Catarina, Centro Tecnológico. Programa de Pós-Graduação em Engenharia de Automação e Sistemas.

Inclui referências

1. Engenharia de sistemas. 2. Tolerância a falha - (Computação). 3. Computadores - Medidas de segurança. I. Fraga, Joni da Silva. II. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Engenharia de Automação e Sistemas. III. Título.

CDU 621.3-231.2(021)

Luciano Barreto

ITVM: UMA ABORDAGEM PARA TOLERÂNCIA A INTRUSÃO UTILIZANDO MÁQUINAS VIRTUAIS

Esta Dissertação foi julgada adequada para obtenção do Título de "Mestre em Engenharia de Automação e Sistemas", e aprovada em sua forma final pelo Programa de Pós-Graduação em Engenharia de Automação e Sistemas.

Florianópolis, 27 de abril de 2012.

_____

Prof. Eduardo Ribeiro Cury, Dr.
Coordenador do Curso

Banca Examinadora:

_____

Prof. Joni da Silva Fraga, Dr.
Orientador

_____

Prof. Carlos Barros Montez, Dr.

_____

Prof. Frank Siqueira, Dr.

_____

Prof. Mário Antônio Ribeiro Dantas, Dr.

AGRADECIMENTOS

Primeiramente agradeço a Deus, por ter me dado forças para continuar mesmo quando tudo pareceu perdido e quando as coisas pareciam impossíveis, e não foram poucas essas situações.

Agradeço também a meus pais, Selma Martins Barreto e Mário Barreto pelo incentivo, por acreditar em minha caminhada e sempre se preocuparem com meu bem estar longe de seus olhos. Aos meus irmãos Adriano Barreto e Marcio Ivan Barreto, pelo apoio e atenção quando de volta a minha casa nas visitas que fiz.

Agradecer aos amigos do LTIC, Diego (Presunto), Renan, Gilmar, Andrew, Jim, Günter, Felipe, Thiago, Vanderlei, Jaque, Vitor (Rattus). Desde março de 2010 vocês foram minha família, me ajudando nas pesquisas, nos trabalhos e também fazendo minha caminhada ser mais fácil, quando saímos para um cinema, um churrasco ou um chopinho. Valeu Galera!

Não posso me esquecer de agradecer Rodrigo Teixeira de Castro (Driguera) e a Joice Gonzales, pela amizade e companheirismo. Pelos planos de futuro, por acreditarem em mim e estarem vivendo toda essa mudança em minha vida, desde o momento em que eu soube do aceite no mestrado. Vocês são muito importantes pra mim!

A todos os novos amigos e pessoas especiais que conheci nessa caminhada!

Agradeço também meu orientador prof. Joni da Silva Fraga, por ter me dados os caminhos para a pesquisa.

Por fim, agradeço a CAPES, pelo apoio financeiro, que me permitiu dedicação total a pesquisa.

Sua tarefa é descobrir o seu trabalho e, então, com todo o coração, dedicar-se a ele.

Buda

RESUMO

A Internet tem sido o meio de comunicação utilizado por muitas empresas para divulgarem seus serviços. Dessa forma a segurança de tais serviços é um assunto de grande importância e necessidade. Como os sistemas atualmente são distribuídos pela Internet, estes estão expostos a um meio extremamente hostil, onde intrusões por entidades mal intencionadas acontecem com grande frequência. Estas intrusões causam perdas e danos muitas vezes em proporções catastróficas, tanto para empresas como para a sociedade. Nesta dissertação apresentamos uma infraestrutura cuja finalidade é fornecer suporte de tolerância a intrusões (faltas maliciosas ou bizantinas) para serviços. Nossa abordagem faz uso da tecnologia de virtualização e de memória compartilhada no sentido de conseguir custos mais baixos na execução de protocolos de máquina de estado (ME) em contexto bizantino. O uso da virtualização permitiu a inserção de um componente confiável no modelo. Como esta tecnologia separa em diferentes camadas os processos que gerenciam as máquinas virtuais e a que executam as máquinas virtuais é possível a inserção de um componente que gerencie as réplicas, iniciando ou desligando estas quando necessário. Além disso, este componente confiável é responsável pela execução do serviço de acordo do protocolo, e permitiu a criação de uma arquitetura com custos reduzidos, onde as requisições de clientes podem ser executadas com um número variável de réplicas (entre $f+1$ e $2f+1$). O trabalho discute os algoritmos, apresentam detalhes de protótipo, testes e um confronto com trabalhos relacionados na literatura, onde mostramos que foi obtida uma redução do número de servidores necessários, assim como reduzimos o número de réplicas necessárias para a execução do serviço replicado.

Palavras-chave: Tolerância a Intrusão, Virtualização, Componente Confiável.

ABSTRACT

The Internet has been a means of communication of daily basis for many companies share their service. Presently, the security of such services is a matter of great importance and necessity. These systems are now distributed over the Internet and they are exposed to an extremely hostile environment, where intrusions by malicious entities occur with great frequency. These intrusions cause damages, often catastrophic, both for companies and for society. This work presents an infrastructure based in virtualization which provides support to intrusion tolerance (Byzantine or malicious faults) to services. The introduced approach makes extensive use of virtualization technology and shared memory in order to reduce costs in the execution of state machine (ME) protocols in the Byzantine context. The use of virtualization technology allowed the insertion of a trusted component that in the model. As this technology separates into different layers, processes that manage the virtual machines and process that execute virtual machines, it is possible to insert a component that manages the replicas, starting or turning off these when necessary. In addition, this reliable component is responsible for the execution of agreement service, and allowed the creation of an architecture with reduced costs, where the customer requests can be performed with a variable number of replicas (between $f+1$ and $2f + 1$). This work discusses the algorithms, presents details of a prototype, tests and a comparison with related work in the literature.

Keywords: Intrusion Tolerance, Virtualization, Trusted Components.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ME	-	Máquina de Estados
BFT	-	Byzantine Fault Tolerance
PBFT	-	Practical Byzantine Fault Tolerance
VMM	-	Virtual Machine Monitor
ITVM	-	Intrusion Tolerance Using Virtual Machines

LISTA DE ALGORITMOS

SUMÁRIO

1 INTRODUÇÃO

1.1 MOTIVAÇÃO

A Internet tem sido um meio de comunicação de extrema importância para empresas estabelecerem e divulgarem seus serviços. Consultas públicas governamentais, redes de comércio eletrônico (*business-to-business*) e mesmo acessos a serviços bancários estão disseminados pela rede mundial e fazem parte de nossas rotinas diárias. Esta profusão de serviços vem também acompanhada por preocupações. Devido à necessidade crescente da utilização destes serviços, estes devem estar disponíveis em tempo integral e funcionar de forma correta, pois, qualquer mau funcionamento pode causar grandes perdas financeiras a empresas ou mesmo criar problemas de ordem pessoal a seus usuários.

Apesar de toda a evolução experimentada pelas tecnologias de segurança, as invasões de provedores de serviço são fatos frequentes na Internet. A dificuldade em lidar com este problema é devido à complexidade sempre crescente dos suportes para aplicações disponíveis. Por mais cuidado que um projetista tenha no desenvolvimento destes sistemas, por mais que se acerque de técnicas de prevenção, vulnerabilidades sempre estarão presentes nestes sistemas e estas continuam sendo exploradas, com o objetivo de causar desordem ou mesmo prejuízo financeiro.

Na dificuldade de evitar acessos ilegais, muitas abordagens têm sido apresentadas na literatura no sentido de conter ou restringir as possíveis intrusões. Neste trabalho, temos interesse nos modelos de tolerância a intrusões para sistemas distribuídos (CORREIA, M. P., 2005). A ideia, neste caso, é manter o sistema distribuído evoluindo com comportamento correto mesmo diante de componentes já corrompidos por intrusões. A ação maliciosa destes componentes invadidos não deve afetar o sistema como um todo. Na literatura sobre segurança de funcionamento (LAPRIE, 1995), a noção de faltas bizantinas (ou faltas maliciosas) é usada para descrever o comportamento de subcomponentes comprometidos de um sistema. As soluções para estas corrupções parciais de um sistema envolvem o uso de técnica de replicação ativa, também conhecida como replicação Máquina de Estado (ME) (SCHNEIDER, 1990).

São muitas as experiências na literatura do uso de replicação ME para manter o serviço disponível mesmo diante de intrusões e réplicas sendo corrompidas (CASTRO; LISKOV, 2002a; CHUN, B. *et al.*, 2008; KOTLA *et al.*, 2007; VERONESE *et al.*, 2009; WOOD *et al.*, 2011). Os suportes para replicações nestes trabalhos faz uso de algoritmos de acordo para ambientes bizantinos. A grande maioria destes trabalhos é baseada no PBFT (*Practical Byzantine Fault Tolerance* (CASTRO; LISKOV, 2002a)). Nestas abordagens de replicações ativa, cada réplica de serviço é executada em uma máquina física diferente. Estas replicações são implementadas em *clusters* ou redes locais devido aos custos dos protocolos de acordo e à dificuldade dos mesmos em lidar com as características das grandes redes (a complexidade em mensagens limita a implantação dos mesmos em escalas maiores, como a Internet, por exemplo). O custo em mensagens destes protocolos, mesmo em execuções favoráveis (sem faltas maliciosas) assumem valores quadráticos $O(N^2)$, sendo N o número de réplicas participantes nos protocolos de acordo que deve satisfazer a condição $N \geq 3f + 1$ para garantir o acordo (execução correta do algoritmo de acordo). O valor de f nesta condição corresponde ao limite de faltas ou intrusões admitidas no sistema distribuído de modo a não comprometer a correção do mesmo.

1.2 PROPOSTA DO TRABALHO

A motivação principal deste trabalho é apresentar uma alternativa aos trabalhos como os acima citados. Neste sentido, os objetivos centrais do trabalho foram concentrados na introdução do modelo de sistema da abordagem e o desenvolvimento de uma infraestrutura que faz uso extensivo da tecnologia de virtualização na implementação de replicação ME em seu suporte de tolerância a intrusões (faltas maliciosas ou bizantinas) para sistemas distribuídos. A abordagem de BFT (*Byzantine Fault Tolerance*) usada em nosso modelo de replicação restringe então, através da tecnologia de virtualização, os passos de nosso protocolo de acordo a acessos em memória compartilhada. Esta restrição permite que seja executado um protocolo tolerante a intrusão com um menor número de réplicas físicas e com menos trocas de mensagens via rede.

O modelo, com o uso da virtualização, possibilita a definição de um elemento confiável que vai determinar a separação de faltas de *crash* e faltas maliciosas assim como a simplificação de nossos algoritmos. As

réplicas de serviço construídas sobre máquinas virtuais ficam envolvidas exclusivamente com protocolos BFT (os mais custosos), portanto limitados a trocas via memória compartilhada. O nosso algoritmo de BFT, nestas condições, concretiza execuções de requisições de clientes com um número variável de réplicas (entre f +1 e 2f+1).

1.3 OBJETIVOS DA DISSERTAÇÃO

Este trabalho tem como objetivo o desenvolvimento de um modelo de sistema distribuído tolerante a intrusões, que ofereça as mesmas garantias dos trabalhos já existentes na literatura, porém com a redução em seus custos de implementação.

Diferentemente da grande parte dos trabalhos que apresentam abordagem tolerantes a intrusão, onde um protocolo de ordenação de requisições é definido através de trocas de mensagens via rede, exploramos a utilização de memória compartilhada e a adoção de um componente confiável para esta tarefa. Apesar da distinção de nosso trabalho para com os demais da literatura, o objetivo final continua o mesmo, uma implementação de ME com as mesmas propriedades definidas em (SCHNEIDER, 1990).

Para que possamos obter a replicação Máquina de Estados com um menor custo e alcançar nossos objetivos, fazemos uso da tecnologia de virtualização. A tecnologia de virtualização irá nos dar a possibilidade da utilização de um menor número de servidores físicos, além de auxiliar na utilização da memória compartilhada e da utilização de um componente confiável.

Para o desenvolvimento deste trabalho, alguns objetivos específicos também foram colocados:

1) Levantamento e análise bibliográfica de tolerância a intrusões, no sentido de estabelecimento de uma base conceitual para referências e comparações dos conceitos e algoritmos que viéssemos a produzir.
2) Concepção de uma abordagem que apresente melhoras em relação à literatura, considerando as necessidades de redundâncias e custos de protocolos.
3) Explorar conceitos como o de componentes confiáveis e de isolamento em ambientes maliciosos.

4) Desenvolvimento de um protótipo baseado nestes protocolos, com o objetivo de validar e provar que nossa abordagem garante que o serviço se mantém seguro mesmo na presença de intrusões.

1.4 ESTRUTURA DO TEXTO

Esta dissertação é composta por 6 capítulos. Os capítulos restantes deste trabalho estão organizados da seguinte maneira:

Capítulo 2 - Conceitos Básicos em Sistemas Distribuídos e Segurança de Funcionamento: Neste capítulo apresentamos os conceitos básicos sobre sistemas distribuídos, para que uma base teórica seja formada e vários conceitos utilizados ao longo do texto possam ser fundamentados.

Capítulo 3 - Serviços Tolerantes a Intrusões: Trabalhos Relacionados: É apresentado o estado da arte em tolerância a intrusões. Além disso, apresentamos trabalhos que podem ser vistos como relacionados à nossa proposta.

Capítulo 4 - Uma Abordagem para Tolerância a Intrusões Usando Máquinas Virtuais: Neste capítulo consta todo o desenvolvimento de nossa abordagem. Apresentamos as premissas assumidas e os protocolos desenvolvidos.

Capítulo 5 - Resultados Experimentais e Análises: Discutimos neste capítulo o desenvolvimento do protótipo e análise dos resultados obtidos. Além disso, fazemos aqui a comparação de nossa abordagem com uma abordagem semelhante.

Capítulo 6 – Conclusões e Perspectivas Futuras: Por fim, neste capítulo, apresentamos as conclusões obtidas com o desenvolvimento de nosso trabalho, bem como os pontos que podem ser ajustados e dar continuidade à nossa abordagem em trabalhos futuros.

2 CONCEITOS BÁSICOS EM SISTEMAS DISTRIBUÍDOS E SEGURANÇA DE FUNCIONAMENTO

O objetivo deste capítulo é apresentar conceitos essenciais de Sistemas Distribuídos e Segurança de Funcionamento. Tais conceitos são fundamentais para a compreensão dos demais capítulos do trabalho. Dentro deste contexto, será apresentado o modelo de sistema no qual este trabalho está inserido e que contém as premissas necessárias para a correção dos algoritmos desenvolvidos para aplicações em ambientes distribuídos. A correção e, por consequência a segurança de funcionamento, são de forma implícita dependentes de problemas já clássicos de sistemas distribuídos, como os problemas de acordo (em especial o problema de consenso). Neste capítulo serão apresentados conceitos relacionados à tolerância a faltas e à tolerância a intrusões, alguns problemas básicos de sistemas distribuídos que são usados para garantir as propriedades de correção e de segurança de funcionamento das aplicações distribuídas. Apresentamos também no capítulo, conceitos sobre a tecnologia de virtualização, a qual é utilizada como ferramenta de apoio na construção de soluções de protocolos tolerantes a intrusões mais eficientes e menos custosos.

2.1 SEGURANÇA DE FUNCIONAMENTO

A Segurança de Funcionamento é uma disciplina que regra projetos e o uso de técnicas no sentido de agregar qualidade de serviço a sistemas computacionais (AVIZIENIS, A. *et al.*, 2004). Um sistema é dito seguro de funcionamento quando o mesmo mantém as seguintes propriedades durante a sua evolução (AVIZIENIS, A. *et al.*, 2004):

- **Confiabilidade**: um sistema é dito confiável quando seus clientes percebem que o mesmo evolui em seu ciclo de vida sem desviar de suas especificações de serviço.
- **Disponibilidade**: o sistema sempre está pronto para executar as operações requisitadas por partes autorizadas.
- **Integridade**: nenhuma alteração imprópria no estado do sistema pode ocorrer, i.e., o estado do sistema só pode ser alterado através da correta execução de suas operações;

- **Confidencialidade**: dados confidenciais não podem ser revelados a partes não autorizadas;

Um sistema nem sempre contempla todos os atributos citados acima. A aplicação e suas necessidades é que determinam as propriedades que devem ser suportadas por um sistema computacional. Nas descrições de nossos estudos neste texto, mostraremos as nossas preocupações com estes atributos listados.

2.2 IMPERFEIÇÕES EM UM SISTEMA

As imperfeições em um sistema podem ser definidas pela relação entre seu comportamento esperado e o percebido pelos seus clientes. E neste sentido a literatura identifica três níveis de imperfeições em sistemas (AVIZIENIS, A. *et al.*, 2004):

- *Falta* é a causa de mau funcionamento de um sistema ou serviço. Faltas de projeto, interação com o ambiente, operação, são exemplos da origem destas faltas em componentes de um sistema (elementos faltosos).
- Um *erro* é a manifestação interna de uma falta no sistema. Estados errôneos são detectados e podem ser corrigidos durante a operação do sistema.
- A *falha* de um sistema ocorre quando o serviço fornecido desvia de suas especificações. É, portanto, a manifestação externa ao sistema proveniente da ativação um elemento faltoso.

As **falhas** são percebidas ou detectadas pelos clientes de um serviço. Os sistemas podem conviver com **erros** internos sem que necessariamente venham a falhar (desviar de suas especificações). Uma **falta** só pode ser evitada (por prevenção no projeto ou na operação) ou tolerada com o uso de redundâncias. As noções de **faltas**, **erros** e **falhas** quando transportadas para o contexto da segurança (*security*) tomam os nomes de vulnerabilidades (faltas), intrusão (erro) e violações (falhas).

2.3 CLASSIFICAÇÃO DE FALTAS SEGUNDO SUAS SEMÂNTICAS DE FALHAS

As classificações de faltas e o conhecimento da natureza de suas falhas são de grande importância no projeto de sistemas distribuídos capazes de tolerar a existência de componentes faltosos (tolerar faltas).

São várias classificações existentes que levam em consideração inúmeros fatores: a *origem* (humanas/projeto/físicas); a *persistência* (transitórias/intermitentes/permanentes), etc. O que quase sempre é considerada no projeto de aplicações distribuídas e nos interessa neste texto é a classificação de faltas segundo a ótica de seus efeitos, ou seja, segundo as sua **Semântica de Falha**.

Se considerarmos as falhas parciais em um sistema distribuído, ou seja, assumindo as abstrações usuais de um modelo computacional, o comportamento de componentes faltosos engloba somente as falhas de processos e as falhas de canais de comunicação. Considerando então os efeitos de falhas de elementos faltosos de um sistema distribuído, podemos classificar as faltas conforme apresentado na Tabela 2.1.

As **faltas de parada** são as mais simples e consideradas benignas, uma vez que ocorrida a falta no sistema, o elemento faltoso correspondente cessa o seu funcionamento. As **faltas de omissão** envolvem a não entrega de resultados em uma instância de serviço (em outras palavras, envolvem o não atendimento de determinadas requisições de serviço); depois, em requisições subsequentes a uma omissão, o serviço volta a funcionar segundo suas especificações. As **faltas de temporização** envolvem o não atendimento de requisições dentro de prazos especificados (restrições temporais).

As falhas de parada e omissão são casos extremos de faltas de temporização (ou seja, estas faltas envolvem somente o domínio do tempo). As faltas ditas **arbitrárias ou bizantinas** correspondem à classe de faltas mais complexa. Neste último caso, além do domínio do tempo, temos também envolvido o domínio de valores.

Os processos desviam do comportamento definido em seu algoritmo, podendo alterar mensagens, omitir mensagens, ou qualquer outro comportamento aleatório. Este último tipo de falta (faltas bizantinas ou maliciosas) podem modelar comportamentos intrusivos presentes em sistemas invadidos e com propriedades de segurança alteradas.

Tabela 2.1 - Classificação de faltas em Sistemas Distribuídos

Classe de Faltas		**Componente**	**Descrição do Efeito (Semântica de Falha)**
Faltas de Parada (Crash)		Processo e Canal	Em uma parada (*crash*) o componente cessa o funcionamento.
Faltas de Omissão (Omission faults)		Processo	Em falta por omissão, o processo não atende a uma solicitação de serviço.
		Canal	Mensagem inserida no canal nunca chega ao *buffer* de saída.
Faltas de temporização (Timing failures)	*Desempenho*	Processo e canal	*Timing failure* ocorre na violação das restrições de tempo: Processo ou comunicação excede restrições temporais. Este tipo de falha é irrelevante em sistemas assíncronos.
	Clock	Processo	Relógio local de processo excede taxa de desvio máxima tolerável.
Faltas Arbitrárias ou Maliciosas (*Faltas Bizantinas*)	*Bizantinas Puras*	Processo e Canal	Processo ou canal com comportamento arbitrário: transmite mensagens com valor arbitrário em tempos arbitrários.
	Bizantinas Autenticadas	Processo e Canal	Falha bizantina menos complexa: processo/canal com comportamento arbitrário, mas mensagens assinadas permitem detecção.

2.4 TOLERÂNCIA A FALTAS E INTRUSÕES

Existem técnicas de **Prevenção de Faltas**, de **Tolerância a Faltas**, de **Remoção de Faltas** e de **Previsão de Faltas**. A **Prevenção de Faltas** envolve métodos e técnicas que têm como objetivo fornecer um sistema onde seja minimizada a ocorrência de faltas. A **Tolerância a Faltas** define técnicas baseadas em redundâncias que garantam o funcionamento correto dos serviços mesmo diante de elementos faltosos em um sistema. A **Remoção e Previsão de Faltas** tentam atender os atributos de segurança de funcionamento pela justificativa de que as especificações são adequadas para o sistema e que o mesmo provavelmente atenderá estas especificações. O presente trabalho se concentra em mecanismos de Tolerância a Faltas para o fornecimento da segurança de funcionamento.

A sobreposição de atributos de confiabilidade com os da segurança fez surgir a Tolerância a Intrusões. Agora sistemas são invadidos e códigos maliciosos são plantados visando tirar proveito do mau funcionamento destes sistemas. Diante destas intrusões e dos comportamentos bizantinos dos elementos invadidos, o uso de redundâncias pode ser determinante para o funcionamento correto destes sistemas. Ou seja, sistemas devem fornecer serviços de forma correta mesmo na presença de faltas e intrusões. Estes sistemas, capazes de permanecer corretos mesmo em circunstâncias adversas, são ditos sistemas tolerantes intrusões (FRAGA, J.; POWELL, 1985), (VERÍSSIMO *et al.*, 2003).

Tolerar faltas sempre envolve a utilização de replicação do serviço fornecido, i.e., o sistema deve ser replicado em um conjunto de servidores, sendo que falhas que possam ocorrer em alguns destes são mascaradas pelos demais servidores corretos. Desta forma, o sistema como um todo permanece correto mesmo que uma parte de seus componentes tenha comportamento falho. Quando consideramos um ambiente sujeito a faltas Bizantinas (LAMPORT, Leslie *et al.*, 1982), o sistema deve ser tolerante a intrusões, i.e., o sistema deve continuar correto mesmo que uma parte dos processos que o compõe exiba qualquer comportamento malicioso. Este comportamento malicioso que algum servidor (processo) pode apresentar geralmente está relacionado com a invasão deste servidor, por isso o sistema é dito tolerante a intrusões.

2.5 MODELOS EM SISTEMAS DISTRIBUÍDOS

Quando sistemas distribuídos são projetados, algumas premissas são assumidas referentes ao ambiente onde tais sistemas e seus algoritmos serão executados. Estas premissas que normalmente definem o comportamento de entidades são determinantes para demonstrar o que são chamados de modelos de um sistema. Tais premissas, portanto, definem situações de ambiente em que algoritmos distribuídos devem funcionar. Em sistemas distribuídos, normalmente é necessária a descrição de três modelos para a caracterização do ambiente de execução de algoritmos distribuídos ou modelo de sistema. São estes: o modelo computacional, o modelo de sincronismo e o modelo de falhas. O modelo computacional descreve as abstrações que deverão compor o processamento da aplicação distribuída. Normalmente, neste modelo computacional são descritos processos e canais de comunicação como as abstrações básicas. Dependendo das propriedades e atributos necessários, estes componentes são acrescidos de funcionalidades ou qualidades de serviço (exemplo, “canais autenticados”).

2.5.1 Modelos de Sincronismo

Para atender requisitos ou propriedade de terminação em sistemas distribuídos, é necessário que os modelos de sistema assumam premissas de sincronismo (DWORK *et al.*, 1988). Em termos de sincronismo, são discutidas as premissas sobre as diferenças máximas de tempo entre os términos de execuções de um mesmo passo de processamento em diferentes processos do sistema, e também a duração máxima das comunicações no sistema (latência das comunicações) que determinam diferentes modelos de sincronismo em sistemas distribuídos (HADZILACOS; TOUEG, 1994), (LYNCH, Nancy A., 1996). Estes modelos de sincronismo podem definir sistemas totalmente dependentes da garantia de atendimento de restrições temporais para as suas correções (sistemas síncronos) (JUNQUEIRA; MARZULLO, 2003), (DOLEV *et al.*, 1987), assim como definir modelos completamente independentes de restrições temporais (sistemas assíncronos) (FISCHER *et al.*, 1985).

Modelos intermediários também são encontrados na literatura (DWORK *et al.*, 1988). Estes modelos relaxam algumas características dos modelos extremos (síncronos e assíncronos), mas não deixam de

preservar a vivacidade ou terminação de suas execuções distribuídas no sistema (*liveness*). Enquanto em sistemas síncronos a diferença entre tempos de execução de um mesmo passo de processamento e comunicação apresenta um tempo limite máximo conhecido, em modelos assíncronos nenhuma hipótese relacionada a restrições temporais é assumida; isto vale tanto para comunicações como para processamentos.

Assumir que o modelo do sistema distribuído é síncrono é muito forte se considerarmos as redes de comunicações comerciais atuais. A não ser por redes muito específicas, como aquelas normalmente usadas em aplicações de tempo real, a grande parte das redes comerciais apresenta fontes de não determinismo, impossibilitando a obtenção de latências máximas. Por outro lado, em modelos de comunicação totalmente assíncronos é comprovada a impossibilidade da solução de problemas de acordo devido à condição FLP (FISCHER *et al.*, 1985): "*é impossível diferenciar a falha de um processo com atraso de uma mensagem em um sistema distribuído assíncrono*".

Desta forma, modelos intermediários são adotados para contornar as impossibilidades e dificuldades dos modelos totalmente assíncrono. Sistemas parcialmente síncronos (também chamados de sistemas com sincronia terminal) são introduzidos em (DWORK *et al.*, 1988) como aqueles em que, para toda execução de um passo de processamento existe um limite Δ' e, um instante de tempo *GST (Global Stabilization Time)*, de maneira que toda mensagem enviada por um processo correto p em um instante de tempo $u > GST$ é recebida por um processo q antes do instante $u + \Delta''$, onde se tem $\Delta', \Delta'' > 0$. Apesar da existência destes limites definidos, os mesmos são desconhecidos pelos processos e também não precisam ser necessariamente o mesmo nas diversas execuções dos sistemas. De forma mais simples, o modelo de sistema parcialmente síncrono define que apesar de o mesmo, na maior parte do tempo, trabalhar de forma assíncrona, existe um momento em que este passa por períodos de estabilidade onde existem limites para as diferenças nos tempos de execução de um mesmo passo de processamento e também para as latências máximas nas comunicações. Desta forma, algoritmos distribuídos baseados em *rounds* de comunicação irão terminar quando este momento de estabilidade acontecer, fazendo com que o sistema seja executado de forma determinista em algum momento.

Em relação aos passos de processamento local do algoritmo, cujos tempos de execução também têm influência sobre o sincronismo do sistema, frequentemente são assumidos como não significativos. Este tipo de premissa é assumido tomando como base o fato de que, apesar de um conjunto de operações locais serem substancialmente pesadas (i.g. operações criptográficas), esses processamentos locais estão pouco sujeito a interferências externas e suas latências são pouco significativas em relação às das comunicações, tornando as suas considerações desprezíveis na prática.

2.5.2 Modelos de Falhas

O modelo de falhas trata do comportamento de componentes em um sistema, quando estes estão sujeitos a faltas. Em sistemas distribuídos, como já foi citado anteriormente, usualmente os componentes considerados em modelos de falhas são processos e canais de comunicação entre estes processos.

Processos ou canais ditos “corretos” são aqueles em que em toda a evolução do sistema apresentam comportamento sempre seguindo suas especificações, ou seja, seguem seus algoritmos fornecendo os serviços esperados pelos seus clientes. Processos ou canais falhos apresentam desvios do comportamento esperado dos mesmos.

Nos modelos de sistema, diferentes tipos de falhas podem ser assumidos envolvendo canais e processos. Certamente, os custos dos algoritmos distribuídos desenvolvidos sobre as premissas destes modelos devem refletir os tipos de faltas assumidos. Algoritmos para faltas bizantinas são mais complexos em relação aos sistemas que envolvem faltas de *crash.* As premissas de falhas de canais seguem basicamente duas definições em sistemas distribuídos (RAYNAL, 2005)**:**

- **Canais confiáveis**: um canal conectando dois processos p_i (emissor) e p_j (receptor) é dito *canal confiável* se este não cria ou duplica mensagens e se toda mensagem enviada por p_i termina por ser recebida em p_j. O fato de não criar ou duplicar mensagens garante a segurança (*safety*) do canal e o fato de sempre chegar as mensagens enviadas indica que a terminação sempre ocorre (*liveness*).

- **Canais justos** (*fair links*): um canal é dito justo se, embora não possa criar ou duplicar mensagens (é um canal *safe*), este pode perder mensagens. O que garante a terminação é a propriedade de que se p_i (emissor) emite um número infinito de mensagens, p_j (receptor) receberá infinitas mensagens de p_i. Este último modelo garante que a repetição de envios de uma mensagem fazem com que a mesma acabe chegando ao receptor.

Portanto, todo o modelo de sistema deve definir o modelo de falha de seus componentes processos (modelo de falhas) e descrever os tipos de canais usados (canais confiáveis ou justos).

2.5.3 Modelos Híbridos de Falhas

Uma abordagem recente em modelos de sistemas é assumir modelos de falhas híbridos (CORREIA, M. *et al.*, 2002), (CHUN, B.-gon *et al.*, 2007), (CHUN, B. *et al.*, 2008), (REISER, Hans P; LISBOA, 2008), (CORREIA, M *et al.*, 2007), onde diferentes partes do sistema podem sofrer diferentes tipos de falhas. Desta forma, enquanto alguns componentes do sistema podem exibir um comportamento de falha controlado (e.g. falhas por *crash*), o restante do sistema apresenta comportamento arbitrário. Estes componentes onde o comportamento é controlado são chamados de componentes confiáveis ou seguros.

A utilização deste tipo de modelo de falhas híbrido dá a possibilidade do desenvolvimento de algoritmos tolerantes a faltas mais simples no trato com semânticas de falhas complexas, como as geradas a partir de faltas bizantinas, podendo diminuir assim o número de réplicas necessárias e também os passos de comunicação desse tipo de algoritmo.

2.6 MECANISMOS E SUPORTES PARA A TOLERÂNCIA A INTRUSÕES

A grande parte dos trabalhos que têm como objetivo a tolerância a intrusões faz uso da técnica de replicação Máquina de Estado (SCHNEIDER, 1990). Esta replicação ativa está fundamentada no Determinismo de Réplicas (POLEDNA, 1996): "*Considerando os buffers de entrada de todas as réplicas do modelo ME e, uma vez garantindo a presença das mesmas requisições de serviço (propriedade de acordo) e a mesma ordem de atendimento das mesmas (propriedade*

de ordem), todas as réplicas corretas partindo do mesmo estado inicial devem percorrer os mesmos estados quando da execução das requisições de serviço". Para a garantia da propriedade de acordo e de ordem na entrada das réplicas do modelo ME normalmente são usados como suporte algoritmos de consenso (GUERRAOUI; SCHIPER, André, 2001) ou de *atomic broadcast* (HADZILACOS; TOUEG, 1994), (CORREIA, M., 2005), (DÉFAGO *et al.*, 2004). As próximas seções apresentam o modelo **Máquina de Estados** (ME) e os problemas clássicos de consenso e de *Atomic Broadcast*. São apresentadas também algumas técnicas adicionais que são usadas para a tolerância a intrusões.

2.6.1 Replicação Máquinas de Estados

A replicação Máquinas de Estados é definida como uma maneira de estruturar um serviço tolerante a faltas por meio da replicação ativa dos servidores. Este modelo envolve a coordenação dos servidores entre si, assim como a coordenação dos mesmos com os seus clientes. O modelo ME teve a sua formalização descrita em (SCHNEIDER, 1990) e se transformou em uma das técnicas mais empregadas para tornar aplicações de sistemas distribuídos tolerantes a faltas. Esta técnica pode ser empregada tanto onde os sistemas podem sofrer faltas por *crash* (LAMPORT, Leslie, 2001) ou mesmo faltas bizantinas (faltas maliciosas) (CASTRO; LISKOV, 2002a).

Em (SCHNEIDER, 1990) é apresentada uma das caracterizações semânticas do modelo ME: *"As saídas de uma máquina de estados são completamente determinadas pela sequência de requisições que a mesma processa, independente do tempo ou qualquer outra atividade em um sistema."*. Esta definição visa explicitar o determinismo das réplicas, de forma que, todas as réplicas partindo de um mesmo estado inicial e executando as mesmas operações em sequência devem chegar a um mesmo estado final. E para que este requisito seja alcançado é preciso que ocorra a coordenação das réplicas, que é sustentado através da garantia de duas propriedades em (SCHNEIDER, 1990):

- **Acordo**: Todas as réplicas corretas recebem as mesmas requisições.
- **Ordem**: Todas das réplicas corretas processam as requisições recebidas em uma mesma ordem.

O atendimento destas propriedades como citado anteriormente pode ser alcançado usando um protocolo de difusão com ordem total (*atomic broadcast*) (DÉFAGO *et al.*, 2004) para troca de mensagens entre clientes e o serviço replicado. O objetivo destes requisitos é que as réplicas apresentem um comportamento determinista, de modo que o estado das mesmas seja sempre mantido de forma consistente, após a execução de uma sequência de requisições. Em [36], tomando como base a equivalência dos problemas de difusão atômica e de consenso, é apresentado uma difusão atômica construída com um protocolo de consenso. Vários outros trabalhos (CASTRO; LISKOV, 2002a), (YIN *et al.*, 2003), (KOTLA *et al.*, 2007) apresentam protocolos cujo objetivo é a garantia do Acordo e Ordem Total em Replicação Máquinas de Estados.

2.6.2 Difusão de Mensagens com Ordem Total (*Atomic Broadcast*)

O problema de *Atomic Broadcast* (*Multicast* com Ordem Total) corresponde a vários emissores enviando mensagens para um grupo e onde a liberação das mensagens para a aplicação, por parte dos componentes do grupo, se dá em uma mesma sequência das mensagens enviadas. Em (HADZILACOS; TOUEG, 1994), é apresentado uma caracterização deste problema, o qual é definido por duas primitivas:

- ***TO-Multicast (m, G)****:* Um processo *p* faz a difusão ao seu grupo *G* de processos de uma mensagem *m*.
- ***TO-Deliver(m):*** Todos os processos corretos do grupo *G* entregam para a aplicação a mensagem *m* obedecendo à mesma ordem total.

Estas duas primitivas, para garantir a ordem total, devem garantir as quatro propriedades abaixo:

- **Validade**: Se um processo correto difunde (*TO-Multicast(m,G)*) uma mensagem *m* em *G* então algum processo correto em *G* terminará por liberar *m* (*TO-Deliver(m))* no futuro[1].
- **Acordo Uniforme**: Se um processo entrega (*TO-Deliver(m)*) uma mensagem *m*, então todos os processos corretos terminarão por liberar *m* para aplicação (*TO-Deliver(m)*) no futuro.

[1] No futuro significa que, em algum momento, a mensagem será recebida. Porém este momento (tempo) é desconhecido.

- **Integridade Uniforme**: Para qualquer mensagem *m* enviada a *G* por um processo *sender(m)*, todos os processos (corretos ou não) de *G* liberam (*TO-Deliver(m*)) *m* no máximo uma vez.
- **Ordem Total Uniforme**: Se *p* e *q*, dois processos em *G*, liberam as mensagens *m* e *m'* (*TO-Deliver(m)* e *TO-Deliver(m')*), então *p* libera *m* antes de *m'* se e somente se *q* libera ambas mensagens na mesma ordem.

O problema de difusão com ordem total vem sendo explorado por quase trinta anos. Várias vertentes do problema se colocam como desafios nos dias de hoje. Um *survey* bastante extenso sobre este problema é apresentado em (DÉFAGO *et al.*, 2004). Neste *survey* podem ser encontrados os principais algoritmos que apresentam soluções para este problema.

2.6.3 Consenso

O consenso também é um problema clássico em sistemas distribuídos. O mesmo foi introduzido inicialmente em (MARSHALL PEASE, ROBERT SHOSTAK, 1980), onde o problema é descrito como um conjunto de processos que devem chegar ao acordo sobre um valor único diante de várias propostas para representar o grupo, isto tudo considerando a presença de processos faltosos. De forma geral, o problema de consenso consiste em um conjunto de processos propondo valores em cada instância do problema, sendo que ao final desta instância de computação distribuída, todos os processos deste grupo devem ter decidido de maneira unânime sobre um dos valores propostos previamente.

Duas primitivas definem também formalmente o problema de consenso:

- ***Propose(v):*** Um processo *p* propõe ao seu grupo de processos, um valor *v*, obtido de um conjunto de valores *V*.
- ***Decide(v):*** Executado o protocolo de consenso, um valor *v* é retornado por esta primitiva como o valor de decisão de todos os processos corretos do grupo.

Para que a decisão unânime entre os processos participantes possa ser garantida, estas primitivas devem satisfazer às seguintes propriedades:

- **Terminação*:*** Todo processo correto acaba por decidir um valor.
- **Acordo*:*** Se um processo correto decide por um valor *v* então todo processo correto acabará por decidir o mesmo valor *v*.
- **Validade*:*** Se um processo correto decide por um valor *v*, então este valor foi previamente proposto por um processo correto (portanto, estava presente no conjunto de valores *V*).
- **Integridade*:*** Todo processo correto decide uma única vez.

Revisando as primitivas definidas, é possível observar que a propriedade de terminação está relacionada com a vivacidade (*liveness*) (ALPERN; SCHNEIDER, 1985) do algoritmo de consenso, a propriedade de acordo garante que todos os processos corretos decidam por um mesmo valor, e a validade garante que os processos corretos somente decidam por valores propostos. As últimas duas propriedades definem os requisitos de segurança do algoritmo de consenso.

As propriedades de **Acordo** e **Validade Uniformes** (CHARRON-BOST, 2004) descrevem que todos os processos, corretos ou não, devem decidir por um mesmo valor. As decisões podem envolver inclusive valores propostos por processos faltosos que não terminam. Apesar de estas propriedades serem aceitáveis no modelo onde os processos podem sofrer faltas por *crash*, em modelos mais abrangentes onde estes processos estão propensos a sofrerem de faltas arbitrárias (comportamentos maliciosos) não é possível (ou aceitável) tomar decisões sobre valores de componentes maliciosos.

Um grande número de trabalhos já foi proposto como solução do problema de consenso, tanto para o caso onde os processos estão sujeitos a falhas por *crash* (LAMPORT, Leslie, 2001), (ZIELIŃSKI, 2004) assim como onde processos podem apresentar comportamento arbitrário (CASTRO; LISKOV, 2002a; KOTLA *et al.*, 2007), (KIHLSTROM *et al.*, 1997). Um dos trabalhos mais importantes envolvendo o problema do consenso é o apresentado em (FISCHER *et al.*, 1985), onde são delimitadas as condições de solução deste problema. Fischer, Lynch e Paterson apresentam neste trabalho a impossibilidade de se resolver este problema em sistemas distribuídos onde o modelo de sincronismo é totalmente assíncrono. Tal impossibilidade é conhecida na literatura como a condição FLP. Dessa forma, modelos com algum tipo de sincronismo mais real e menos

impeditivo que o assincronismo puro são adotados para contornar esta impossibilidade.

Como citado anteriormente, o problema de consenso é equivalente ao problema de difusão atômica [32], o qual é a base para a replicação Máquinas de Estados, amplamente adotada na implementação de arquiteturas distribuídas tolerantes a intrusão.

2.7 DIVERSIDADE DE PROJETO

Uma das técnicas baseadas em redundâncias que se faz necessária quando tratamos com faltas maliciosas é a diversidade de projeto (AVIZIENIS, Algirdas; KELLY, 1984). A tolerância a intrusões necessita fortemente da independência entre as vulnerabilidades de cada réplica de um serviço. Ou seja, as vulnerabilidades usadas por um atacante para invadir uma réplica de serviço não podem se reproduzir da mesma maneira nas outras réplicas do serviço. Para se conseguir esta independência entre vulnerabilidades de réplicas, é necessário usar a diversidade de projeto onde a replicação do serviço se baseia na construção dos diversos servidores, usando diferentes metodologias e ferramentas no desenvolvimento de cada réplica. O uso desta diversidade minimiza a probabilidade de repetição das mesmas vulnerabilidades nas diferentes réplicas.

A utilização dessa técnica de projeto auxilia o não comprometimento de todas ou de um grande número de réplicas, em um curto espaço de tempo, uma vez que, apesar dos servidores que hospedam a replicação não serem livres de vulnerabilidades, um atacante deverá explorar um grande número de diferentes vulnerabilidades para o comprometimento de todos ou grande parte destes servidores. Esta técnica de diversidade torna, portanto, difícil que os servidores de uma replicação sejam comprometidos em sua maioria. Para que esta replicação não seja totalmente comprometida, cada uma das réplicas deve ser projetada de forma diferente, por exemplo, usando diferentes sistemas operacionais e diferentes formas de programação do algoritmo (LITTLEWOOD; STRIGINI, 2004), (BESSANI *et al.*, 2008), (AVIZIENIS, A. *et al.*, 2004).

2.8 VIRTUALIZAÇÃO

A tecnologia de virtualização vem sendo amplamente usada porque torna possível executar várias máquinas virtuais sobre uma única máquina física, otimizando o uso dos recursos físicos dos servidores. Se considerarmos aplicações em sistemas distribuídos, podemos dizer que na máquina física, na maior parte do tempo, a taxa de utilização dos recursos físicos pelos servidores é relativamente pequena, dessa forma os recursos do servidor físico pode ser melhor empregado com a utilização de máquinas virtuais. Em sistemas distribuídos ainda, esta tecnologia aporta novos atributos de flexibilidade, além do isolamento de servidores. A possível migração de máquinas virtuais representa um poderoso instrumento na solução de problemas de desempenho ou ainda de tolerância a falhas.

2.8.1 Conceitos Gerais sobre Virtualização

A virtualização foi inicialmente usada na década de 60 pela IBM Corporation, onde o objetivo era a partição dos recursos físicos dos *mainframes* da época em um grande número de instâncias lógicas a serem utilizadas separadamente. Um dos primeiros estudos sobre essa tecnologia foi apresentado em (ROBERT P GOLDBERT, 1974), onde uma série de conceitos introduzidos neste trabalho é ainda usada nos dia de hoje.

Apesar de o termo virtualização ser amplamente utilizado nos dias atuais para se referenciar a utilização de máquinas virtuais em maquinas físicas, o termo pode se referir em uma série de outras formas de virtualização, tal como virtualização de hardware (SAHOO; MOHAPATRA; LATH, 2010a), virtualização em nível de software, dentre outras. Neste trabalho, quando citarmos o termo virtualização, estamos exclusivamente falando sobre virtualização de sistema operacional (NANDA; CHIUEH, 2005).

Alguns termos são amplamente utilizados quando é empregado a virtualização de sistemas operacionais, tais como:

- **Máquina Virtual** (*Virtual Machine*)**:** Instância de uma máquina virtual que executa um sistema operacional próprio. Cada máquina física pode conter várias instâncias de máquinas virtuais.

Usualmente máquinas virtuais são chamadas de *sistemas convidados (guest)*.

- **Monitor de Máquina Virtual** (*Virtual Machine Monitor*): Chamado também de VMM ou *hypervisor* (BARHAM *et al.*, 2003), é normalmente uma camada que fica situada entre o hardware do sistema hospedeiro e as máquinas virtuais. Trata-se de uma camada importante, pois tem a função de gerenciar e dar suporte a várias execuções de máquinas virtuais.

- **Sistema Hospedeiro:** O Sistema hospedeiro (*host*) é o sistema físico sobre o qual o VMM assim como as máquinas virtuais são executadas.

Para cumprir adequadamente com as suas funções de gerenciamento dos diversos ambientes virtuais construídos sobre o mesmo ambiente físico, é necessário que este VMM procure atender os seguintes atributos (NANDA; CHIUEH, 2005; POPEK; GOLDBERG, 1974):

- **Eficiência**: Na procura do bom desempenho, o VMM deve permitir o acesso direto a instruções do hardware físico às maquinas virtuais desde que estas instruções não atentem contra a segurança e a correção do conjunto.

- **Controle de Acesso a Recursos Físicos**: O VMM deve ter controle completo sobre o hardware. Quando as máquinas virtuais necessitam de acesso ao hardware compartilhado, a demanda de acesso deve ser verificada segundo políticas e roteada via VMM.

- **Equivalência**: Qualquer sistema e/ou aplicação sendo executado em máquinas virtuais deve funcionar de maneira indistinguível ao caso onde não exista a virtualização e o VMM.

Outro atributo de extrema importância é o isolamento (CHUN, B. *et al.*, 2008; FRASER, K *et al.*, 2004; ROSENBLUM, 2004). O VMM deve oferecer isolamento de forma que as máquinas virtuais não interfiram ou tenham acesso a recursos das demais máquinas virtuais. Este isolamento também inclui o isolamento dos recursos do próprio VMM.

2.8.2 Tipos de Virtualização

Quando se fala sobre virtualização, diferentes tipos de arquiteturas são definidas. Cada uma delas apresenta diferentes camadas de abstração na implementação de máquinas virtuais sobre o VMM. As arquiteturas existentes são classificadas, segundo suas estratificações, como apresentado abaixo (SAHOO; MOHAPATRA; LATH, 2010a):

- **Virtualização Total:** Nesta abordagem, ilustrada na Figura 2.1, o VMM é executado sobre o sistema operacional do sistema hospedeiro, no mesmo nível de aplicações usuário. Todo o hardware oferecido pela VMM é simulado e as máquinas virtuais não tem qualquer acesso direto ao hardware do sistema hospedeiro. Este tipo de virtualização tem como vantagem a sua fácil utilização. Em contrapartida, o número de camadas necessárias na abordagem faz com que seu desempenho seja menos eficiente.

Figura 2.1 - Virtualização Total

- **Paravirtualização:** Na paravirtualização, Figura 2.2, as camadas existentes entre a máquina virtual e o sistema hospedeiro são diminuídas, de forma que o VMM seja executado diretamente sobre o hardware. Dentre as principais características da paravirtualização está a simplicidade do VMM e o fato de parte do hardware físico poder ser acessado diretamente por máquinas virtuais. Tais características fazem com que o desempenho das máquinas virtuais paravirtualizadas seja próximo ao desempenho da execução de um sistema diretamente sobre o hardware. Porém a existência da virtualização deixa de ser transparente para que sistemas (aplicações em máquinas virtuais) possam fazer o uso direto dos recursos físicos oferecidos pelo hospedeiro, de forma

que os sistema operacionais devam ser modificados para este tipo de virtualização.

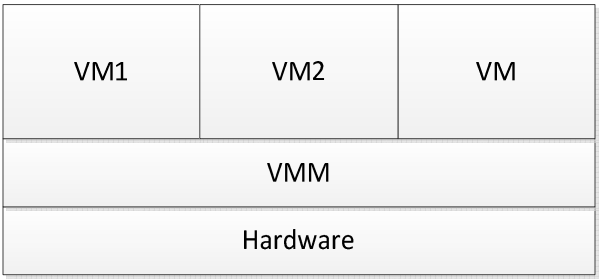

Figura 2.2 – Paravirtualização

- **Virtualização em nível de Sistema Operacional:** É um tipo de virtualização denominada *virtualização baseada em container*. Nesta abordagem, ilustrada na Figura 2.3, o núcleo do sistema operacional da máquina hospedeiro é compartilhado com as máquinas virtuais, ou seja, as máquinas virtuais usam o mesmo núcleo de sistema operacional do sistema hospedeiro. O fato de usar um único núcleo faz com que o desempenho do sistema seja aumentado. Porém, uma grande desvantagem deste tipo de abordagem é que os sistemas operacionais executados nas máquinas virtuais devem ser o mesmo executado na máquina física.

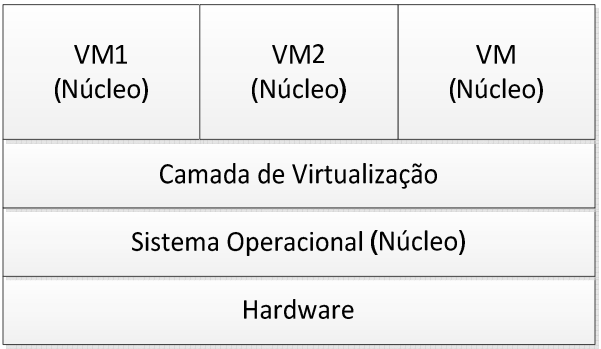

Figura 2.3 – Virtualização em nível de Sistema Operacional

2.8.3 Considerações sobre Virtualização

A tecnologia de virtualização vem sendo um tópico de grande estudo atualmente. Suas aplicações vêm crescendo, devida suas vantagens, tais como a flexibilidade, onde as máquinas virtuais sendo executadas podem ser facilmente gerenciadas, e a disponibilidade permitida pela facilidade com que máquinas virtuais podem ser transportadas para diferentes servidores físicos com um tempo de *downtime* extremamente pequeno (CLARK *et al.*, 2005). A escalabilidade é também um atributo facilitado com o uso da virtualização. Neste caso, os recursos (virtuais) mantidos disponíveis para aplicações através de máquinas virtuais podem ser aumentados facilmente de forma dinâmica. Porém, uma das características mais vantajosas no uso da virtualização é a economia no uso do hardware, pois os recursos de hardware podem ser melhor utilizados quando particionados entre diferentes aplicações construídas sobre recursos virtuais.

Em nosso trabalho fazemos uso extensivo da tecnologia de virtualização, mais especificamente da paravirtualização. Nossas réplicas de servidores são construídas sobre máquinas virtuais. Devido à dinâmica de nosso algoritmo de ME tolerante a intrusão, este servidores (máquinas virtuais) podem ser adicionados ou retirados na execução da replicação ativa, uma prática que teria um grande custo caso estivéssemos executando cada uma de nossas réplicas em máquinas físicas próprias.

2.9 CONSIDERAÇÕES FINAIS

Este capítulo apresentou os conceitos básicos de sistemas distribuídos, segurança e virtualização, sobre quais o trabalho desta dissertação está fundamentado. Estes conceitos serão utilizados nos demais capítulos, e, quando necessário, serão revisados ou revisitados para uma melhor compreensão. Na primeira seção do capítulo foram apresentados os principais conceitos referentes à segurança de funcionamento. Apresentamos também como os componentes faltosos de um sistema distribuído se comportam.

Os modelos de sistemas com suas premissas de sincronização também foram objeto de discussão neste capítulo. As abstrações usuais de sistemas distribuídos são canais e processos. Apresentamos premissas

de sincronização normalmente consideradas para processos e canais em sistemas distribuídos.

Na seção 2.4, nos concentramos nos aspectos referentes à tolerância a intrusões (tolerância a faltas bizantinas). Introduzimos a técnica de replicação Máquina de Estados que é usada na construção de sistemas tolerantes a intrusões. Técnicas e mecanismos adicionais foram citados, além da caracterização de problemas que servem de base para a construção de replicações ativas: o consenso e a difusão com ordem total. O objetivo do presente trabalho foi fundamentado em muitos dos trabalhos citados nesta parte do capítulo.

Por fim, apresentamos neste capítulo conceitos sobre virtualização, a tecnologia que foi utilizada no desenvolvimento de protótipos de nossa abordagem.

Nos capítulos subsequentes exploramos a bibliografia relacionada e apresentamos a nossa abordagem para serviços tolerantes a intrusões para sistemas distribuídos em ambientes como a Internet.

3 SERVIÇOS TOLERANTES A INTRUSÕES: TRABALHOS RELACIONADOS

Nos dias de hoje, as dificuldades com intrusões e modificações de comportamento de serviços são uma constante em ambientes como a Internet. Para sobreviver a estas intrusões e às corrupções correspondentes, em muitos *sites* é utilizada a técnica de replicação destes serviços. As técnicas de replicação e, principalmente, a replicação Máquina de Estados (ME) se mantidos os limites de faltas e intrusões, transformam o comportamento dos serviços replicados corretos e disponíveis, mesmo diante destas imperfeições.

Neste capítulo, apresentamos alguns trabalhos com este propósito, ou seja, a continuidade do serviço. Estes trabalhos descrevem o número de servidores necessários para manter um sistema seguro mesmo na presença de intrusões, assim como os passos necessários para que este objetivo seja alcançado. Inicialmente, discutimos alguns algoritmos de BFT (*Byzantine Fault Tolerance*) que servem de suporte para replicações ativas (ME) em ambientes intrusivos. Na sequência, tratamos com experiências práticas que usam estes algoritmos para implementar serviços tolerantes a intrusões. Estes trabalhos são descritos nas seções subsequentes.

3.1 PRACTICAL BYZANTINE FAULT TOLERANCE (PBFT)

O protocolo PBFT (CASTRO; LISKOV, 2002a), introduzido por Castro e Liskov, foi o primeiro trabalho prático que demonstrou a viabilidade da implementação de sistemas tolerantes a faltas bizantinas sendo, portanto, uma referência obrigatória. Este trabalho trata da construção de um protocolo de consenso tolerante a faltas bizantinas e que serve de suporte para uma replicação Máquinas de Estados. Os autores explicitam que, mesmo na ocorrência de réplicas maliciosas, o serviço modelado como uma ME continua em sua execução correta.

O PBFT apresenta no seu modelo de sistema um número não conhecido de clientes, possivelmente infinito, e n réplicas de um serviço onde $n \geq 3f + 1$. A execução correta do sistema é garantida se o número máximo de réplicas com comportamento malicioso for limitado por $f = \left\lfloor \frac{n-1}{3} \right\rfloor$.

Cada uma destas réplicas implementa uma máquina de estados determinista, e também executa a aplicação do serviço oferecido. É assumido um modelo de sistema assíncrono na comunicação entre cliente e réplicas e, réplica-réplicas. Estes canais são também autenticados: as mensagens trocadas utilizam técnica HMAC baseada em criptografia simétrica (STALLINGS, 2006). A adoção dessa técnica ocorre devido ao fato de que, o número de mensagens trocadas pelo protocolo é grande. Com isso, a utilização da técnica de criptográfica assimétrica poderia causar uma sobrecarga de processamento no sistema, devido ao seu elevado custo computacional.

A execução do protocolo é baseada em visões, onde em cada uma destas visões um servidor faz o papel de réplica primária e as demais fazem papel de réplica *backup*. A réplica primária no protocolo tem o papel de definir a ordem com que as requisições recebidas dos clientes serão executadas. Já as réplicas *backup* somente executam as requisições ordenadas. O algoritmo geral é executado em 3 fases: PRE-PREPARE, PREPARE e COMMIT. Em caso de presença de réplica maliciosa, fases adicionais se tornam presentes no protocolo.

No funcionamento normal (sem falhas) do protocolo, ilustrado na Figura 3.1, o cliente *C* envia uma mensagem de requisição <*REQUEST*, *o*, *t*, *c*, Sig_c> onde *o* é a operação a ser executada, *t* é o *timestamp* da requisição usado para garantir a semântica *at-most-once*, assegurando que a requisição será executada uma única vez, *c* é a identificação do cliente.

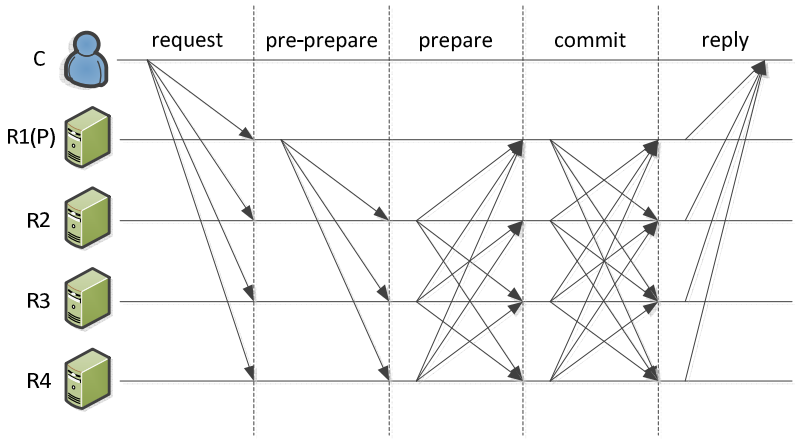

Figura 3.1- Execução PBFT sem falhas

A mensagem é ainda assinada (Sig_c) pelo emissor antes do envio para todas as réplicas. Na recepção, caso essa requisição seja válida (autenticação e *timestamp* válidos), a réplica primária (P) da visão inicia o protocolo de três fases para a ordenação da requisição.

O primeiro passo ocorre com a réplica primária enviando a ordem de execução da requisição através de uma mensagem <*PRE-PREPARE, v, n, D(m),* Sig_p> para todas as réplicas *backup*, onde *v* é a visão atual, *n* é a ordem proposta para a execução, *D(m)* é o resumo criptográfico (*digest D(m) = hash*(*m*)) da mensagem *m* (que é a requisição enviada pelo cliente). O resumo da mensagem enviado pela réplica primária tem como objetivo a identificação da requisição que está sendo ordenada.

Na segunda fase do protocolo, as réplicas ao receberem a mensagem PRE-PREPARE, fazem o envio de uma mensagem <*PREPARE, v, n, D(m), i,* Sig_i> para todas as réplicas, onde *v* é a visão atual, *n* é a ordem proposta para a execução (enviada pela réplica primária), *D(m)* é o resumo da requisição enviada pelo cliente e *i* é a identificação da réplica *backup*. Essa mensagem é assinada pela réplica emissora (Sig_i). A mensagem PREPARE serve como confirmação do recebimento da mensagem PRE-PREPARE, mostrando também a aceitação da ordem proposta pela réplica primária para a requisição do cliente. Para que essa mensagem seja aceita como válida, o número de ordenação proposto não pode ter sido usado em outra requisição, e também cada uma das réplicas deve receber *2f* mensagens PREPARE válidas para a requisição em questão. Para finalizar esta fase, de posse dos certificados de validade (mensagens PREPARE), as réplicas difundem uma mensagem <*COMMIT, v, n, D(m), i,* Sig_i > para todas as réplicas, onde *v* é a visão atual, *n* é a ordem proposta para a execução, *D(m)* é o resumo da mensagem *m* enviada pelo cliente, *i* é a identificação da réplica e Sig_i é a assinatura da réplica *i* emissora do *Commit*.

A terceira fase é finalizada quando as réplicas recebem *2f + 1* mensagens COMMIT válidas vindas de diferentes réplicas, para uma requisição *m* e um mesmo número de visão *v*. A terceira fase tem o objetivo de garantir que todas as réplicas corretas estão de acordo sobre uma mesma ordem de execução para uma dada requisição. Após o COMMIT as réplicas executam a operação enviada pelo cliente e enviam diretamente para este cliente o resultado dessa execução, através de uma mensagem <*REPLY, v, t, c, i, r,* Sig_i>, onde *v* é a visão atual, *t* é o *timestamp* da requisição, *c* é o cliente, *i* a identificação da réplica, *r* o

resultado da operação enviada e Sig_i é a assinatura da réplica *i* que emite a resposta.

Para que a operação seja finalizada, o cliente deve receber pelo menos *f + 1* respostas válidas de diferentes réplicas, com isso garantindo que pelo menos uma mensagem seja correta. Caso o cliente não receba as respostas dentro de um espaço de tempo definido em um temporizador, o cliente retransmite a requisição para todas as réplicas. O cliente efetua retransmissões até que receba o número de respostas válidas suficientes para finalizar a operação.

Como os servidores podem estar sujeitos a apresentar um comportamento arbitrário, um protocolo de troca de visão é executado quando as réplicas *backup* suspeitam que a réplica primária seja maliciosa. Na Figura 3.2 é ilustrada uma execução do protocolo onde ocorre a troca de visão e um novo primário é definido.

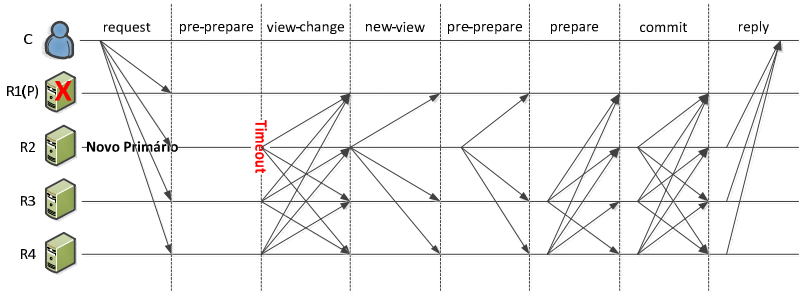

Figura 3.2 - Execução PBFT com troca de primário

As réplicas ao suspeitarem do primário em uma visão *v*, enviam uma mensagem <*VIEW-CHANGE, v + 1, n, C, P, i,* Sig_i>, propondo a troca de visão para uma visão *v + 1*, para todas as réplicas. O parâmetro *n* enviado nessa mensagem corresponde ao número de sequência do último *checkpoint* estável, o parâmetro *C* contém um conjunto de *2f + 1* mensagens de *checkpoint* provando a corretude do último *checkpoint* estável, *P* é o conjunto de mensagens que foram ordenadas após o *checkpoint n* e *i* a identificação da réplica**.**

A nova réplica primária a ser definida por $p = v \bmod |\mathcal{R}|$ (*v* é visão atual e $|\mathcal{R}|$ número de réplicas no protocolo) ao receber *2f* mensagens VIEW-CHANGE válidas de diferentes réplicas, envia uma mensagem <*NEW-VIEW, v + 1, V, O,* Sig_p>, para todas as réplicas, instalando a nova visão a partir da qual ela será a réplica primária. O

parâmetro *V* contém o relatório de todas as mensagens VIEW-CHANGE válidas recebidas de todas as outras réplicas e *O* contém o conjunto de mensagens PREPARE (certificados), vindos no parâmetro *P* das mensagens VIEW-CHANGE. O objetivo do parâmetro *O* é de que, todas as réplicas estejam com uma visão homogênea de todas as requisições que foram ordenadas e executadas por todas as outras réplicas.

Periodicamente o protocolo PBFT executa a coleta de lixo dos *logs* de mensagens. Porém, o protocolo não pode simplesmente descartar as informações, pois algumas dessas informações podem ser necessárias como provas de certificados. Dessa forma, antes de executar a coleta de lixo, o protocolo executa uma operação de *checkpoint*. Nesta operação, as réplicas trocam informações sobre o estado atual da aplicação e esperam que pelo menos *2f + 1* réplicas estejam com um estado igual para que seja criado um certificado de estado estável. Quando todas as réplicas recebem as mensagens de estado necessárias para se obter este certificado, as mesmas podem descartar seus *logs* de mensagens. Além disso, na eleição de um novo coordenador, as réplicas devem enviar este certificado para garantir que, até o presente momento, todas estavam com o mesmo estado da aplicação.

Uma série de otimizações são ainda propostas no trabalho, tal como:

- **Redução do custo de comunicação:** O cliente, no envio da requisição, define que somente uma das réplicas irá enviar a resposta completa da execução da requisição. As demais réplicas simplesmente enviam o resumo da resposta. A validação da resposta é verificada ao se comparar se o resumo da resposta completa recebida é igual aos resumos enviados pelas outras réplicas.
- **Execução provisória:** Nesta otimização, as réplicas executam a requisição exatamente ao receberem o certificado de PREPARE, eliminando a última fase do protocolo, onde é aguardado um certificado COMMIT.
- **Operações Somente-Leitura:** As operações que não alteram o estado da aplicação, como por exemplo, operações somente-leitura, são respondidas aos clientes antes mesmo da execução do protocolo de ordenação. O cliente aguarda o quórum necessário com o mesmo resultado para aceitar a resposta.

- **Ordenação de requisições em lote:** A utilização desta técnica faz com que seja definido um único número de sequência para um conjunto de requisições. Para isso, a réplica primária aguarda até que o número de requisições para um conjunto seja atingido ou o limite de um temporizador tenha expirado, para então iniciar o protocolo de ordenação para esse conjunto de requisições.

Apesar de o protocolo PBFT ser custoso devido ao grande número de trocas de mensagens, a importância deste protocolo é indiscutível, sendo considerado o trabalho seminal que deu origem ao desenvolvimento nos últimos anos em tolerância a faltas maliciosas. A grande parte dos protocolos que surgiram depois da sua publicação não difere muito deste, tendo como modificações o uso de diferentes técnicas, com o objetivo da redução do número de réplicas ou redução nos passos de comunicação.

3.2 ZYZZYVA: Speculative Byzantine Fault Tolerance

Zyzzyva (KOTLA *et al.*, 2007) é um protocolo tolerante a faltas bizantinas que implementa a replicação Máquinas de Estados. Assim como no PBFT (CASTRO; LISKOV, 2002a), o modelo de sistema adotado no Zyzzyva é também composto por clientes e um grupo de servidores. Um destes servidores faz o papel de réplica primária, ou seja, é responsável pela ordenação das requisições vindas dos clientes, enquanto os demais fazem papel de réplicas secundárias, que somente executam requisições e enviam respostas ao cliente. Porém, diferente do PBFT, o protocolo é executado de forma especulativa: as réplicas executam as requisições e enviam a resposta para o cliente sem iniciarem um protocolo de acordo. Ou seja, assumem uma posição otimista confiando na ordem enviada pela réplica primária.

Contudo, com esta execução especulativa, as réplicas podem acabar provocando, em algum momento, inconsistências na sequência de execuções e nos estados das réplicas. Neste caso, o cliente assume uma grande importância no protocolo, sendo o mesmo responsável pela identificação de possíveis inconsistências que podem ocorrer entre as réplicas. Outra diferença entre o Zyzzyva e os demais protocolos BFT é que, neste protocolo, o cliente aguarda $3f + 1$ mensagens de resposta válidas vindas de diferentes réplicas para aceitar o resultado, diferentemente das $f + 1$ mensagens de resposta normalmente adotadas por grande parte dos protocolos.

Na Figura 3.3 é ilustrada a execução do protocolo sem a ocorrência de réplicas maliciosas. O cliente envia a requisição <*REQUEST, o, t, c, Sig*$_c$> para a réplica primária (P), onde *o* é a operação a ser executada, *t* é o *timestamp* da requisição e *c* é a identificação do cliente. A réplica primária, inicialmente verifica se o *timestamp* da requisição recebida (mensagem *m*) é válido, ou seja, igual ou maior ao da última requisição processada que foi enviada pelo mesmo cliente. Com o *timestamp* válido, a réplica primária define um número de sequência (*n*) para a execução da requisição e envia o mesmo para todas as réplicas secundárias, na mensagem de envio de ordem (<*ORDER-REQ, v, n, h*$_n$*, D(m), Sig*$_p$ >). Nesta mensagem, *v* indica a visão atual, *n* é o número de sequência proposto para a requisição *m*, *h*$_n$ é o resumo sumarizando o histórico dos resumos gerados anteriormente e *D(m)* é o resumo da requisição *m*.

As réplicas secundárias, ao receberem uma mensagem ORDER-REQ, executam a requisição sem o devido conhecimento do acordo ou recebimento de tal mensagem pelas demais réplicas.

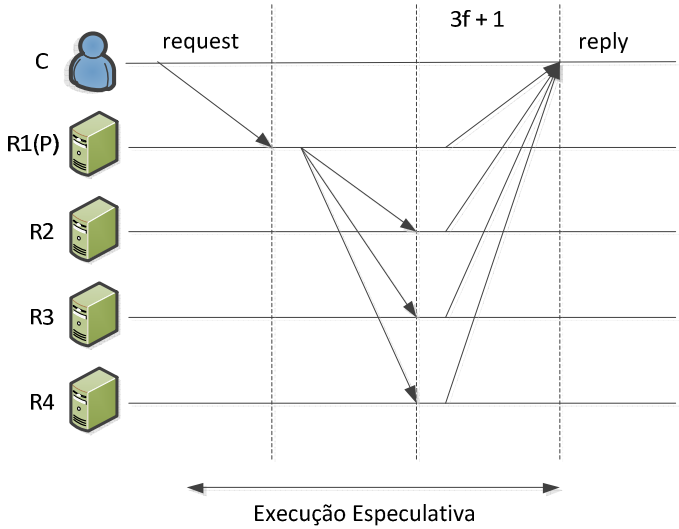

Figura 3.3 - Execução Especulativa Zyzzyva

As réplicas, portanto, executam a operação e na sequência respondem ao cliente enviando a mensagem <*SPEC-RESPONSE, v, n, h*$_n$*, c, t, i, r, H(r),OR, Sig*$_i$>. Nesta mensagem, *i* é a identificação da

réplica, *r* é a resposta obtida na execução da requisição, *H(r)* é o resumo da resposta e *OR* é a mensagem ORDER-REQUEST enviada pelo primário definindo a ordem referente a esta requisição do cliente. A mensagem *OR* anexada na resposta ao cliente serve como um certificado para as verificações do cliente. O cliente, ao receber *3f + 1* respostas válidas de diferentes réplicas, considera a operação completada. Caso esse limite de respostas não seja alcançado, o cliente retransmite a requisição, agora para todas as réplicas, como ilustrado na Figura 3.4.

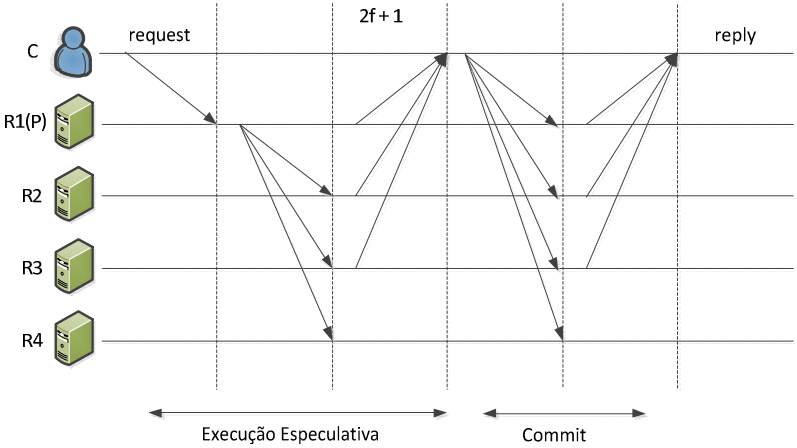

Figura 3.4 – Execução com menos de 3f + 1 respostas para o cliente

As réplicas, ao receberem uma retransmissão, verificam se já executaram a requisição do cliente. Em caso afirmativo, simplesmente reenviam a resposta. Caso a réplica secundária ainda não tenha executado esta requisição, a mesma deve enviar uma mensagem para o primário solicitando a ordem definida para a requisição. Se o primário responder ao pedido de reenvio da ordem da requisição, a réplica que fez a solicitação então dá continuidade a sua atividade no protocolo de forma normal, executando a requisição e respondendo na sequência ao cliente. Porém, se ao final de um temporizador a réplica primária não responder à solicitação desta réplica secundária, uma troca de visão é proposta pela solicitante.

Outra situação que pode ocorrer é a do cliente receber diferentes respostas para uma mesma requisição. Neste caso, o acordo sobre a ordem de execução das requisições não ocorreu corretamente e as réplicas executaram as requisições em uma sequência diferente da

esperada. O cliente então envia para todas as réplicas um relatório com as respostas recebidas. Cada réplica ao receber esse relatório verifica as inconsistências e tentam executar as requisições na ordem correta, se preciso, retrocedendo seus estados.

Como mencionado anteriormente, um protocolo de troca de visão é executado quando as réplicas de *backup* corretas passam a desconfiar que o primário está apresentando comportamento malicioso. Quando isso ocorre, estas réplicas enviam uma mensagem <*I-HATE-THE-PRIMARY, v, i,* Sig_i> para todas as réplicas, onde *v* é a visão atual. Caso a réplica que fez o envio da mensagem de suspeita receba *f + 1* mensagens *I-HATE-THE-PRIMARY*, a mesma faz o envio de uma mensagem de troca de visão anexando como evidências estas *f + 1* respostas de suspeita recebidas, confirmando então a troca do primário. As trocas de visão exigem que os estados das réplicas estejam consistentes. Para manter o estado consistente, periodicamente um protocolo de *checkpoint* é executado.

Este protocolo de *checkpoint* mantém o estado da réplica bem como um *snapshot* do estado da aplicação sendo executada pelas réplicas. Quando uma réplica inicia a execução deste protocolo, a mesma gera resumos criptográficos do estado da réplica e do estado da aplicação e envia a mensagem de *CHECKPOINT* assinada com estas informações para todas as outras réplicas. As réplicas, ao receberem essa mensagem, executam as mesmas operações, enviando para todas as réplicas uma mensagem com seu resumo de estado e aplicação. Um *checkpoint* é considerado estável quando as réplicas recebem *2f +1* mensagens válidas de diferentes réplicas e com resumos iguais para o mesmo *checkpoint*.

As contribuições do protocolo Zyzzyva foram a exploração de um protocolo especulativo, onde se espera que o comportamento da réplica primária esteja correto. Com isto, em execuções de situações normais, o protocolo ganha em eficiência devido a trocas de mensagens menos custosas entre as réplicas, se comparado com o PBFT.

3.3 SEPARANDO O PROCOTOLO DE ACORDO DO PROTOCOLO DE EXECUÇÃO

Em (YIN *et al.*, 2003) é definida uma arquitetura onde processos que executam o serviço de acordo são mantidos separados das réplicas que executam o serviço. Nesta arquitetura, são necessárias *3f + 1*

processos para a execução do chamado serviço de acordo, que é responsável pela ordenação das requisições dos clientes; e *2g+ 1* réplicas de execução do serviço de aplicação (réplicas que executam as requisições ordenadas). Os parâmetros *f* e *g* são, respectivamente, os limites nos dois serviços citados para processos e réplicas que podem apresentar comportamento malicioso. O objetivo da separação entre estes dois grupos de serviços é o fato de que, algoritmos de BFT requerem pelo menos *3f + 1* réplicas para a ordenação (acordo) das requisições, enquanto um simples conjunto de onde se possa extrair uma maioria de respostas iguais é o suficiente para a execução das requisições. Um dos pontos chaves discutidos é o fato de que, réplicas de execução necessitam de um maior poder de processamento enquanto os processos de acordo que executam o protocolo de ordenação utilizam menos poder de processamento, devido às operações simples do protocolo. A restrição das réplicas somente na execução da aplicação (processamento mais pesado) traz melhoras no desempenho do sistema.

Outra vantagem da separação é a possibilidade do conjunto de processos do serviço de acordo ser utilizado por mais de um serviço de replicação ao mesmo tempo, não se restringindo a uma única Máquina de Estados. No artigo os autores denominam os grupos de réplicas do serviços de acordo e de replicação ME de *clusters*: o *cluster* de acordo e o *cluster* de execução.

É ilustrado na Figura 3.5 o algoritmo deste modelo que enfatiza a separação de funções. O cliente começa enviando uma requisição $<REQUEST, o, t, c, Sig_c>$ para o cluster de acordo *A* (Passo 1 - Figura 3.5), contendo a operação a ser executada no campo *o*, o *timestamp* da mensagem no campo *t* e sua identificação no campo *c*. Os processos do *cluster* de acordo A, ao receberem esta requisição, verificam a existência da resposta para essa requisição em seus *caches*. Caso os mesmos tenham a resposta, enviam esta para o cliente e evitam um novo processamento da requisição. Caso não tenha a resposta, as réplicas executam um protocolo BFT de três fases com o objetivo de definir uma sequência de execução (um número de ordem) para a requisição.

Após a execução do protocolo de ordenação, o *cluster* de acordo envia uma mensagem $<COMMIT, v, n, D(m), m, A, Sig_A>$ para as réplicas do cluster E de execução (Passo 2 - Figura 3.5), definindo a ordem para a execução da requisição do cliente, onde cada uma das réplicas assina a sua própria mensagem. Os parâmetros *v* e *n* são, respectivamente, a visão atual e o número de sequência atribuído para a

requisição do cliente, *D(m)* é o resumo da requisição do cliente, e o certificado autenticado por pelo menos *2f + 1* réplicas que estão de acordo com a ordem definidas para a requisição**.**

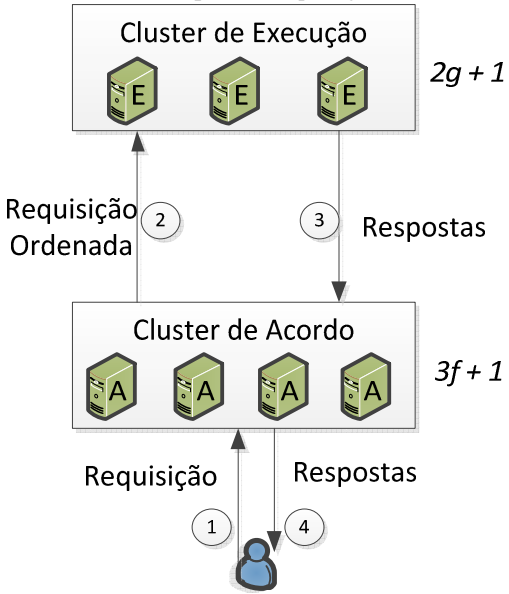

Figura 3.5 – Arquitetura de Separação de Acordo e Execução

Cada réplica do *cluster* de Execução, ao receber uma mensagem de requisição ordenada e certificada, executa a operação e envia sua resposta (<*REPLY, v, n, t, c, r, i, Sig*$_i$>) novamente ao *cluster* de Acordo (Passo 3 - Figura 3.5). Nesta mensagem, o parâmetro *v* é a visão atual, *n* é o número de sequência da requisição, *t* é o *timestamp* enviado na mensagem do cliente *c* e *r* é o resultado da operação. O *cluster* de Acordo aguarda por pelo menos *g + 1* respostas iguais e válidas vindas das réplicas de execução, para então enviar esta como resposta para o cliente (Passo 4 - Figura 3.5). Uma cópia desta mensagem de resposta é também armazenada no *cache* dos processos corretos do *cluster* de Acordo para possíveis retransmissões.

Apesar do custo da implementação deste protocolo ser maior em número de processos (processos de acordo e réplicas de execução) e isto impactar no tempo de resposta devido a um maior número de trocas de mensagens, os autores apontam um aumento na confiabilidade do

sistema (AVIZIENIS, A. *et al.*, 2004), o que pode ser um ponto positivo em sistemas onde a confiabilidade é um requisito mais importante que o desempenho. Um fato que pode atenuar os aspectos de desempenho é que um *cluster* de Acordo pode ser utilizado por vários *clusters* de execução.

3.4 MODELOS HÍBRIDOS DE FALTAS EM AMBIENTES BIZANTINOS

Vários esquemas foram propostos na literatura enfatizando modelos de faltas híbridos em sistemas que atuam em ambientes bizantinos. Estão entre estas técnicas aquelas que enfatizam o uso de componentes confiáveis. Este tipo de componente (confiável) normalmente é visto como seguro ou a prova de invasão, de forma que os mesmos, quando desviam de suas especificações, só possam experimentar comportamentos de falha por *crash*. Tais componentes são normalmente desenvolvidos sobre um hardware especial e possuem seus softwares facilmente verificáveis quanto a suas correções. O fato de usarem hardwares especiais está baseado na ideia de mantê-los isolados de possíveis acessos externos, evitando assim, possíveis ações intrusivas que visam à corrupção dos mesmos.

A inserção desse tipo de componente torna híbrido o modelo de faltas do sistema, de forma que, enquanto componentes confiáveis somente podem falhar por *crash*, o restante do sistema pode apresentar comportamento bizantino. Modelos de sistemas que permitem componentes confiáveis tornam possível o desenvolvimento de algoritmos mais leves, podendo diminuir o número de réplicas necessárias e também o número de passos de comunicação entre estas réplicas na execução de seus protocolos de acordo. Nas próximas seções descrevemos algumas experiências centradas na ideia de elementos confiáveis.

3.4.1 Trusted Timely Computing Base (TTCB)

Em (CORREIA, M. *et al.*, 2002) é apresentada uma arquitetura que faz uso de componentes confiáveis, que no conjunto formam um TTCB (uma base de computação confiável síncrona). O objetivo desta base computacional confiável é permitir protocolos de difusão atômica menos complexos em sistemas assíncronos. A base confiável segundo

os autores forma um canal síncrono e confiável no ambiente assíncrono, permitindo acordos com latências delimitadas.

A inserção de componentes confiáveis faz com que o sistema apresente um modelo híbrido de falhas. No caso da arquitetura TTCB, as réplicas podem apresentar comportamento malicioso, enquanto componentes confiáveis do TTCB, quando falhos, sofrem apenas de *crash* (param). Neste sistema, estes elementos do TTCB se fazem presente em cada uma das réplicas da Máquina de Estados.

Na Figura 3.6 é apresentada a arquitetura do sistema com os seus componentes de TTBC. Cada um dos *hosts* é composto dos componentes usuais (sistema operacional, aplicação, etc) e um LocalTTCB. Esses componentes LocalTTCB são conectados por um canal de comunicação privado e síncrono (Canal de Controle), enquanto as réplicas são conectadas por um canal de comunicação assíncrono.

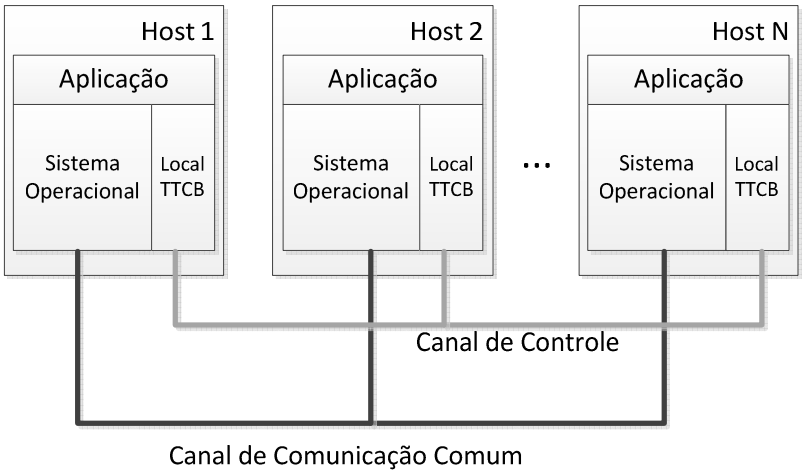

Figura 3.6 – Arquitetura híbrida com uso de TTCB

Essencialmente esse componente confiável oferece dois serviços:

- **Serviço de Autenticação Local:** Possibilita a comunicação segura dos processos de aplicação com o TTCB. Os processos que desejam utilizar o TTCB, primeiramente precisam se autenticar junto ao mesmo. Uma vez concretizada esta

autenticação, o processo recebe credenciais que permitem o uso de serviços do TTCB.

- **Serviço Confiável de Acordo:** Este serviço de acordo permite decisões dos processos participantes sobre valores propostos. São oferecidas três operações para este serviço: *TCB_propose*, onde os processos propõem seus valores para a decisão; *TCB_decide*, que retorna o resultado do acordo sobre os valores propostos; e uma função *decision*, usada para definir a função a ser executada pelos processos participantes na decisão sobre o conjunto de valores propostos.

A principal característica do algoritmo executado pelos servidores é o protocolo de difusão atômica, que deve garantir que, se o emissor é correto, todos os servidores corretos entregam a mensagem na mesma ordem. Para isso os servidores utilizam o TTCB e as funções oferecidas por ele.

O algoritmo segue em rodadas, tendo sempre um dos servidores com o papel de coordenador, o qual envia as propostas (mensagens) para todos os servidores do grupo. Todos os servidores, no recebimento da mensagem enviada pelo coordenador, informam o resumo criptográfico desta em seu TTCBLocal. Este resumo criptográfico é transmitido através do canal confiável para todos os demais servidores. Quando os servidores recebem pelo menos $f + 1$ resumos iguais para dada mensagem, a proposta é aceita e é assumido que o consenso foi atingido.

Caso os servidores não recebam os $f + 1$ resumos, estes retransmitem a mensagem para os servidores que não informaram o resumo. A retransmissão é repetida até que uma confirmação seja recebida ou o grau de omissão definido seja ultrapassado. O grau de omissão é baseado em hipóteses síncronas, onde existe um tempo máximo para que uma resposta seja recebida uma vez que a retransmissão ocorreu. Caso o grau de omissão seja ultrapassado, é considerado que o servidor sofreu uma parada (*crash*).

O artigo apresenta o uso de um componente confiável: o TTCB e seu serviço de difusão atômica. A presença desta base computacional confiável permite a redução do número de réplicas para $n \geq f + 2$. Outro ponto da abordagem a ser observado é o modelo híbrido de falhas apresentado, onde diferentes tipos de falhas ocorrem em componentes distintos.

3.4.2 MINPBFT

MinPBFT (VERONESE *et al.*, 2009) é uma variação similar ao PBFT(CASTRO; LISKOV, 2002a), onde é implementado um serviço com replicação Máquinas de Estados em ambiente bizantino. A ideia fundamental do trabalho é a utilização de um componente confiável denominado *USIG Service (Unique Sequential Identifier Generator)*, cuja finalidade é atribuir um número de sequência e produzir um certificado assinado para as mensagens trocadas no algoritmo. Esse certificado serve como uma prova de que o número de sequência atribuído a uma mensagem não será utilizado mais que uma vez ou mesmo em qualquer outra mensagem. O uso deste componente confiável permite o desenvolvimento de algoritmos que fazem uso de *2f + 1* réplicas; o que corresponde a uma redução se comparado com os algoritmos convencionais, como o PBFT, que necessitam de pelo menos *3f + 1* réplicas (CASTRO; LISKOV, 2002a), (KOTLA *et al.*, 2007).

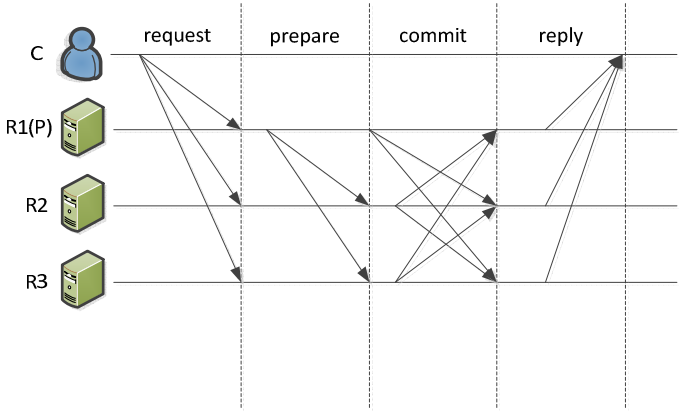

Figura 3.7 – Execução normal do protocolo MinPBFT

É ilustrado na Figura 3.7 a execução do protocolo em caso da operação normal (sem ocorrência de réplicas maliciosas).

A operação é iniciada como o envio da requisição assinada $<REQUEST, c, seq_c, op, Sig_c>$ pelo cliente (C), onde *c* é a identificação do cliente, seq_c é o número de sequência da requisição dado pelo ambiente do cliente e *op* é a operação a ser executada pelas réplicas. A réplica primária (P) atribui um número de sequência global para esta requisição em mensagem $<PREPARE, v, s_i, D(m), UI_i, Sig_i>$ enviada

para todos os outros servidores do grupo. Os parâmetros presentes nesta mensagem são: *v* que é a visão atual; s_i é a identificação do servidor; *D*(*m*) é o resumo da mensagem enviada pelo cliente; e UI_i é o identificador único da requisição. Tanto o resumo da mensagem (*D(m))* quanto a identificação única da mensagem (UI_i) são criados pelo *USIG Service*. A ordem de execução da requisição é dada pelo campo UI_i gerado pela réplica primária.

Cada um dos servidores, ao receber uma mensagem PREPARE válida do primário, faz o envio de uma mensagem de <*COMMIT, v,* s_j, s_i, *m,* UI_i, UI_j>, atestando o recebimento da mensagem. Além dos servidores utilizarem na mensagem COMMIT parâmetros vindos na mensagem PREPARE enviado pelo primário, ainda adicionam a própria identidade de servidor s_j e seu identificador único UI_j. Dessa forma, uma mensagem COMMIT é composta pela identificação única do servidor que criou a requisição (PREPARE) e definiu a ordem de execução e o identificador único do servidor que aceitou a ordem proposta. Os servidores, ao receberem *f + 1* mensagens de COMMIT válidas vindas de diferentes servidores, aceitam e executam a requisição.

Após a execução da requisição, os servidores respondem diretamente ao cliente através de uma mensagem <*REPLY, s, Seq, res*>. O cliente completa a operação ao receber *f + 1* respostas válidas de diferentes servidores e com o mesmo valor de resposta (*res*).

O componente confiável *USIG Service* oferece duas operações: a operação de criação de um identificador único para uma mensagem, através do comando *createUI(m)*, onde o parâmetro é a mensagem para a qual se deseja obter um identificador único, e a operação de verificação da validade da mensagem, através do comando *verifyUI(PK, UI, m)*, onde é verificado se o identificador único (UI) para uma mensagem *m* é válido dada uma chave pública *PK*. O uso do componente confiável garante que servidores maliciosos não forjem numerações. Mensagens com números de identificação corrompidos (UI) são descartados.

Além do protocolo de ordenação, o sistema possui protocolos para troca de visão (troca de primário), de *checkpoint* e de coleta de lixo. Da mesma forma que na execução normal do algoritmo de ordenação, as mensagens de *checkpoint* ou de troca de visão são certificadas pelo *USIG Service*. Os servidores aceitam um *checkpoint* como estável (ou uma troca de visão) quando recebem *f + 1* mensagens de *checkpoint* (ou

de VIEW-CHANGE) válidas. Tendo um *checkpoint* válido recebido, mensagens antigas são descartadas, efetuando assim a coleta de lixo.

O uso do componente *USIG* permite a diminuição do número de réplicas (de $3f + 1$ para $2f + 1$) e dos passos de comunicação do algoritmo, se comparado com o PBFT.

3.5 SERVIÇOS BFT USANDO VIRTUALIZAÇÃO

A tecnologia de virtualização vem sendo largamente empregada para redução do número de servidores físicos. Com esta tecnologia, um servidor físico pode executar vários servidores virtuais, economizando com o hardware e aproveitando melhor seus recursos.

Um dos maiores custos quando se trabalha com replicação Máquinas de Estados é o fato de que o serviço deve ser executado em várias máquinas físicas (diferentes *hardwares*). O isolamento aportado pelos diferentes *hardwares* contribui para a independência do modo de falha ou intrusão. A tecnologia de virtualização contribui para este isolamento fazendo o uso de menos *hardware*. O uso da virtualização permite que cada uma das réplicas possa ser executada em uma máquina virtual sobre uma única máquina física, por exemplo. Além dessa economia no número de servidores físicos, as ferramentas de virtualização oferecem facilidades para o gerenciamento destas máquinas virtuais. Iniciar ou desligar máquinas virtuais são operações simples quando da utilização de tais ferramentas, o que permite também que protocolos tolerantes a faltas possam ser recuperados na disponibilidade máxima de suas réplicas, tratando réplicas corrompidas de forma dinâmica. Réplicas são facilmente ativadas, desativadas ou mesmo adicionadas no sistema. Com a divisão em camadas na construção de protocolos em ambientes virtualizados, sempre será possível a utilização de elementos confiáveis, o que torna o modelo de falhas do sistema híbrido.

Abaixo seguem trabalhos que fazem uso da virtualização na construção de seus protocolos.

3.5.1 VM-FIT: Supporting Intrusion Tolerance with Virtualization Technology

VM-FIT (REISER, Hans P, 2007) foi uma das primeiras arquiteturas tolerante a faltas bizantinas a utilizar a virtualização e, como consequência, provendo também componentes confiáveis. Em VM-FIT, uma das funções desse componente confiável é a comunicação com a rede externa e a comunicação com os demais servidores participantes do grupo de réplicas de serviço. A arquitetura utiliza a terminologia inspirada no *hypervisor Xen*(XEN, 2011), onde cada réplica (máquina virtual) constitui um domínio. Dessa forma, cada servidor físico é composto: por máquinas virtuais que executam réplicas do serviço; pelo *domínio 0* (*Domain0*) que é o domínio responsável pelo controle e execução das máquinas virtuais; e por um domínio particular da arquitetura VM-FIT denominado de *domínio NV* (*Network/Voting*) que trata da comunicação com os clientes, comunicação entre as réplicas e também é o local onde se encontra a função responsável pela votação aplicada sobre as respostas das réplicas de execução.

A interação entre o cliente e o serviço é interceptada pelo domínio NV, o qual recebe as requisições do cliente e as distribui para todas as réplicas através de um serviço de comunicação em grupo. Cada réplica executa e envia a resposta para a réplica remetente da requisição através do domínio NV, que então seleciona a resposta correta através de uma função de maioria e envia esta resposta para o cliente.

Uma das vantagens dessa arquitetura é a transparência da replicação: o cliente envia a requisição para somente um servidor e recebe uma resposta desse mesmo servidor, não tendo conhecimento de estar se comunicando com um serviço replicado. Outra vantagem citada pelos autores é que os componentes estão expostos a diferentes tipos de faltas: os domínios convidados (máquinas virtuais) são passíveis de falhas bizantinas, enquanto os domínios NV podem falhar somente por *crash*. Uma das justificativas deste modelo híbrido de falhas é que os diferentes componentes (domínios) estão sujeitos a diferentes complexidades e acessos. Os domínios convidados são compostos de sistemas operacionais legados os quais executam todos os códigos da aplicação, o que os torna complexos e mais propensos a falhas arbitrárias (sujeitos a corrupções por intrusões). Domínios especiais como o *domínio NV* e *domínio 0* são totalmente isolados e suas funcionalidades podem ter seus códigos facilmente verificados devido a sua simplicidade.

Outra característica apresentada na arquitetura é a utilização da recuperação proativa, de forma que as réplicas possam ser periodicamente rejuvenescidas com uma nova cópia do sistema, aumentando a disponibilidade e o tempo de vida da replicação ME.

Técnicas de recuperação proativa foram apresentadas anteriormente em vários protocolos mais tradicionais que não fazem uso de virtualização(CASTRO; LISKOV, 2002b)(CASTRO; LISKOV, 2000)(ISHIKAWA *et al.*, 2005). Porém, nestes casos, estas técnicas se mostram complexas. Estas recuperações proativas rejuvenesciam máquinas físicas e eram baseadas em hardwares especiais. Com a tecnologia de virtualização o rejuvenescimento se torna mais simples. No caso da proposta em análise, uma vez que o *domínio 0* tem total controle sobre as máquinas virtuais, essa operação de recuperação proativa se torna mais rápida e eficiente, além de não ter custo com hardware adicional.

O VM-FIT possui um modelo de sistema que foi aplicado em duas arquiteturas distintas. Uma destas experiências é o RESH (REISER, H. *et al.*, 2006) (ilustrada na Figura 3.8) que usa somente uma máquina física executando todas as máquinas virtuais que fornecem o serviço replicado ao cliente, enquanto a segunda REMH (REISER, H.P.; KAPITZA, 2007), Figura 3.9, utiliza uma máquina física para cada réplica de serviço e demais componentes, os quais se comunicam através de um canal privado, utilizando um protocolo de comunicação em grupo. Enquanto em RESH uma falha por *crash* no servidor físico torna o serviço indisponível, REMH tem certa tolerância caso uma das máquinas dos servidores sofra *crash*.

Em ambas as arquiteturas, o número de réplicas necessárias para execução do serviço tolerante a falhas é $n \geq 2f + 1$ para tolerar f réplicas maliciosas, onde essas réplicas são físicas em REMH e máquinas virtuais em RESH.

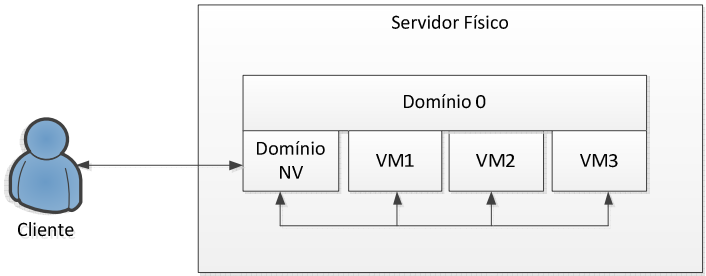

Figura 3.8 – Arquitetura RESH

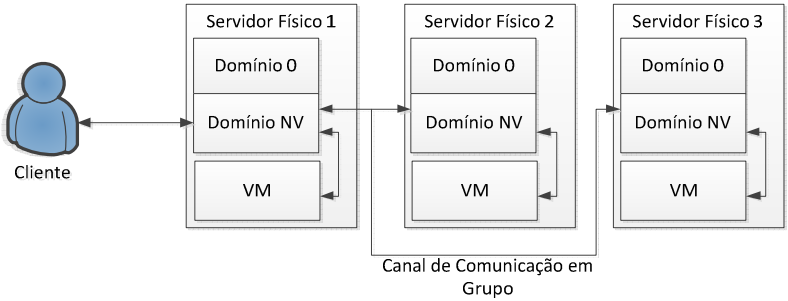

Figura 3.9 – Arquitetura REMH

VM-FIT foi um dos primeiros trabalhos a explorar o uso da virtualização no desenvolvimento de sistemas tolerantes a faltas bizantinas. Os autores ainda apresentam a possibilidade da utilização da tecnologia para o desenvolvimento de componente confiáveis baseados em software, criando domínios confiáveis para trabalhar como tais componentes. Apesar do uso da virtualização neste modelo, esta não apresentou redução nos custos com relação ao número de réplicas necessárias, tanto réplicas virtuais (RESH) ou mesmo em réplicas físicas (REMH).

3.5.2 Diverse Replication For Single-Machine Byzantine-Fault Tolerance (LBFT)

Em LBFT (CHUN, B. *et al.*, 2008) é apresentada uma discussão sobre as novas possibilidades e desafios do emprego da virtualização para protocolos tolerantes a faltas bizantinas. O uso da virtualização dá a possiblidade de um melhor aproveitamento dos recursos de processamento dos hardwares atuais.

O artigo trata dos desafios de se conseguir a independência de falhas com o uso do isolamento e da diversidade de projeto de forma que a falha (ou intrusão) em uma máquina virtual não seja propagada para as demais máquinas virtuais. Para isso, é necessário que essas máquinas virtuais estejam totalmente isoladas e que os sistemas operacionais executados sobre cada uma dessas máquinas virtuais sejam diversos, de forma que uma mesma falha (ou vulnerabilidade) em um servidor não possa ser explorada nos demais.

Os autores abordam três pontos principais no desenvolvimento de sistemas tolerantes a faltas bizantinas utilizando virtualização. O primeiro ponto discute a criação de um coordenador confiável. Esse coordenador somente oferece funções simples, como por exemplo, ordenação e votação. O mesmo é mantido isolado, constituindo o que poderíamos considerar uma TCB (*Trusting Computing Base*). Como já citamos anteriormente, todo o componente confiável dá a possiblidade da criação de protocolos tolerantes a faltas maliciosas bem mais simples que o PBFT.

Outro aspecto abordado pelos autores é a questão da sincronia. Se for assumido que, a memória e os dispositivos de entrada e saída do hardware que executam máquinas virtuais são livres de falha, e que as réplicas mantêm a mesma velocidade, independentemente da diversidade de projeto, então toda comunicação entre as réplicas executadas em máquinas virtuais diferentes, mas dentro de uma mesma máquina física, pode ser tratada como síncrona. Assim como no caso de empregar um coordenador confiável, caso a comunicação entre as réplicas possa ser considerada como síncrona, protocolos mais simples podem ser desenvolvidos.

Por fim, é abordada a questão da transparência da replicação vista pelos clientes. Nos sistemas replicados tradicionais, o cliente interage com várias réplicas. Com o uso de um coordenador para a ordenação e votação, o mesmo pode coletar as respostas dos servidores e enviar diretamente para o cliente somente a resposta correta, dando a ilusão para o cliente que o mesmo está se comunicando com um serviço não replicado. Essa transparência também dá a possiblidade de diminuição da utilização de banda entre o cliente e o servidor, visto que o mesmo ocorre só com o envio da requisição e o recebimento de uma resposta.

Diante das características discutidas foram propostos dois protocolos, denominados LBFT (*Ligthweight BFT*).

O primeiro, LBFT1, Figura 3.10, é assumido um coordenador confiável (*Coordinator Core*), assíncrono (os tempos de computação desta entidade não são conhecidos) e uma replicação não transparente formada com *2f+1* réplicas de execução. A execução do protocolo segue com o cliente enviando uma requisição para o coordenador confiável. Esse, por sua vez, adiciona um número de sequência para essa requisição e difunde para todas as réplicas de execução. As réplicas executam a requisição e enviam a resposta diretamente para o cliente, que aguarda um número mínimo de respostas iguais para completar a

operação. Neste caso, o cliente sabe que está interagindo com um serviço replicado. Somente as réplicas e os clientes executam operações criptográficas. Vale a pena salientar que este coordenador deve ser implementado em uma base de computação confiável (TCB).

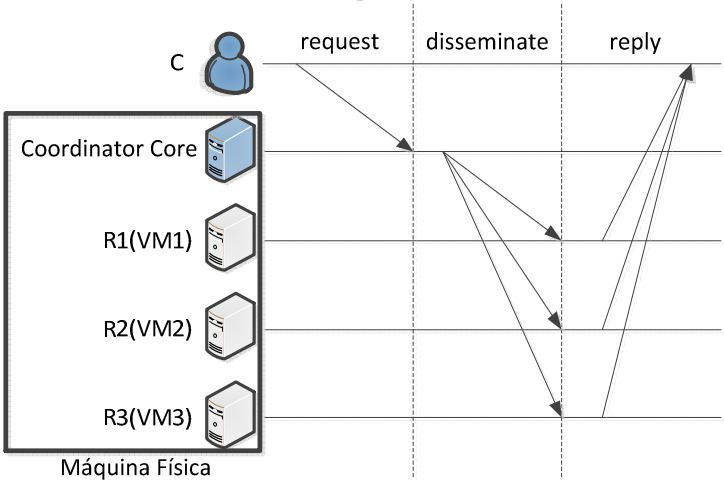

Figura 3.10 – LBFT1 – Serviço Não Transparente

No protocolo LBFT2, Figura 3.11, é também assumido um coordenador confiável, com a diferença que as réplicas não enviam a resposta diretamente para o cliente. Além de definir um número de sequência para as requisições recebidas, este coordenador recebe as respostas vindas das réplicas, verifica se o autenticador da mensagem é válido e aguarda até que sejam coletadas $f + 1$ respostas iguais vindas dessas réplicas, para então assinar e encaminhar a resposta para o cliente. Neste caso, o serviço visto pelo cliente é transparente, pois o mesmo recebe somente uma resposta. Esse protocolo também diminui o número de operações criptográficas nas trocas de mensagens.

Os estudos propostos não tiveram uma experimentação prática e são muito econômicos em detalhes. O objetivo do texto era mostrar novas possibilidades com o emprego da virtualização sobre somente uma máquina física.

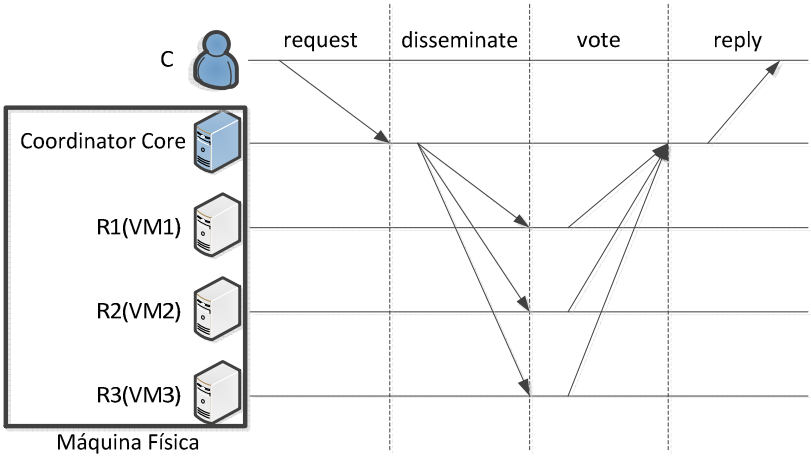

Figura 3.11 – LBFT2 – Serviço Transparente

3.5.3 ZZ and the Art of Practical BFT Execution

ZZ (WOOD *et al.*, 2011) é uma arquitetura tolerante a falhas bizantinas que utiliza a tecnologia de virtualização para execução dos servidores e aplica o conceito de separação do protocolo de acordo e execução, definido por Yin (YIN *et al.*, 2003). Dessa forma, a arquitetura é composta por dois tipos de processos: as $f+1$ réplicas de execução que mantém o estado da aplicação e executam as requisições dos clientes; e os $3g+1$ réplicas de acordo que definem uma ordem para as requisições recebidas dos clientes. Estes processos e réplicas podem falhar independentemente, sendo definidos limites separados para cada um desses tipos de servidores (acordo e execução).

O ZZ usa a tecnologia de virtualização para a execução dos diversos processos em máquinas virtuais, porém, para manter um requisito de independência de faltas do protocolo proposto, não é executado mais que um processo de acordo e uma réplica de execução em um servidor físico. A separação entre processos de acordo e réplicas de execução, como mencionado na seção 3.3, permite no ZZ a redução do número de réplicas de execução para $f + 1$ na execução do protocolo sem a presença de réplicas maliciosas. Quando réplicas são comprometidas, o número de réplicas de execução é incrementado para $2f + 1$.

A redução para *f + 1* réplicas de execução é possível em períodos livres de falhas, quando nenhuma das réplicas foi invadida e comprometida. Quando ocorre um desacordo nas *f + 1* respostas vindas dessas réplicas temos uma indicação que existem réplicas maliciosas entre as réplicas de execução. Nesse caso, ZZ inicia novas *f* máquinas virtuais (*f* réplicas que estão em *standby*) para formar um novo quórum que agora assume uma replicação total de *2f+1*. Para garantir uma maioria de respostas corretas ao cliente (*f+1*), precisamos somar estas novas *f* réplicas ativadas a partir do desacordo inicial. Por exemplo, se definirmos o valor de *f* = 1, o protocolo deve inicialmente ser executado com *n = 2* (*n = f+1*) réplicas de execução.

No caso onde não há falhas, se as duas réplicas executarem uma requisição ordenada e apresentarem a mesma resposta o protocolo pode prosseguir normalmente. Porém, no caso onde uma dessas duas réplicas é maliciosa não é possível saber qual das respostas é a correta. Então, uma terceira réplica é iniciada para resolver esse impasse. Nesse período de falha, o número de réplicas é incrementado para *2f + 1*, retornando depois para *f + 1*, desligando a réplica que apresentou comportamento malicioso e resposta incorreta. Na execução do protocolo, clientes enviam a requisição para o cluster de acordo, com o objetivo de atribuir um número de sequência para essa requisição. Definida a ordem de execução para essa requisição, os servidores do cluster de acordo enviam essa para os servidores do cluster de execução, seguindo os mesmos princípios de (YIN *et al.*, 2003).

As réplicas de execução executam uma requisição se receberem um conjunto de pelo menos *2g + 1* mensagens válidas enviadas do cluster de acordo. Essa execução irá gerar uma resposta, que será enviada para o cliente e também um relatório para o cluster de acordo.

O relatório enviado para o cluster de acordo contém um resumo criptográfico da resposta assim como os objetos alterados durante a execução dessa requisição com seus resumos criptográficos. Os objetos alterados correspondem às alterações feitas no estado do sistema (por exemplo, arquivos alterados em um sistema de arquivos, registros em um banco de dados).

Se o cliente não recebe as *f +1* respostas esperadas (em *match*) das réplicas de execução, o mesmo reenvia a requisição para o *cluster* de acordo. Os processos deste *cluster*, se já tiverem recebido o relatório dessa requisição das réplicas de execução (se as réplicas já tiverem executado a requisição e enviado *f + 1* respostas em *match*), respondem

para o cliente com este relatório de execução. Se o cliente receber *g + 1* resumos do cluster de acordo e tiver pelo menos uma resposta que corresponde a esse resumo criptográfico, este pode aceitar essa resposta como válida.

As réplicas armazenam em seus *logs* os relatórios de execução das requisições que são utilizadas para o envio no caso de retransmissões de clientes. Assim como em grande parte dos protocolos BFT, é necessário que as informações antigas sejam descartadas para evitar o problema do *buffer* infinito. Para que os servidores possam fazer a coleta de lixo de seus *logs* de mensagens, periodicamente é executada uma rotina de *checkpoint*. Essa operação de *checkpoint* é feita pelos processos do *cluster* de execução. Um *checkpoint* é dito como estável se uma réplica de execução recebe um certificado de *checkpoint*, contendo *f + 1* resumos criptográficos iguais para um mesmo *checkpoint*, vindos de diferentes réplicas. Recebendo um *checkpoint* estável, essa réplica pode descartar os *checkpoints* feitos anteriormente e efetuar a coleta de lixo de seus *logs*. Além disso, este relatório de *checkpoint* estável é enviado para o cluster de acordo.

O cluster de acordo é responsável pela detecção de falhas em ZZ. No caso normal, as réplicas de acordo aguardam por *f + 1* relatórios de execução correspondentes, vindos das réplicas de execução. Quando os processos do cluster de acordo não recebem essas respostas dentro de um tempo pré-determinado ou, recebem os *f + 1* relatórios, porém esses não são iguais, os processos do cluster de acordo enviam uma mensagem de recuperação para o *hypervisor* que controla as réplicas (máquinas virtuais) que estão em *standby*. Quando o *hypervisor* de uma réplica que está em *standby* recebe *g + 1* mensagens pedindo ativação da mesma, este inicia esta nova réplica.

A arquitetura ZZ apresenta o efeito do uso da tecnologia de virtualização, o que torna a manipulação das réplicas mais dinâmica. O fato da detecção das réplicas maliciosas e a respectiva remoção das mesmas faz com que o sistema permaneça sempre com réplicas de execução corretas. Apesar disso, os custos com as réplicas ainda continuam um pouco elevados, pois, como é definido um limite de no máximo uma réplica de acordo e uma de execução por servidor físico, caso se utilize um protocolo de acordo baseado no PBFT, seriam necessários pelo menos *3f + 1* servidores físicos.

3.5.4 SMIT: Shared Memory based Intrusion Tolerance

SMIT (JÚNIOR *et al.*, 2010) é uma arquitetura para tolerância a faltas bizantinas usando máquinas virtuais. A arquitetura é composta por clientes e servidores que se comunicam através da rede local. Esses servidores por sua vez são executados em máquinas virtuais sobre uma única máquina física. Um dos diferenciais dessa arquitetura, além de executar os servidores em máquina virtuais, é o fato de que a arquitetura utiliza uma área de memória compartilhada, denominada caixa postal (*postbox*), através da qual os servidores se comunicam, não necessitando da rede de comunicação para esse fim. Essa memória compartilhada é oferecida pelo sistema anfitrião, o qual é descrito como o sistema que fica sobre as máquinas virtuais, podendo ser representado pelo monitor de máquinas virtuais (VMM) seguindo os conceitos de virtualização. Outro diferencial dessa arquitetura é o número de servidores utilizados para prover o serviço, reduzindo o número de $3f + 1$ da literatura original para $2f + 1$ na arquitetura apresentada devido ao uso de conceito de componente confiável, que neste trabalho, é representado por uma memória compartilhada.

Para isso, ao invés de utilizar um protocolo de consenso com trocas de mensagens via rede, a arquitetura faz uso dessa área de memória compartilhada para prover a funcionalidade de ordenação das requisições, que serão executadas pelos servidores. Duas operações estão disponíveis para o acesso a essa área de memória. A operação *read*, que retorna um valor anteriormente escrito na memória compartilhada e a operação *append*, que insere um valor na memória compartilhada. Uma propriedade importante da memória compartilhada é que a mesma opera em modo *append-only*, ou seja, uma vez que um valor é inserido na memória o mesmo não pode ser alterado, evitando que uma réplica maliciosa forneça valores diferentes para serem lidos pelas demais réplicas.

O algoritmo segue de forma que um desses servidores faça o papel de réplica primária, a qual vai definir uma ordem para a requisição, enquanto os outros façam o papel de réplica secundária (*backup*), que são as réplicas que somente irão executar uma requisição ordenada. Os clientes enviam suas requisições para todas as réplicas. A réplica primária, por sua vez, escreve na memória compartilhada a sua proposta de ordem para a requisição recebida do cliente. As demais

réplicas leem esse valor de proposição de ordem da memória e executam a requisição correspondente.

Uma vez que todas as réplicas leram a ordem da memória compartilhada e executaram a requisição, cada uma delas envia a resposta para o cliente. O cliente, por sua vez, aguarda que pelo menos *f + 1* mensagens com respostas válidas iguais, vindas de réplicas diferentes, para finalizar a operação. Como essa arquitetura utiliza uma das réplicas como sendo a réplica primária, responsável pela ordenação, um protocolo de troca de visão se torna necessário, pois essa réplica pode apresentar comportamento malicioso, não informando um número de sequência para as requisições recebidas, fazendo com que a execução do sistema não evolua.

Ao receber uma mensagem do cliente, as réplicas iniciam um temporizador. Caso esse temporizador expire e as réplicas ainda não tenham encontrado uma proposta de ordem para essa requisição na memória compartilhada, as mesmas escrevem uma mensagem de *view-change* na memória. Quando as réplicas corretas leem pelo menos *f + 1* mensagens de *view-change* via memória compartilhada, um novo primário é eleito e o protocolo pode seguir sua execução normal.

A arquitetura SMIT apresenta o uso de memória compartilhada e virtualização na construção de uma arquitetura tolerante a faltas bizantinas. A utilização da memória compartilhada como componente confiável faz com que haja uma redução no número de mensagens trocadas pelo algoritmo. Outra contribuição foi o desenvolvimento de um algoritmo de consenso que possibilitou a redução do número de réplicas para *2f + 1*.

3.6 CONSIDERAÇÕES FINAIS

Este capítulo teve como objetivo uma revisão da literatura sobre tolerância a faltas. Primeiramente buscamos apresentar os trabalhos clássicos da literatura, os quais são base para a grande maioria dos trabalhos sobre tolerância a faltas e tolerância a intrusão apresentados atualmente. Apresentamos também trabalhos onde o uso de componentes confiáveis mostra novas possibilidades no desenvolvimento de algoritmos de consenso, com reduções no número de réplicas e reduções em passos de comunicações. Estas reduções possibilitam a criação de protocolos mais simples, mais rápidos e mais leves, visto que quanto maior o número de trocas de mensagens, mais

operações criptográficas são executadas em grande parte dos algoritmos. Por fim, apresentamos trabalhos mais recentes que fazem uso da tecnologia de virtualização como ferramenta de apoio para construção de sistemas tolerantes a faltas. A utilização desta tecnologia oferece novas possibilidades, facilitando a criação de protocolos mais dinâmicos e também reduzindo o número de servidores físicos necessários para prover um serviço seguro. Esta tecnologia também permite, com o uso de isolamento, que se construa a ideia de componentes confiáveis.

A Tabela 3.1 resume uma comparação dos protocolos descritos neste capítulo sobre vários aspectos. Alguns destes aspectos não facilitam as comparações destes protocolos, mas das discussões apresentadas fica evidente que números de réplicas e passos de um protocolo são métricas que podem ser usadas na comparação. Na tabela aparecem os termos **Rép. Aco.** que corresponde a processos do serviço de acordo no protocolo considerado. **Rép. Exec.** considera o número de réplicas de execução no protocolo. A coluna **Passos N.** consideram o número de passos (trocas de mensagens) necessários para a execução do protocolo sem a ocorrência de falhas e **Passos F.** consideram o número de passos (trocas de mensagens) necessários para a execução do protocolo na presença de réplicas maliciosas **.Comp. Conf.** consideram o uso de componentes confiáveis e **Virt.** considera o uso da tecnologia de virtualização na construção do protocolo.

Da tabela pode-se concluir que os custos maiores estão com o PBFT. Este usa o maior número de réplicas ($3f+1$) e não faz a separação de acordo e execução, além de ter um grande número de passos. O Zyzzyva na sua abordagem otimista, dispensado o acordo na numeração proposta consegue em situações sem falha o melhor desempenho (1 passo de protocolo). Propostas que separam o acordo da replicação ME, embora possuam mais processos envolvidos, possuem a ME sustentada por menos réplicas de execução; é o caso da proposta de Yin. Os melhores resultados são esquemas que usam elementos confiáveis e virtualização (VM-FIT e SMIT).

Tabela 3.1 - Tabela comparativa dos trabalhos relacionados

Protocolo	Rép. Aco	Rép. Exec	Passos N.	Passos F.	Comp. Com.	Virt.
PBFT (CASTRO; LISKOV, 2002a)	*3f + 1*	*3f + 1*	3	6	Não	Não
Zyzzyva (KOTLA *et al.*, 2007)	*2f + 1*	*3f + 1*	1	6	Não	Não
Yin (YIN *et al.*, 2003)	*2f +1*	*3f + 1*	3	6	Não	Não
TTCB (CORREIA, M. *et al.*, 2002)	*f + 2*	*2f + 1*	NA	NA	Sim	Não
MinBFT(VERONESE *et al.*, 2009)	*2f + 1*	*2f + 1*	2	4	Sim	Não
VM-FIT(REISER, Hans P, 2007)	*2f + 1*	*2f + 1*	2	2	Sim	Sim
LBFT (CHUN, B. *et al.*, 2008)	*2f + 1*	*2f + 1*	2	2	Não	Sim
ZZ (WOOD *et al.*, 2011)	*3f + 1*	*f + 1*	3	6	Não	Sim
SMIT (JÚNIOR *et al.*, 2010)	*2f + 1*	*2f + 1*	2	4	Sim	Sim

4 UMA ABORDAGEM PARA TOLERÂNCIA A INTRUSÕES USANDO MÁQUINAS VIRTUAIS

Vários trabalhos com o objetivo do desenvolvimento de sistemas ou serviços tolerantes a faltas bizantinas têm sido publicados na última década. As abordagens introduzidas nestes trabalhos são baseadas em replicações ativas (Máquinas de Estados) que mascaram réplicas maliciosas, mantendo o funcionamento correto do sistema mesmo na presença de intrusões.

Com o intuito de obter uma abordagem para serviços que sobrevivam a intrusões a um custo reduzido, tanto no quesito de número de passos de comunicação quanto no número de réplicas necessárias, é proposta neste trabalho uma abordagem baseada em replicação Máquina de Estados (ME) para tolerância a faltas bizantinas. Esta abordagem faz uso de virtualização e do conceito de componentes confiáveis que são citados em outros trabalhos sobre replicação ME (REISER, H.P.; KAPITZA, 2007), (REISER, H. *et al.*, 2006), (REISER, Hans P, 2007). A abordagem proposta, através do uso de um componente confiável, é caracterizada por um modelo de falhas híbrido que permite replicação Máquinas de Estados para ambientes bizantinos com número de réplicas menores do que o usual (*3f+1*).

Os pontos chave da abordagem proposta são o uso de um componente confiável (*Agreement Service*) e o uso de uma abstração de memória compartilhada. Esta memória compartilhada provê um mecanismo simples e menos custoso de comunicação entre este componente confiável e as réplicas de execução da abordagem.

Este componente confiável executa a função de um protocolo de acordo e coordena as réplicas da ME em execução na abordagem. Assim como em (WOOD *et al.*, 2011), um dos diferenciais de nossa abordagem é a necessidade de somente $f + 1$ réplicas para a execução do protocolo quando da ausência de réplicas maliciosas.

Nossa abordagem apresenta uma dinâmica no que diz respeito ao número de réplicas de execução da ME. A dinâmica citada consiste em adicionar ou remover réplicas de execução no protocolo conforme as necessidades de quórum na execução de requisições do cliente. A adição de novas réplicas ocorre quando há a detecção de réplicas maliciosas e o quórum de $f + 1$ se torna insuficiente para resolver possíveis desacordos entre as respostas emitidas para a execução de uma requisição de cliente.

As remoções de réplicas ocorrem na detecção da réplica maliciosa e também para o retorno da replicação a *f + 1* réplicas. A abordagem desenvolvida é denominada neste texto de ITVM ("*Intrusion Tolerance using Virtual Machines*").

4.1 MODELO DE SISTEMA

Nosso modelo é composto por dois grupos de processos que interagem através de comunicação pela rede: os servidores e os clientes. Também chamados de réplicas de execução, o conjunto de servidores, S= {S_0, S_1,..,S_{n-1}} é formado por *n* processos com a mesma funcionalidade, executando a replicação Máquina de Estados. Os clientes são processos que compõem o conjunto C= {C_0, C_1,..C_m}, que enviam requisições para os servidores através da rede. Neste modelo de sistema assumimos um comportamento *parcialmente síncrono* (DWORK *et al.*, 1988), ou seja, as interações entre processos clientes e o serviço no sistema distribuído admitem intervalos de assincronismo intercalados por períodos estáveis (síncronos). Estes períodos síncronos permitem a terminação destas comunicações em prazos de tempo definidos, porém não conhecidos. Os canais de comunicação são também assumidos no modelo como confiáveis[2].

Além disso, as réplicas de execução se comunicam com o serviço de acordo denominado *Agreement Service* (AS) exclusivamente através de uma abstração de memória compartilhada (*Shared Memory*). Uma visão geral do modelo de interação entre os processos é ilustrado na Figura 4.1.

No modelo, processos (clientes e servidores) são classificados como corretos ou faltosos. Processos corretos se comportam segundo as especificações do algoritmo, enquanto processos faltosos podem se comportar de maneira arbitrária, desviando de suas especificações, podendo omitir mensagens, parar de responder, modificar o conteúdo das mensagens, e fazer conluio com outros processos faltosos no sentido de dificultar a evolução correta da aplicação. Este tipo de comportamento é associado na literatura a faltas arbitrárias, bizantinas ou maliciosas (intrusões).

---

[2] Canais de comunicação são ditos *confiáveis* quando mensagens não são criadas ou duplicadas e cada mensagem enviada por um processo P_i terminará por ser recebida pelo receptor P_j se o emissor (P_i) não for faltoso ou malicioso (RAYNAL, 2005)

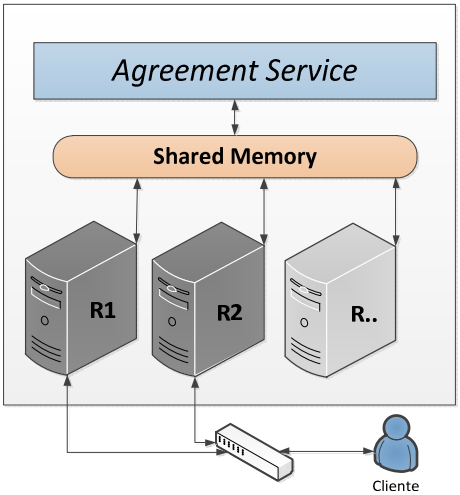

Figura 4.1 – Modelo de Interação entre os Processos

Para garantir a autenticidade das mensagens trocadas entre processos usamos a técnica de criptografia assimétrica. A verificação das assinaturas envolve também o envio de certificados das chaves públicas dos assinantes nas mensagens trocadas. A certificação das chaves públicas das entidades participantes não precisa, necessariamente, envolver ACs (Autoridades Certificadoras) oficiais e reconhecidas. Estes certificados podem ser emitidos por entidades como o comitê gestor (ou o administrador) do serviço replicado que passa a ser entendido pelos participantes das comunicações como confiável. Estas assinaturas envolvem também *hashes*, *nonces* e outros mecanismos usuais adotados em assinaturas digitais, mas que não aparecem explicitamente nos algoritmos apresentados neste trabalho.

Na adoção desta técnica criptográfica foi levado em consideração que o número de mensagens trocadas que necessitam de criptografia em nossos algoritmos é pequeno, portanto, o custo com operações criptográficas é reduzido. Caso o número de requisições que necessitassem de criptografia fosse grande, o uso desta técnica se tornaria inviável na prática.

Além de técnicas de criptografia assimétrica para manter a segurança do sistema, usamos técnicas de diversidade de software (AVIZIENIS, Algirdas; KELLY, 1984), (LITTLEWOOD; STRIGINI, 2004), (BESSANI *et al.*, 2008) na construção do nosso modelo e na execução das réplicas. O uso de diversidade garante uma maior

independência de falhas, de modo que, as réplicas de execução são construídas sobre sistemas operacionais distintos, sendo que tais sistemas não compartilham as mesmas vulnerabilidades. Logo, a corrupção de uma réplica não implica diretamente no comportamento malicioso das demais réplicas do sistema e impede que esta mesma vulnerabilidade seja explorada em todos os servidores.

4.2 ABORDAGEM DE REPLICAÇÃO DO MODELO

As soluções para problemas de BFT ("*Byzantine Fault Tolerance*") sempre tomam como base a técnica de replicação Máquina de Estado (SCHNEIDER, 1990). Em nossa proposta, na submissão de uma requisição de cliente, a implementação desta técnica inicialmente envolve um procedimento otimista com o uso de somente $f + 1$ réplicas respondendo a requisição. Porém, na detecção de um eventual desacordo entre as $f + 1$ respostas, novas réplicas são ativadas, aumentando o número de respostas das réplicas para $2f + 1$. O valor de f corresponde ao limite de faltas ou intrusões admitidas no sistema distribuído de modo a não comprometer a corretude do mesmo.

Nossa abordagem de BFT envolve também a identificação e substituição das réplicas corrompidas que provocaram a inconsistência (divergência de respostas) nas trocas com o cliente.

4.3 UTILIZAÇÃO DE VIRTUALIZAÇÃO NO MODELO

Conforme já citado, o objetivo deste trabalho é dar suporte para serviços de modo que os mesmos sobrevivam a intrusões. No modelo de replicação, as réplicas invadidas e corrompidas não devem comprometer o atendimento das requisições pelo serviço replicado. Para atender a estes objetivos, a nossa abordagem de BFT é então construída através do uso da tecnologia de virtualização: os servidores são implementados e executados separadamente, usando diferentes máquinas virtuais.

Como nos fixamos no contexto de faltas maliciosas envolvendo réplicas de execução comprometidas por intrusões, definimos o mapeamento das diversas máquinas virtuais que executam as réplicas em uma única máquina física servidora. Este enfoque implica em um modelo híbrido de faltas que separa os diferentes tipos falhas. Ou seja, em nível de máquinas virtuais tratamos exclusivamente faltas bizantinas (maliciosas). O problema de faltas de parada (*crash faults*) na máquina

física servidora pode ser tratado com protocolos mais brandos em nível de sistema distribuído. Neste trabalho, nos limitamos à discussão de nossas soluções para faltas maliciosas nas máquinas virtuais.

O uso da tecnologia de virtualização na implementação da nossa abordagem de BFT também oferece facilidades para o gerenciamento das diferentes réplicas do modelo. Como colocado anteriormente, novas réplicas podem ser ativadas quando do desacordo das $f + 1$ respostas a uma requisição de cliente e, também, réplicas identificadas como corrompidas devem ser excluídas do modelo. Nesta tecnologia, máquinas virtuais podem ser facilmente iniciadas ou revogadas quando necessário.

Diante do exposto, nosso modelo é composto, em relação ao ambiente de virtualização, por dois níveis de estratificação: o primeiro, o **sistema hospedeiro** que é representado por um *hypervisor* e/ou um monitor de máquina virtual (*VMM - Virtual Machine Monitor*), e o **sistema convidado** que em nosso caso é formado pelos servidores executados sobre as máquinas virtuais dispostas sobre o servidor físico, por vez chamadas de réplicas, réplicas de execução ou mesmo servidores. Dependendo do contexto, as máquinas virtuais também são denominadas domínios (XEN, 2011).

Em nosso modelo, assumimos que o *hypervisor* e o VMM podem apresentar vulnerabilidades, porém estas não podem ser exploradas externamente via rede ou pelas máquinas virtuais locais. Para garantir essa suposição, confiamos que o VMM forneça o isolamento adequado para a execução segura das máquinas virtuais. Esta é uma das premissas da tecnologia de virtualização (CHUN, B. *et al.*, 2008), (SAHOO; MOHAPATRA; LATH, 2010b). Além disso, é necessário que o *hypervisor* e o VMM sejam externamente inacessíveis, ou seja, entidades externas não podem ter acesso direto a estes níveis de sistema. Para isso, o *hypervisor* desabilita os protocolos de rede para estes níveis. Apesar de o sistema hospedeiro ficar inacessível, nada impede que o VMM garanta o acesso das máquinas virtuais à rede. Desta forma, os clientes podem se comunicar com as máquinas virtuais e suas réplicas de execução.

4.4 UTILIZAÇÃO DE MEMÓRIA COMPARTILHADA NO MODELO

Uma abstração de memória compartilhada é definida no modelo como um conjunto de registradores que armazenam valores. Esta memória pode ser acessada por dois tipos de operações: escrita e leitura. Processos (réplicas e serviço de acordo) utilizam estes registradores para se comunicarem (CACHIN *et al.*, 2006). Essa memória compartilhada é oferecida e gerenciada pelo VMM e se trata de uma memória compartilhada local, que utiliza os registradores de memória da máquina física para sua implementação. Políticas de acesso são definidas para que réplicas de execução tenham acesso de leitura e escrita em áreas próprias desta memória, ou seja, cada máquina virtual só pode escrever em sua área particular para envio (nos *buffers* $reqBuffer_{Rj}$, $repBuffer_{Rj}$ e $hashBuffer_{Rj}$). Já as leituras destas máquinas virtuais (réplicas) concorrem sobre os *buffers* de leitura (*buffer_toExecute*, *buffer_toSend*, *buffer_stableCheckpoint* e *buffer_updateRange*), que tem como escritor um serviço de suporte: o *Agreement Service*. As políticas definidas para esta memória compartilhada são necessárias para garantir a integridade das requisições de clientes já ordenadas e das respostas correspondentes, de forma que réplicas maliciosas não possam modificar os valores escritos em registradores particulares de outros processos. Na Figura 4.2 são ilustrados os mecanismos de memória compartilhada considerando seus aspectos funcionais na comunicação[3].

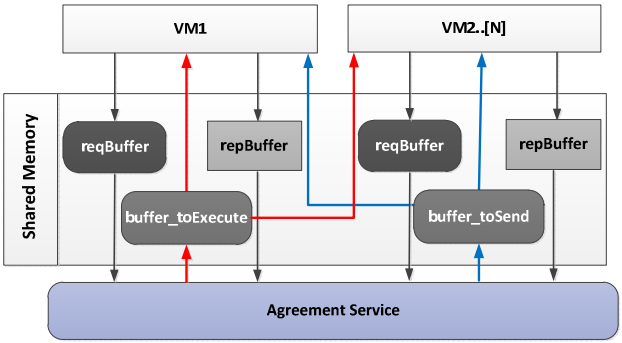

Figura 4.2 – Mecanismo de Comunicação através de Memória Compartilhada

[3] Foram suprimidos "*buffers*" na apresentação da imagem, porém estes mantêm o mesmo formato e as mesmas politicas de acesso dos *buffers* apresentados nesta seção.

A correção nos acessos de memória compartilhada necessita de garantias de *liveness*. Em (FERNANDEZ *et al.*, 2007) é introduzida à propriedade *AWB1,* que define que um processo não se comporta indefinidamente como uma execução assíncrona. Como consequência, a memória compartilhada por processos sempre permitirá acessos em tempo limitado, mas não necessariamente conhecido. Assumimos com isto que a memória compartilhada do modelo proposto também possui um comportamento parcialmente síncrono.

Qualquer arcabouço de memória compartilhada (Memória Compartilhada, Memória Compartilhada Distribuída, etc) que ofereças as garantias citadas pode ser utilizada em nosso modelo. A memória compartilhada em nosso modelo é basicamente um meio pelo qual processos se comunicam (Réplicas e Componente Confiável), não sendo um componente trivial na abordagem.

4.5 AGREEMENT SERVICE

Um componente fundamental no nosso modelo é o *Agreement Service* (AS). Este serviço de suporte é responsável pela ordenação das requisições, verificação das respostas das réplicas de execução e geração e assinatura da resposta que será enviada para o cliente. Além disso, o AS coordena a validação dos *checkpoints* feitos pelas réplicas de execução e a ativação de novas réplicas em caso de desacordo nas respostas ou *checkpoints*. Ou seja, este componente trata de aspectos não funcionais, de forma que pode ser usado em qualquer tipo de aplicação sem modificações.

A inserção deste componente faz com que nosso modelo também possa ser considerado como uma abordagem de separação dos serviços de ordenação e de execução da aplicação, a exemplo do trabalho descrito em (YIN *et al.*, 2003). Desta forma, as réplicas que executam a aplicação ficam expostas aos clientes via rede, enquanto o *Agreement Service* trata da ordenação total das requisições. Devido a sua importância para o funcionamento de nossa abordagem, este componente é mantido de maneira segura e isolada, por isso em grande parte deste texto, o definimos como componente confiável.

O *Agreement Service* é construído na mesma camada de virtualização onde o VMM é executado. A escolha de manter este componente confiável neste nível de estratificação se deve ao fato de

que a tecnologia de virtualização oferece isolamento em relação aos domínios que executam as máquinas virtuais e o VMM.

Além do isolamento garantido, a inserção do *Agreement Service* neste nível possibilita que o mesmo gerencie o funcionamento das máquinas virtuais, visto que, com o decorrer do andamento do protocolo, novas réplicas podem ser iniciadas ou revogadas em caso de desacordo nas respostas.

4.6 PROTOCOLO ITVM

O protocolo desenvolvido neste trabalho, denominado ITVM (*Intrusion Tolerance using Virtual Machines*) será descrito nas subseções seguintes. Primeiramente são apresentados as execução do protocolo sem a presença de falhas (modo otimista) e com a presença de falhas. São descritos também os algoritmos de cada uma das entidades envolvidas no protocolo.

4.6.1 Execução Normal do Protocolo ITVM

Quando da não presença de réplicas maliciosas no sistema, o protocolo ITVM faz sua execução em modo normal, o que definimos como modo otimista. No modo otimista, são necessárias somente $f + 1$ réplicas para a execução do protocolo. Na Figura 4.3 é ilustrada a execução do protocolo.

O protocolo inicia com o cliente *C* enviando uma requisição para todas as réplicas de execução (*R1* e *R2*). As réplicas de execução, após o recebimento da requisição, escrevem esta na memória compartilhada (*Shared Memory*) (Passo 1 – Figura 4.3). Escrita a requisição na memória compartilhada, o *Agreement Service* faz então a leitura e ordenação desta requisição e a escreve novamente na memória compartilhada (Passo 2 - Figura 4.3).

As réplicas, ao lerem da memória compartilhada a requisição ordenada pelo *Agreement Service*, executam a operação enviada pelo cliente e inserem o resultado (resposta) da execução na memória compartilhada (Passo 3 - Figura 4.3).

Após a inserção das respostas na memória compartilhada, o *Agreement Service* verifica se existem $f + 1$ respostas iguais para a requisição e em caso afirmativo, cria uma resposta assinada e a disponibilizada na memória compartilhada (Passo 4 - Figura 4.3). Os servidores então leem esta resposta assinada e a enviam para o cliente.

Para que a operação seja finalizada, o cliente aguarda o recebimento de pelo menos uma resposta vinda de qualquer réplica de execução e com assinatura válida efetuada pelo *Agreement Service*.

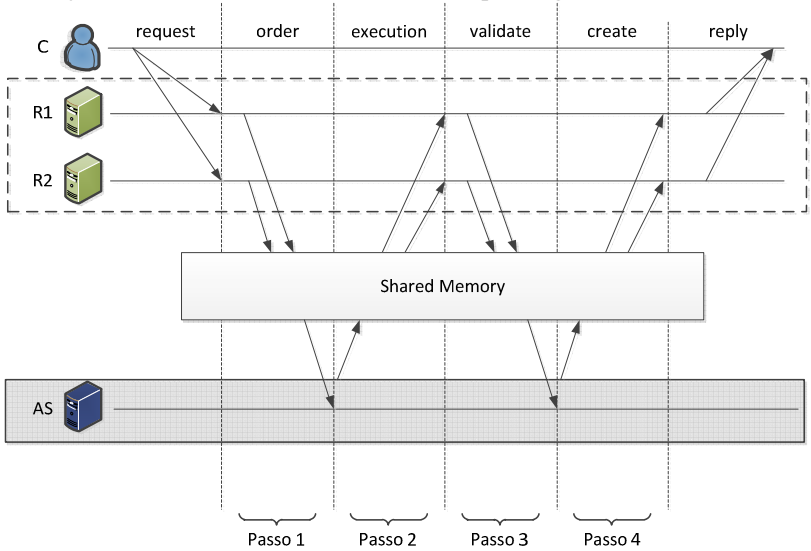

Figura 4.3 - Execução Otimista do protocolo ITVM

4.6.2 Execução do Protocolo ITVM na presença de Réplicas Maliciosas

Como assumimos que as réplicas podem apresentar comportamento malicioso, o sistema deve estar preparado para lidar com tal situação. É ilustrado na Figura 4.4 a execução do protocolo na presença de réplicas maliciosas e a estratégia do protocolo para resolver tal situação.

Assim como no modo otimista, o protocolo é iniciado com o envio das requisições pelo cliente *C* para todas as réplicas de execução ativas (*R1* e *R2*), que escreverão esta requisição na memória compartilhada (*Shared Memory*) e aguardarão a ordenação desta pelo *Agreement Service* (Passo 1 - Figura 4.4). Porém, neste caso, após a leitura da requisição ordenada e a execução da operação (Passo 2 - Figura 4.4), a réplica maliciosa (*R2*) escreve um valor (resposta) diferente do esperado pela execução do algoritmo. Ocorrerá então o que

chamamos de desacordo entre as respostas das réplicas de execução (Passo 3 - Figura 4.4).

O desacordo nos valores das *f + 1* respostas emitidas pelas réplicas de execução torna impossível a criação de uma resposta correta para ser enviada para o cliente, pois, em nosso protocolo, é necessário que o número de respostas iguais apresente frequência de pelo menos *f +1*.

Para resolver o desacordo ocorrido diante de diferentes respostas lidas da memória compartilhada, o *Agreement Service* inicializa *f* novas réplicas (*R3*), aumentando a replicação para *2f + 1* réplicas em execução no protocolo. Estas novas réplicas iniciadas deverão recuperar o estado do sistema (cópia dos dados da aplicação sendo executada) para poder responder corretamente as requisições do cliente (Passo 4 - Figura 4.4). Após a recuperação do estado do sistema, essa nova réplica (*R3*) executa a requisição em que ocorreu o desacordo (Passo 5 - Figura 4.4) e escreve sua resposta na memória compartilhada. A recuperação da réplica será discutido mais detalhadamente na seção 4.6.4.3.

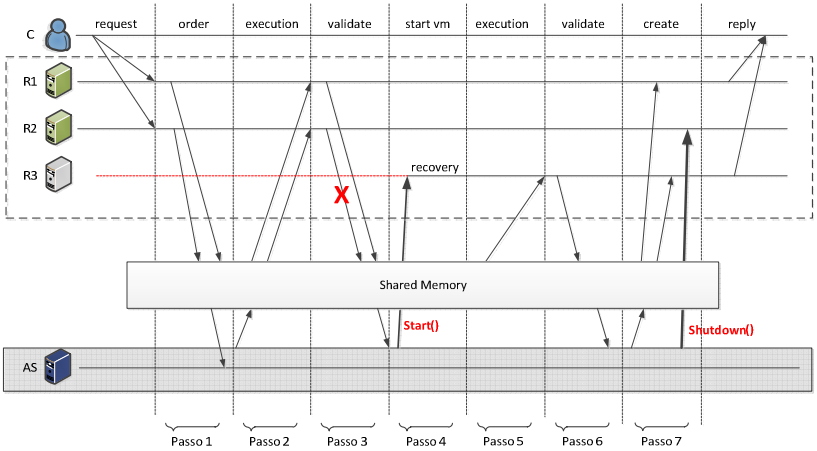

Figura 4.4 - Execução do protocolo ITVM na presença de réplicas maliciosas

O *Agreement Service*, agora de posse de *2f + 1* respostas (R1, R2, R3), pode identificar *f + 1* (quórum mínimo) respostas iguais das réplicas corretas, validar a resposta (Passo 6 - Figura 4.4) e então criar a resposta assinada. Após este passo, as réplicas corretas seguem a

especificação do algoritmo, lendo a resposta assinada da memória compartilhada e enviando para o cliente(Passo 7 e Passo8 - Figura 4.4).

Como a resposta correta foi identificada (a resposta que apresentou *f + 1* ocorrências dentro do conjunto das *2f + 1* respostas lidas), as réplicas que inseriram a resposta incorreta na memória compartilhada podem ser identificadas e suas máquinas virtuais desligadas (Passo 8 - Figura 4.4), retornando para *f + 1* o número de réplicas em execução no protocolo.

4.6.3 Comportamento do Cliente no Protocolo

O cliente no protocolo ITVM interage com o sistema replicado enviando requisições para todas as réplicas de execução através de uma mensagem com o formato <*REQUEST*, $id_{Ci,}$ *op*, *t*, $Sign_{Ci}$, $Cert_{Ci}$> (Algoritmo 4.1 – Linha 7), onde $id_{Ci,}$ representa a identificação do cliente, *op* é a operação a ser executada pelas réplicas, *t* é o *timestamp* da mensagem, com o objetivo de manter a semântica *exactly-once*, evitando que as réplicas executem uma mesma requisição mais que uma vez. Dessa forma, o cliente incrementa este *timestamp* no envio de uma nova requisição para as réplicas (Algoritmo 4.1 – Linha 6). O campo $Sign_{Ci}$ contém a assinatura da requisição emitida pelo cliente (Algoritmo 4.1 – Linha 4) e $Cert_{Ci}$ contém o certificado necessário para a validação desta requisição. As próprias entidades que assinam as mensagens fazem o envio de seu certificado de chave pública para os destinatários, para que estes possam verificar a validade das mensagens recebidas.

Após o envio da requisição, o cliente aguarda por pelo menos uma mensagem de resposta válida (Algoritmo 4.1 – Linha 10). A mensagem de resposta tem o formato <*REPLY*, id_{Rj}, $id_{Ci,}$, *t*, $resp_k$, $Sign_{AS}$, $Cert_{AS}$>, onde id_{Rj} é a identificação da réplica que fez o envio da resposta para o cliente, $id_{Ci,}$ é a identificação do cliente, *t* o *timestamp* da mensagem enviada pelo cliente e $resp_k$ é a resposta da operação executada pelas réplicas. Os campos $Sign_{AS}$ e $Cert_{AS}$ são, respectivamente, a assinatura e o certificado necessário para a validação desta mensagem, emitidos pelo *Agreement Service.*

Na maioria das abordagens que citamos na seção 3, o cliente necessita receber pelo menos *f + 1* respostas iguais de diferentes réplicas para aceitar a mensagem como válida. Em nossa abordagem, o recebimento de uma mensagem de resposta válida (com assinatura e certificados válidos, emitidos pelo *Agreement Service*) já é suficiente

para que o cliente possa finalizar a operação (Algoritmo 4.1 – Linhas 12 e 13). Isto se deve ao fato de que, em nossa abordagem, o *Agreement Service* é tratado como componente confiável e à prova de intrusões, dessa forma mensagens recebidas com sua assinatura e certificado são vistas como corretas. Mensagens com assinatura incorreta ou vindas de outras entidades senão do *Agreement Service*, assim como mensagens antigas ou duplicadas são automaticamente descartadas pelo cliente (Algoritmo 4.1 – Linhas 16 e 19).

Algoritmo 4.1. Protocolo Cliente C_i

```
Variables
1:  t = 0;                                              //Inicializa o timestamp
2:  δc = Δ;                                             //Define o valor de timeout como Δ
Init:
3:  procedure send (op)
4:      SignCi ← Signature(<REQUEST, idCi,t, op >);     //Cria a assinatura da requisição
5:      reqk ← <REQUEST, idCi, op, SignCi, CertCi>;     //Cria a requisição a ser enviada
6:      t = t +1;                                       //Incrementa o timestamp
7:      send reqk to all Rj;                            //Envia para todas as réplicas
8:      timeoutc = t( ) + δc;                           //Temporizador para chegada das respostas
9:  while (waiting_reply)
10:     upon Receive replyk from Rj do
11:     if (replyk.t == t) & !(delivered( replyk.)) then   //Se timestamp é válido e reply não foi entregue
12:         if VerifySign(replyk.SignAS , replyk.CertAS )  //Verifica a validade da assinatura
          then
13:             Delivery(replyk.resp);                  //Entrega para a aplicação
14:             timeoutc = Ø ;                          //Encerra o temporizador
15:         else
16:             Discard(replyk);                        //Descarta a requisição
17:         end if
18:     else
19:         Discard(replyk);                            //Descarta a requisição
20:     end if
21: upon t( ) ≥ timeoutc at Ci do                       //Se o temporizador expirar
22:     δc; = 2 * δc;                                   //Dobra o valor do timeout
23:     send reqk to all Rj;                            //Reenvia a requisição para todas as réplicas
24:     timeoutc = t( ) + δc;                           //Define o novo timeout para a requisição
```

Algoritmo 4.1 – Participação do Cliente no Protocolo

Caso o cliente não receba uma resposta dentro de um tempo definido em um temporizador (Algoritmo 4.1 – Linha 21), este retransmite a requisição (Algoritmo 4.1 – Linha 23), duplicando o tempo de espera deste temporizador (Algoritmo 4.1 – Linha 22), assim como a técnica de limiar utilizada em (CASTRO; LISKOV, 2002a).

4.6.4 Comportamento das Réplicas de Execução no Protocolo

Servidores ou réplicas de execução são as entidades que executam as requisições enviadas pelo cliente e mantêm o estado do sistema. Basicamente, as réplicas executam três subprotocolos, que serão descritos abaixo.

4.6.4.1 Execução das Réplicas

As réplicas de execução, ao receberem uma requisição de cliente, verificam se a resposta para esta requisição já não consta em seu *cache* de respostas (Algoritmo 4.2 – Linha 3). Se a resposta for encontrada, a réplica prontamente a envia para o cliente (Algoritmo 4.2 – Linha 4). Caso a resposta para a requisição não tenha sido encontrada em seu *cache*, as réplicas inserem esta requisição na memória compartilhada (*shared_mem)* em seu *buffer* particular (*reqBuffer*) para que seja definida uma ordem para a execução desta (Algoritmo 4.2 – Linha 6). Após a inserção da requisição na memória compartilhada, as réplicas aguardam uma notificação de que o processo de ordenação foi efetuado e leem esta mensagem do *buffer* de leitura de requisições ordenadas (*buffer_toExecute*, Algoritmo 4.2 – Linhas 8 e 9).

Algoritmo 4.2. Protocolo de Execução das Réplicas R_j

```
Variables
1:  replycache = Ø;                                   //Inicializa o cache de respostas
Init:
2:  upon Receive req_k at R_J
3:      if replycache.exists(req_k) then              //Se a requisição já foi executada
4:          send reply_kj to C_i ;                    //Envia a resposta para o cliente
5:      else
6:          shared_mem.reqBuffer_Rj .write (req_k);   //Escreve a requisição na memória
7:      end if                                        //  para que a mesma seja ordenada

8:  upon ReqSignal at R_j do
9:      req_toExec ← shared_mem.buffer_toExecute.read();   //Lê a requisição ordenada
10:     if VerifySign(req_toExec.Sign, req_toExec.Cert ) then   //Verifica a validade da mensagem
11:         resp_k ← thread.execute(req_toExec.op);   //Executa a operação
12:         shared_mem.repBuffer_Rj .write (resp_k);  //Escreve a resposta na memória
13:     end if

14: upon RepSignal  at R_j do
15:     reply_j ← shared_mem.buffer_toSend.read();    //Lê a reposta assinada da memória
16:     reply_j.id ← id_Rj;                           //Adiciona seu identificador
17:     send reply_kj to C_i ;                        //Envia a resposta para o cliente
18:     replycache ← replycache + reply_kj;           //Adiciona a resposta no cache
```

Algoritmo 4.2 – Protocolo de Execução das Réplicas

De posse da requisição ordenada, as réplicas verificam se a requisição lida é válida, verificando a assinatura desta utilizando o

certificado enviado pelo cliente (Algoritmo 4.2 – Linha 10). Caso a mensagem seja válida (assinatura correta verificada com o certificado), as réplicas executam a operação enviada pelo cliente e escrevem a resposta ($resp_k$) da execução desta operação na memória compartilhada nos *buffer*s individuais de resposta (*repBuffer*) (Algoritmo 4.2 – Linhas 11 e 12).

Após finalizar o processo de execução e escrita, as réplicas aguardam a notificação da presença da resposta assinada na memória compartilhada no *buffer buffer_toSend*. De posse da resposta assinada ($<REPLY, id_{Rj}, id_{Ci,} t, resp_{k,} Sign_{AS,} Cert_{AS}>$) (Algoritmo 4.2 – Linha 15), as réplicas inserem o seu identificador nesta mensagem (Algoritmo 4.2 – Linha 16) e fazem o envio para o cliente (Algoritmo 4.2 – Linha 17).

Além disso, uma cópia desta resposta é armazenada em seu *cache* local para atender a possíveis retransmissões (Algoritmo 4.2 – Linha 18).

Detalhes referentes à verificação da validade da requisição devem ainda ser esclarecidos.

O protocolo de ordenação executado pelo *Agreement Service* simplesmente lê as requisições da memória compartilhada, previamente escritas pelas réplicas de execução, e as insere em ordem no *buffer* de leitura de mensagens ordenadas, não fazendo qualquer verificação da validade ou integridade destas requisições. Desta forma, réplicas maliciosas poderiam tentar forjar mensagens na memória compartilhada, inserindo possíveis requisições de clientes, como se esta réplica realmente as tivesse recebido de um cliente correto.

Ainda que réplicas maliciosas tentem forjar requisições, estas serão descartadas após as réplicas corretas efetuarem sua leitura da memória compartilhada. Isso se deve ao fato de que as réplicas irão verificar que as assinaturas não têm sua validade comprovada, pois as réplicas (maliciosas ou corretas) não possuem os certificados de chave privada dos clientes.

Porém, se a verificação da validade da requisição fosse efetuada antes da ordenação feita pelo *Agreement Service*, réplicas corretas poderiam ler mensagens forjadas pelas réplicas maliciosas e executar suas operações normalmente. A execução de requisições incorretas causaria um problema no estado do sistema e no restante da execução do protocolo.

Outro detalhe importante a ser ressaltado, é o fato de que as réplicas de execução não assinam as mensagens que são enviadas para o cliente. Estas somente inserem sua identificação para que cliente tenha conhecimento de qual servidor recebeu a resposta.

A não necessidade de assinatura por partes das réplicas nas mensagens de resposta se deve ao fato de que os clientes esperam as mensagens válidas assinadas pelo *Agreement Service*. Não é necessário que as réplicas assinem tais mensagens novamente. Ainda que réplicas maliciosas tentem forjar mensagens de respostas e as enviar para o cliente, tais mensagens serão descartadas, pois as réplicas maliciosas não possuem a chave privada do *Agreement Service* para assinar estas mensagens, o que é indispensável para provar sua validade ao ser recebida pelo cliente.

4.6.4.2 Operação de *Checkpoint* das Réplicas

Um subprotocolo de *checkpoint* é executado periodicamente nas réplicas de execução. Este protocolo tem o objetivo de manter uma cópia (*snapshot*) do estado do sistema. Além disso, quando o *checkpoint* é efetuado, é possível verificar se todas as réplicas de execução estão mantendo o estado do sistema consistente (todas as réplicas de execução corretas devem manter o mesmo estado do sistema).

A operação de c*heckpoint* também é efetuada para que as réplicas possam fazer a coleta de lixo de mensagens antigas e para manter um estado consistente que possa ser utilizado por novas réplicas ao serem adicionadas ao protocolo.

Quando o subprotocolo de *checkpoint* é iniciado, o subprotocolo de execução das réplicas (apresentado na seção 4.6.4.1) é interrompido (Algoritmo 4.3 – Linha 2), para que não ocorra qualquer alteração no estado até que o *checkpoint* seja finalizado.

As réplicas de execução então geram um *hash* do estado atual do sistema (Algoritmo 4.3 – Linha 3) e escrevem este valor de *hash* na memória compartilhada em seus *buffer*s específicos ($hashBuffer_{Rj}$) (Algoritmo 4.3 – Linha 4).

Algoritmo 4.3. Protocolo de Checkpoint das Réplicas R_I

```
1:  procedure Checkpoint(state)
2:      executionThread.stop;
3:      hash_ki ← calculateHash(state_ki);          //Calcula o hash do estado
4:      shared_mem.hashBuffer_Ri.write(hash_ki);    //Escreve o hash na memória
5:      δ_p ← t() ;                                 //Inicia um novo período de checkpoint
6:      executionThread.resume;
7:  upon (t() - δ_p) ≥ period_Ck at R_I do
8:      Checkpoint(state_ki);                       //Executa o procedimento de checkpoint
                                                    // passando como parâmetro o estado k
```

Algoritmo 4.3 – Protocolo de Checkpoint das Réplicas

Ao final da escrita do valor de *hash* na memória compartilhada, um novo período para a realização do próximo *checkpoint* é iniciado (Algoritmo 4.3 – Linha 2), a operação de *checkpoint* é finalizada e o subprotocolo de execução é retomado em sua execução normal (Algoritmo 4.3 – Linha 6).

Réplicas maliciosas podem escrever *hash* aleatórios ou mesmo tentar corromper o estado do sistema. Para solução deste problema, a operação de *checkpoint* é coordenada pelo *Agreement Service*, de forma similar a apresentada na execução das réplicas. A coordenação do *checkpoint* será descrita na seção 4.6.5.3.

4.6.4.3 Recuperação das Réplicas

Novas réplicas são adicionadas ao grupo quando do desacordo nas respostas à execução das requisições ou do desacordo nos *checkpoints*. Estas novas réplicas são adicionadas ao grupo com o objetivo de resolver os desacordos causados por réplicas maliciosas e também fazer a substituição destas. Para isso, são mantidas *f* réplicas em máquinas virtuais em estado pausado. Estas *f* réplicas, apesar de estarem pausadas, não utilizam recursos de processamento e não são acessadas por qualquer entidade.

As novas réplicas ao serem iniciadas pelo *Agreement Service*, não têm conhecimento do estado atual do sistema, sendo assim, não podem responder imediatamente as requisições dos clientes. Para que as novas réplicas possam responder corretamente às requisições dos clientes, primeiramente devem recuperar um estado válido de alguma réplica de execução ativa.

Dessa forma, o primeiro passo executado por uma nova réplica ao ser iniciada é ler da memória compartilhada no *buffer buffer_stableCheckpoint* (Algoritmo 4.4 – Linha 6) qual o valor *hash* do último *checkpoint* estável executado pelas réplicas. Este valor é escrito neste *buffer* pelo *Agreement Service* ao executar o protocolo de acordo sobre os valores *hash* de *checkpoint*, descrito na seção 4.6.5.3. Além disso, esta nova réplica solicita uma cópia do estado do sistema (*snapshot*) de alguma das réplicas de execução ativas. Este estado recuperado é referente ao último *checkpoint* estável executado.

Algoritmo 4.4. Protocolo de Recuperação das Réplicas R_j

```
Variables
1:  state = Ø;                                          //Estado do sistema
2:  temp_state = Ø;                                     //Estado temporário
3:  stateHash = Ø;                                      //Hash do estado estável
4:  execRange[][] = Ø;                                  //Intervalo de req. a serem executadas
Init:
5:  upon Start at NEW_REPLICA_Rj
6:      stateHash ← shared_mem.buffer_stableCheckpoint.read();   //Lê o último hash estável da memória
7:      execRange ← shared_mem.buffer_updateRange.read();        //Lê o intervalo a ser executado
8:      while (state == Ø)                              //Enquanto não obter estado válido
9:          try
10:             temp_state ← getState() from any Ri ;   //Obtém o estado de alguma réplica
11:             if GetHash(temp_state) == stateHash then //Se os valores de hash forem iguais
12:                 state ← temp_State;                 //Recupera o estado
13:             end if
14:         end try
15:     end while
16: for execRange.Initial to execRange.Final do         //Executa as requisições do intervalo
17:     req_toExec ← shared_mem.buffer_toExecute.read(); //Lê a requisição ordenada
18:     if VerifySign(req_toExec.Sign, req_toExec.Cert ) then  //Verifica a validade da mensagem
19:         resp_k ← thread.execute(req_toExec.op);     //Executa a operação
20:         shared_mem.repBuffer_Rj .write (resp_k);    //Escreve a resposta na memória
21:     end if
22: end for
```

Algoritmo 4.4 – Protocolo de Recuperação das Réplicas

De posse do estado do sistema, essa nova réplica verifica se o *snapshot* recebido é compatível com o valor *hash* lido da memória compartilhada. Em caso afirmativo, a nova réplica recupera este estado para seu sistema, caso contrário, solicita o estado de outra réplica. O processo é repetido até que um estado válido seja recuperado (Algoritmo 4.4 – Linhas 8 – 15). O fato de o protocolo conter pelo menos $f + 1$ réplicas de execução garante que pelo menos uma dessas réplicas conterá um estado válido.

A recuperação do estado da aplicação ainda não possibilita que essa nova réplica responda ao cliente. Isto se deve ao fato de que as

réplicas que estavam em execução até o momento do desacordo podem ter executado uma série de requisições do cliente que não constam no estado do último *checkpoint*, uma vez que este é executado periodicamente. Por exemplo, um processo de *checkpoint* ocorre no momento em que as réplicas terminaram de executar a requisição com número de sequência 100. Porém, o instante em que ocorreu o desacordo entre estas réplicas foi na requisição de número 150. A nova réplica, apesar de recuperar o estado do último *checkpoint*, não executou estas 50 requisições posteriores ao *checkpoint*.

Para resolver este problema, além do valor do *hash* do *checkpoint*, a nova réplica lê da memória compartilhada o intervalo de requisições que foram executadas desde o último *checkpoint* do *buffer buffer_updateRange* (Algoritmo 4.4– Linha 7). A nova réplica então executa todas as requisições do intervalo lido até atingir a requisição onde houve o desacordo de respostas (Algoritmo 4.4 – Linhas 16 - 22).

Com o estado da aplicação atualizado, a resposta dessa nova réplica irá resolver o desacordo causado pelas réplicas maliciosas e o protocolo seguirá sua execução normal.

4.6.5 Comportamento do *Agreement Service* no Protocolo

O *Agreement Service* (AS) é o componente central na abordagem do protocolo ITVM. Ele agrega grande parte das funções relativas aos protocolos de ordenação e acordo sobre as respostas das réplicas de execução. Além disso, este componente também tem controle sobre o comportamento dinâmico do protocolo. Na identificação de desacordos, adiciona novas réplicas e remove réplicas comprometidas.

Este componente é composto pelos subprotocolos que são descritos nas próximas seções.

4.6.5.1 Ordenação das Requisições pelo AS

O objetivo deste subprotocolo é definir uma sequência para execução das requisições, para que todas as réplicas as executem na mesma ordem, seguindo os objetivos de uma replicação Máquina de Estados.

Quando notificado da presença de novas requisições na memória, o subprotocolo faz a leitura dos *buffer*s *reqBuffer* de cada uma das

réplicas de execução (Algoritmo 4.5 – Linha 3). Este subprotocolo verifica se tal requisição ainda não foi ordenada (Algoritmo 4.5 – Linha 4). Caso a requisição já tenha sido ordenada, a mesma é descartada (Algoritmo 4.5 – Linha 9), caso contrário, a mensagem é inserida na memória compartilhada no *buffer buffer_toExecute* (Algoritmo 4.5 – Linha 5), definindo uma sequência para a execução da requisição pelas réplicas.

Algoritmo 4.5. Protocolo de Ordenação de Requisições no **AS**

```
Variables
1:    orderedList = Ø;                                  //Lista das requisições ordenadas
Init:
2:  upon ReqSignal  at AS do
3:    while ∀j, ∃(req_k) ∈ shared_mem.reqBuffer_Rj      //Lê todos os buffers de requisição das réplicas
4:        if req_k ∉ orderedList then                   //Se a requisição não foi ordenada
5:          shared_mem.buffer_toExecute .write(req_k);  //Insere a requisição ordenada na memória
6:          orderedList ← orderedList ∪ req_k;          //Adiciona a requisição à lista de ordenadas
7:          δ_k ← t();                                  //Inicia o temporizador para recebimento das respostas
8:        else
9:          Discard(req_k);                             //Descarta a requisição já ordenada
10:       end if
11:   end while
                                                        //Se o temporizador de resposta expirar
12: upon  (t()  - δ_k ) ≥ timeout_k  at AS do
13:     initiatedVMs(f, bufferHash);                    //Inicia f novas réplicas
14:     δ_k ← t() ;                                     //Reinicia o temporizador
```

Algoritmo 4.5 - Protocolo de Ordenação das Requisições do *Agreement Service*

Como o *buffer buffer_toExecute* somente pode ser escrito pelo *Agreement Service*, a ordem de sequência das requisições e feita linearmente, ou seja, a ordem de sequência é definida pela posição em que a requisição é inserida neste *buffer*. Desta maneira, as leituras das requisições armazenadas neste *buffer*, sendo respeitada uma ordem *FIFO* (*First-In-First-Out)*, já garante que todas as réplicas irão executar as requisições na mesma ordem.

Com a requisição disponibilizada no *buffer buffer_toExecute*, um temporizador é iniciado, aguardando as resposta para esta requisição (Algoritmo 4.5 – Linha 7). No caso do recebimento de todas as respostas pelo *Agreement Service* para dada requisição, o temporizador é desativado. Porém, o não recebimento de todas as respostas até o temporizador expirar acarreta na ativação de novas réplicas (Algoritmo 4.5 – Linhas 12 e 13). Isso se deve ao fato de que réplicas maliciosas podem estar se negando a executar a requisição. Dessa forma, este temporizador é necessário para que o protocolo não fique indefinidamente aguardando por respostas para uma requisição. Os

detalhes sobre ativação de novas réplicas serão discutidos na seção 4.6.5.4

As réplicas de execução, ao receberem a requisição de um cliente, a inserem na memória compartilhada. O protocolo de ordenação então lê estas mensagens da memória compartilhada para a ordenação. Ainda que réplicas diferentes insiram a mesma mensagem na memória compartilhada, o protocolo irá ordenar somente uma delas. Para isso, uma lista de mensagens já ordenadas é mantida. As requisições já ordenadas são descartadas. (Algoritmo 4.5 – Linha 9).

Como definimos anteriormente, o *Agreement Service* encontra-se isolado dos demais domínios e da rede externa, de forma que o mesmo não possa ser comprometido e apresentar comportamento arbitrário. Confiamos que tal componente irá seguir as especificações do algoritmo e não apresentar comportamento malicioso, como por exemplo, ordenar uma mesma requisição várias vezes.

4.6.5.2 Verificação das respostas das réplicas de execução pelo AS

Como discutido na seção 4.6.4.1, as réplicas de execução inserem a resposta na memória compartilhada ao executarem uma operação enviada pelo cliente. Ao ser sinalizado, o subprotocolo faz a leitura do *buffer repBuffer* de cada uma das $f + 1$ réplicas obtendo a resposta da execução da operação (Algoritmo 4.6 – Linha 4). De posse desses valores de resposta, duas condições podem ser atingidas. A primeira delas é que, todas as respostas são iguais (caso otimista), ou estas respostas diferem entre si.

Caso as $f + 1$ respostas lidas da memória compartilhada sejam iguais (Algoritmo 4.6 – Linha 7), uma resposta assinada é criada pelo subprotocolo para ser enviada para o cliente. A resposta com o formato $<REPLY, id_{Rj}, id_{Ci,} t, resp_{k,} Sign_{AS,} Cert_{AS}>$ descrito na seção 4.6.3, é então criada e assinada pelo *Agreement Service* (Algoritmo 4.6 - Linha 8 e Linha 9). Esta resposta é então disponibilizada na memória compartilhada no *buffer buffer_toSend* (Algoritmo 4.6 – Linha 10) para que as réplicas de execução possam ler esta resposta e a enviar para o cliente. Após a escrita da resposta na memória compartilhada, o temporizador definido no algoritmo de ordenação, responsável pelo aguardo do recebimento das respostas, é removido (Algoritmo 4.6 – Linha 11). É imprescindível que a assinatura ($Sign_{AS}$) e o certificado

($Cert_{AS}$) sejam efetuados, pois são estes campos que garantem a validade da resposta quando verificada pelo cliente.

Algoritmo 4.6. Protocolo de Acordo de Respostas no **AS**

```
Variables
1:    repList = Ø;                                          //Lista das respostas das réplicas
2:    needRestore = false;                                  //Flag de necessidade de restauração
Init:
3:  upon RepSignal  at AS do
4:    while ∀j, ∃(rep_k) ∈ shared_mem.repBuffer_Rj  do      //Lê todas as respostas das réplicas
5:        repList ← repList ∪ {shared_mem. repBuffer_Rj .read() };   //Adiciona as respostas a lista
6:          if  |repList| ≥ f +1 then                       //Se o tamanho da lista for ≥ f +1
7:            if  match (repList ) =  f +1 then             //Se houverem f +1 respostas iguais
8:              Sign_AS ← Signature( <REPLY, rep_k.id_Ci , t, rep_k.resp>);   //Gera a assinatura da resposta
9:              reply_k ← <REPLY, null, id_Ci, resp_k , Sign_AS , Cert_AS>;   //Cria a resposta para o cliente
10:             shared_mem.buffer_toSend.write(reply_k);      //Escreve a resposta na memória
11:               δ_k ← Ø;                                    //Remove o temporizador de respostas
12:               if needRestore then                         //Se necessitar restauração
13:                 restoreVMs(f + 1) ;                       //Restaura a replicação para f + 1
14:                 needRestore ← false;
15:               end if;
16:           else                                          //Se não ouver f + 1 respostas iguais
17:             initiatedVMs( f );                          //Inicia f novas réplicas
18:             needRestore ← true;                         //Após a inicialização de f novas
19:           end if;                                       // réplicas é necessário a restauração
20:         end if;
21:   end while;
```

Algoritmo 4.6 – Protocolo de Acordo de Respostas no *Agreement Service*

Quando não são obtidas $f + 1$ respostas iguais para uma mesma requisição (Algoritmo 4.6 – Linha 16) é detectado que existem réplicas apresentando comportamento malicioso e escrevendo valores de respostas que divergem do esperado pela execução da aplicação. Neste caso, f novas réplicas que são mantidas pausadas são então iniciadas para a resolução do desacordo entre as respostas (Algoritmo 4.6 – Linha 17).

Neste modelo com somente $f + 1$ réplicas, caso o limite de réplicas maliciosas f seja superior ou igual a 2, a identificação das réplicas maliciosas é mais complexa. Por exemplo, se trabalharmos com o valor de $f = 2$, teremos um total de três réplicas totais na replicação e diferentes situações poderão ocorrer com relação às respostas destas réplicas. São ilustradas na Tabela 4.1 estas situações:

Assumimos no exemplo que o valor esperado de resposta é representado pelo valor (1) e o valor incorreto representado pelo valor (2). No Caso 1 é apresentada a situação onde todas as réplicas apresentam a resposta correta (1), não havendo desacordo. No Caso 2, a Réplica 3 tem comportamento malicioso e apresenta uma resposta

incorreta (2), enquanto as demais apresentam a resposta correta (1). Por fim, no Caso 3 é apresentado o caso onde somente a Réplica 1 é correta, apresentando o valor de resposta correto (1) enquanto as demais são maliciosas e apresentam o valor de resposta incorreto (2).

Tabela 4.1 - Tabela de Respostas com *f + 1* réplicas

	Caso 1	**Caso 2**	**Caso 3**
Réplica 1	1	1	1
Réplica 2	1	1	2
Réplica 3	1	2	2

As situações problemáticas são os casos 2 e 3, pois ambos apresentam duas respostas iguais e uma resposta diferente, no conjunto de três respostas. No Caso 2 seria possível identificar a Réplica 3 como maliciosa, pois é a única que apresenta valor diferente das réplicas corretas (se soubéssemos que o valor correto é (1)). Porém, no Caso 3, a Réplica 1 apresenta um valor diferente, mas, é a única correta. A distinção entre réplicas corretas e incorretas é impossível de ser feita com somente *f + 1* respostas.

A introdução de *f* novas réplicas aumenta o número de respostas para *2f + 1*, transpondo a impossibilidade de decisão sobre as respostas ocorrida com somente *f + 1* réplicas, e desta forma podendo ser identificada a resposta correta do conjunto (a resposta que apresentar maioria é considerada a resposta correta). A obtenção da resposta correta permite a continuidade da execução do protocolo da mesma forma que o caso otimista.

4.6.5.3 Coordenação do *Checkpoint* no AS

Para que o estado do sistema das réplicas seja mantido de forma consistente, é executado no *Agreement Service* um subprotocolo de coordenação destes *checkpoints*.

A coordenação do *checkpoint* funciona de forma análoga ao protocolo de acordo das respostas definido na seção 4.6.5.2. Periodicamente, as réplicas de execução efetuam o *checkpoint* de seu estado (seção 4.6.4.2) e escrevem o valor de *hash* deste estado na memória compartilhada em seus buffers *hashBuffer*. O subprotocolo de coordenação de *checkpoint* então é sinalizado para a leitura destes

buffers (Algoritmo 4.7 – Linha 5). Após a leitura destes valores de hash, o protocolo verifica se estes valores escritos pelas réplicas tem a frequência de *f + 1* valores iguais (Algoritmo 4.7 – Linha 7).

Algoritmo 4.7. Protocolo de Acordo de *Checkpoint* no **AS**

```
Variables
1:   hashList = Ø;                                          //Lista dos hashes das réplicas
2:   needRestore = false;                                   //Flag de necessidade de restauração
Init:
3:  upon CkPointSignal at AS do
4:    while ∀j, ∃(hash_k) ∈ shared_mem.hashBuffer_Rj do     //Lê todas os hashes das réplicas
5:      hashList ← hashList ∪ {shared_mem. hashBuffer_Rj .read() };  //Adiciona os hashes a lista
6:        if |hashList| ≥ f +1 then                         //Se o tamanho da lista for ≥ f +1
7:          if match (hashList ) = f +1 then                //Se houverem f +1 hashes iguais
8:            shared_mem.buffer_stableCheckpoint.write(hash_k);  //Escreve a hash do C.P. estável
9:              if needRestore then                       //Se necessitar restauração
10:               restoreVMs(f + 1) ;                     //Restaura a replicação para f + 1
11:               needRestore ← false;
12:             end if;
13:         else                                          //Se não ouver f + 1 hashes iguais
14:           initiateVMs( f );                           //Inicia f novas réplicas
15:           needRestore ← true;                         //Após a inicialização de f novas
16:         end if;                                       // réplicas é necessário a restauração
17:       end if;
18:   end while;
```

Algoritmo 4.7 – Protocolo de Acordo de *Checkpoint* no *Agreement Service*

Em caso afirmativo, o *Agreement Service* escreve na memória compartilhada no *buffer buffer_stableCheckpoin*t este valor *hash* (Algoritmo 4.7- Linha 8) que é lido pelas novas réplicas adicionadas em caso de desacordo, como descrito na seção 4.6.3.

Caso não tenham sido escritos *f + 1* valores *hash* iguais, (Algoritmo 4.7 – Linhas 13 – 15) é identificado que existem réplicas maliciosas e *f* novas réplicas são iniciadas para resolver este desacordo, assim com ocorre no protocolo de acordo de respostas.

4.6.5.4 Ativação de Novas Réplicas no AS

A ativação de novas réplicas é necessária para que o quórum de réplicas em execução seja suficiente para chegar a um acordo sobre os valores escritos (*respostas* e *checkpoint*). A ativação de *f* novas réplicas é uma operação simples em nossa abordagem. Uma vez que o *Agreement Service* é executado no mesmo nível de virtualização do

VMM, este componente tem controle sobre as réplicas, tanto para inicialização quanto para o seu desligamento.

Como as réplicas são executadas em máquinas virtuais, estas *f* réplicas sobressalentes são mantidas pausadas, para que sua inicialização ocorra de forma mais rápida e o tempo de inatividade (*downtime)* do protocolo ITVM seja reduzido. Porém, nada impede que estas máquinas virtuais sejam mantidas desligadas ou mantidas em estados similares.

Algoritmo 4.8. Protocolo de Ativação de Novas Réplicas no **AS**

```
Variables
1:   newReplicasList = {f} ;                    //Conjunto com f réplicas pausadas
2:   updateRange[][] = Ø;                       //Requisições que foram precisam ser executadas
Init:                                           // para atualizar o estado da réplica
3:  procedure initiateVMs( f );
4:   updatRange.first ← |LastCheckpoint_Update#| +1 ;   //Primeira requisição depois do ultimo C.P
5:   updatRange.Last ← Disagreement_Update# ;          //Requisição em que houve o desacordo
6:   shared_mem.buffer_updateRange.write(updateRange);  //Escreve o intervalo na memória

7:   for each replica Ri ∈ newReplicasList
8:       StartVM(Ri);                           //Inicia a réplica
9:   end for;
```

Algoritmo 4.8 – Protocolo de Ativação de Novas Réplicas no *Agreement Service*

No processo de inicialização de novas réplicas, o *Agreement Service* envia comandos ao VMM, para que este inicie as máquinas virtuais referentes a estas réplicas (Algoritmo 4.8 – Linha 3). Além disso, o *Agreement Service* disponibiliza informações indispensáveis para que estas novas réplicas, ao serem iniciadas, possam recuperar o estado do sistema necessário para poder responder corretamente ao cliente. Especificamente, estes valores são o *hash* do último *checkpoint* válido, descrito na seção 4.6.3.3, e também qual o intervalo de requisições que foram executadas pelas réplicas até ocorrer o desacordo que causou a inicialização desta.

Este intervalo é definido da seguinte maneira: O valor inicial é o número da requisição em que houve o último *checkpoint*, acrescido de 1 (Algoritmo 4.8 – Linha 4), e o valor final é exatamente o número da requisição em que houve o desacordo (Algoritmo 4.8 – Linha 5).

4.6.5.5 Recuperação da Replicação para *f + 1* réplicas no AS

Para garantir que nossa proposta continue sendo válida para *f + 1* réplicas, ao final da solução dos desacordos, as réplicas que tiveram seu comportamento malicioso detectado são desligadas.

Além disso, podem ocorrer situações onde nem todas as réplicas maliciosas são detectadas. Por exemplo, quando o número de réplicas maliciosas tolerada é maior ou igual a 2 ($f \geq 2$), se uma das réplicas que estiver em execução apresentar comportamento malicioso, duas novas réplicas serão iniciadas. Ao final da execução, essa única réplica que apresentou comportamento malicioso será identificada e desligada, porém, como foram iniciadas duas novas réplicas novas, uma correta adicional continuará em execução.

Neste caso, onde existem réplicas corretas sobressalentes, qualquer uma das réplicas corretas será desligada, retornando para *f + 1* o número de réplicas ativas na replicação.

4.6.6 COLETA DE LIXO

Visto que a memória compartilhada possui nos seus diversos *buffers* capacidade de armazenamento finito, é necessário que se faça a remoção das informações antigas. Um *deadline* foi especificado no AS para impedir o problema de *buffer* infinito, dessa forma, periodicamente informações antigas são removidas. Já nas réplicas ao ser efetuado um *checkpoint*, as requisições anteriores a este podem ser removidas.

4.7 CORREÇÃO DO SUPORTE DO ALGORITMO

No desenvolvimento de sistemas distribuídos, mostrar a correção de algoritmos é sempre importante e nem sempre uma tarefa fácil. A correção de algoritmos em ambientes distribuídos é sempre baseada na garantia das propriedades conhecidas como S*afety (Segurança)* e *Liveness (Vivacidade)*. Dizer que a propriedade de *Safety* é garantida, significa dizer que “coisas ruins” não irão acontecer no período de execução do protocolo, e dizer que a propriedade de *Liveness* é garantida, significa afirmar que “coisas boas” irão, em algum momento, acontecer na execução do algoritmo (LAMPORT, L., 1977).

Em replicação Máquina de Estados, abordagem na qual este trabalho está inserido, estas propriedades podem ser descritas da seguinte forma:

- ***Safety***: Todas as requisições enviadas pelos clientes são sempre executadas pelas réplicas corretas, que devem evoluir de maneira idêntica executando todas as requisições na mesma ordem, mesmo na presença de réplicas maliciosas.
- ***Liveness***: Todas as requisições enviadas pelos clientes em algum momento serão executadas, de forma que o sistema sempre faça progresso, com as réplicas evoluindo sempre segundo o determinismo de réplicas.

Ou seja, todas as requisições enviadas pelos clientes corretos ou não devem ser recebidas por todas as réplicas corretas (Acordo) e tais requisições devem ser executadas na mesma ordem de sequência (Ordem Total) (SCHNEIDER, 1990).

Para verificar estas propriedades de correção indicadas acima, enumeramos um conjunto de condições que devem ser asseguradas pelos nossos algoritmos de modo a garantir as propriedades de correção.

4.7.1 Condições para a Correção da Abordagem Proposta

No trabalho, assumimos a hipótese de que nosso serviço de acordo é um componente confiável, não desviando de suas especificações. Assumidos ainda o uso de canais confiáveis e parcialmente síncronos. Diante disso, descrevemos abaixo as condições para a correção de nossa abordagem:

A. Clientes corretos enviam requisições e retransmitem até que uma resposta válida seja recebida.

- O uso de canais confiáveis garante que as mensagens enviadas por clientes terminam por chegar as réplicas. Assumimos que, ainda que os canais funcionem de maneira assíncrona na maior parte do tempo, existem momentos em que estes canais se comportam de maneira síncrona (atingem o *Global Stabilization Time*), onde os envios das requisições são concluídos.

B. Pelo menos uma réplica correta recebe a requisição do cliente e insere a mesma na memória compartilhada.

- Através da condição A garantimos que mensagens enviadas por clientes corretos chegam, em algum momento, às réplicas. Como contamos inicialmente com *f + 1* réplicas, garantimos que pelo menos uma destas réplicas é correta e não omite o recebimento da requisição. De posse da requisição recebida do cliente e dado que os *buffer*s de escrita (*reqBuffer*) de requisições são individuais, todas as réplicas corretas escrevem a requisição recebida na memória compartilhada. Ainda que as escritas na memória compartilhada sejam assíncronas na maior parte do tempo, a propriedade AWB1 definida em (FERNANDEZ *et al.*, 2007) garante que os processos em algum momento assumirão o comportamento síncrono, finalizando a escrita da requisição na memória compartilhada em tempo finito (premissa do modelo).

C. O *Agreement Service* sempre irá ordenar as requisições.

- Assumimos no modelo que, devido ao isolamento e à simplicidade das funcionalidades do *Agreement Service*, o mesmo é confiável, ou seja, não irá desviar das especificações de seus algoritmos. Através da condição B, garantimos que o *Agreement Service* em algum momento irá ler a requisição da memória compartilhada (*bufferReq)*. Depois disto seu comportamento correto garante que o mesmo deve ordenar e disponibilizar a mesma via a escrita em *buffer_toExecute.* Como este *buffer* tem como único escritor o *Agreement Service*, e este é correto, as réplicas de execução, as quais são leitoras deste *buffer*, sempre terão uma visão única das requisições ordenadas e nunca lerão requisições diferentes com um mesmo número de sequência.

D. Réplicas corretas irão executar a requisição na mesma ordem (Ordem Total).

- A condição C garante que as requisições recebidas dos clientes pelas réplicas corretas serão inseridas na memória compartilhada já ordenadas pelo *Agreement Service*. Dada uma mensagem ordenada e presente na memória compartilhada para leitura, em algum momento réplicas corretas devem acessar esta requisição e executar a operação nela presente. Como o *buffer* de leitura de requisições ordenadas é único e o processo escritor deste *buffer*, o

Agreement Service, já assumido como correto em nossa hipótese, todas as réplicas corretas irão ler as requisições na mesma ordem.

E. O *Agreement Service* deve receber pelo menos *f + 1* respostas iguais para uma dada requisição de cliente.

- Dado que as réplicas corretas irão executar as requisições na mesma ordem (condição D) e estes escrevem estas respostas em seus *buffer*s particulares de resposta (*bufferRep*), em algum momento o *Agreement Service* irá receber *f + 1* respostas iguais (quórum de aceitação). Como a implementação desta técnica inicialmente envolve um procedimento otimista, com o uso de somente *f + 1* réplicas e, se contarmos com a presença de réplicas maliciosas entre estas *f + 1* réplicas, então a condição do quórum de aceitação (*f + 1* respostas iguais) fica ameaçada de não se concretizar. Porém, na presença de réplicas maliciosas, e no não recebimento de *f + 1* respostas iguais, *f* novas réplicas devem ser iniciadas, aumentando o número de réplicas para *2f + 1*. A existência de *2f + 1* réplicas garante que existirão pelo menos *f + 1* réplicas corretas produzindo *f + 1* respostas corretas (o quórum de aceitação necessário para a requisição).

F. O *Agreement Service* sempre disponibilizará a resposta assinada na memória.

- A condição E garante que o *Agreement Service* sempre irá receber *f + 1* respostas iguais e corretas, mesmo na presença de réplicas maliciosas. De posse dessas *f + 1* respostas iguais, o *Agreement Service* desativa as *f* réplicas discordantes fazendo o sistema voltar à replicação original (*f + 1* réplicas). Na sequência, cria e assina a resposta a ser enviada para o cliente. Essa resposta é então escrita na memória compartilhada no *buffer buffer_toSend*, cujo único escritor é o *Agreement Service*, deixando esta resposta assinada disponível para que as *f + 1* réplicas restantes leiam e enviem a mesma para o cliente.

G. Alguma réplica correta enviará a resposta válida para o cliente.

- A condição F garante que a resposta a ser enviada para o cliente será disponibilizada na memória compartilhada pelo *Agreement Service*. Ainda que *f* entre as *f + 1* réplicas em execução omitam o envio da resposta para o cliente, pelo menos uma réplica correta deve fazer a leitura desta da memória compartilhada e, na

sequência, a envia para o cliente. Caso o cliente não receba esta resposta e seu temporizador expire, o mesmo retransmitirá a sua requisição. As réplicas corretas, de posse da resposta assinada, irão responder ao cliente, o que garante que em algum momento o cliente irá receber pelo menos uma resposta válida assinada pelo *Agreement Service*, garantindo que a condição A seja válida.

As condições acima devem, portanto, ser verificadas no nosso modelo, para que o mesmo forneça garantias de acordo e ordem sobre as mesmas, fazendo com que a propriedade *Safety* seja alcançada. Além disso, estas condições garantem que o algoritmo termine, como consequência, a propriedade *Liveness* é alcançada.

4.8 CONSIDERAÇÕES FINAIS

Neste capítulo introduzimos o ITVM, uma proposta de modelo de replicação Máquinas de Estados que faz uso de virtualização em sua construção. Apresentamos o modelo do sistema e os algoritmos que o compõe. Nosso modelo faz reduções no número de réplicas de execução de $3f + 1$ da literatura original (CASTRO; LISKOV, 2002a) para $f + 1$ réplicas de execução no caso otimista, onde não existem réplicas apresentando comportamento malicioso, utilizando um único servidor físico.

Esta redução para $f + 1$ réplicas de execução foi possível em nossa abordagem através do uso de um componente confiável. Este componente confiável implementa as funções de serviço de acordo do protocolo. Além disso, faz a detecção de réplicas maliciosas e as substitui por réplicas corretas.

Para garantir que este componente seja considerado como confiável, o mesmo fica inacessível a entidades externas que poderiam tentar corromper seu funcionamento. Apesar de seu isolamento, este se comunica com as réplicas de execução através de uma memória compartilhada.

Por fim, apresentamos as hipóteses assumidas e as condições para que o modelo garanta que as propriedades de *Safety* e *Liveness* sejam atingidas.

5 RESULTADOS EXPERIMENTAIS E ANÁLISES

Para testar nossas proposições, implementamos um serviço replicado segundo a abordagem proposta. Neste capítulo apresentaremos detalhes de nossa implementação e os resultados obtidos diante de diferentes situações de testes ao qual o protocolo ITVM foi exposto. Os testes foram elaborados para evidenciar possíveis virtudes e também problemas com a nossa proposta. É objetivo também deste capítulo, tanto nos testes como em avaliações qualitativas, a comparação com outras propostas que foram apresentadas no capítulo de Trabalhos Relacionados. Primeiramente, descreveremos os detalhes do ambiente onde foram realizados os testes, seção 5.1. Uma análise dos testes realizados e os resultados são apresentados na seção 5.4.

5.1 DETALHES DE IMPLEMENTAÇÃO DO PROTÓTIPO

São ilustrados na Figura 5.1 as camadas que definem nosso protótipo. O software de virtualização (VMM) empregado foi o *hypervisor* XEN (XEN, 2011), sendo executado sobre o sistema operacional Linux *OpenSuse*, o qual faz parte do suporte da virtualização. A escolha do XEN (XEN, 2011) como ferramenta de virtualização nos dá a possibilidade da utilização de uma área de memória compartilhada oferecida pela própria ferramenta, denominada *XenStore*. Este recurso oferece registradores de memória que são compartilhados entre as máquinas virtuais, de forma que as informações trocadas pelas réplicas de execução e o *Agreement Service* seja feito exclusivamente através dessa memória compartilhada. Além disso, o próprio VMM fica responsável pelo controle de acesso a essa memória compartilhada. Cada um dos registradores é definido com politicas de acessos restritivas, onde cada réplica só possa ler e/ou escrever em registradores específicos, o que é requisito para manter nossa memória compartilhada consistente e segura.

O VMM utilizado contém um domínio especial, denominado *Domain0*. Este é um domínio único, oferecido pelo *hypervisor* é utilizado para o gerenciamento das máquinas virtuais e da memória compartilhada. Nele se encontram as funções necessárias para iniciar/desligar as máquinas virtuais e também o controle de acessos à memória compartilhada. Diante das características apresentadas, inserimos este nosso componente confiável no *Domain0*, pois é através

do mesmo que as réplicas da nossa abordagem (réplicas de execução e réplicas em espera) são controladas na evolução do nosso modelo de ME tolerante a intrusões. Pela própria construção do protótipo e da ferramenta usada de virtualização, este domínio é mantido isolado das demais réplicas. Além disso, para manter este domínio totalmente inacessível via rede, desabilitamos todos os pontos de acesso (portas) via rede, presentes no sistema para esse nível.

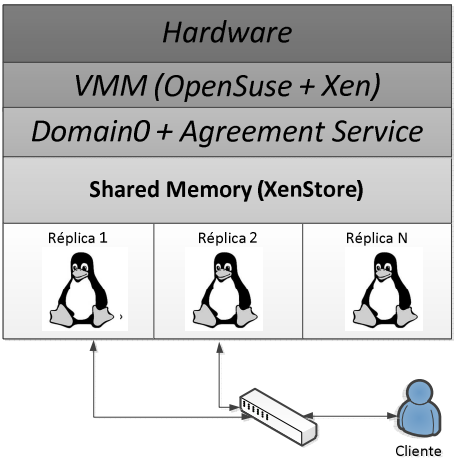

Figura 5.1 – Camadas do protótipo

Os sistemas operacionais executados nas máquinas virtuais foram diferentes distribuições do sistema operacional Linux e com diferentes versões de *kernel*. O objetivo é preservar a independência de falhas ou intrusões (as faltas ou vulnerabilidades devem se reproduzir com baixa probabilidade nas diferentes máquinas virtuais), por isso a escolha de diferentes distribuições e versões de *kernel*. Nos clientes, utilizamos diferentes arquiteturas de *hardware* e diferentes sistemas operacionais, tais como diferentes versões de Linux e Windows, valorizando o protótipo como ambiente heterogêneo.

As mensagens trocadas pelo cliente e o *Agreement Service* são assinadas utilizando técnica de criptográfia assimétrica. Adotamos o algoritmo RSA para a assinatura dessas mensagens. Nos testes realizados utilizamos uma chave criptográfica de tamanho 1024 bits.

5.2 SERVIÇO DE APLICAÇÃO USADO NA REPLICAÇÃO ME

Para evidenciar as propriedades da nossa abordagem, desenvolvemos uma aplicação a ser usada como uma ME sobre o protótipo desenvolvido. Esta aplicação implementada na forma de um serviço replicado corresponde a algo próximo de transações bancárias. O cliente faz o envio das requisições com operações de CRÉDITO ou DÉBITO para todos os servidores replicados. Os servidores após a ordenação da requisição executam a operação enviada pelo cliente. A execução da requisição faz com que o estado da aplicação (conta do cliente) seja modificado conforme o tipo de operação especificado na requisição. Os servidores, após executarem a operação, inserem a resposta (saldo da conta do cliente) a esta execução na memória compartilhada. O *Agreement Service* aguarda por saldos iguais vindos dos diferentes servidores para criar a resposta a ser enviada para o cliente.

Operações de *checkpoint* são efetuadas pelas réplicas com o objetivo de manter consistente o estado da aplicação. Então, nestas operações, as réplicas salvam um *snapshot* das contas dos clientes. Em nossos testes, as operações de *checkpoint* foram efetuadas a cada 50 requisições executadas pelas réplicas.

5.3 CONSIDERAÇÕES SOBRE AS IMPLEMENTAÇÕES

No desenvolvimento do protótipo foram feitas observações importantes. Todos os processos de nosso protótipo foram implementados usando a linguagem de programação Java 6. A utilização da linguagem Java nos permitiu a criação do protótipo de forma mais ágil, por conta do grande número de estruturas de dados prontas. Porém, a integração desta linguagem de programação para a manipulação da memória compartilhada oferecida pelo *hypervisor* XEN causou um alto *overhead*, visto que, com a utilização desta tecnologia em específico, o acesso aos recursos de memória compartilhada é efetuado somente por linguagens de programação específicas (Linguagem C e Python). Dessa forma, a grande parte do impacto nos tempos de resposta deve ser creditada às operações de escrita e leitura em memória compartilhada. O uso de outro modelo de memória compartilhada, ou a implementação deste protótipo nas linguagens de

programação acima citadas podem fazer com que o desempenho seja aumentado.

Outro fator a ser observado foi a utilização da virtualização em um servidor com hardware comum. A utilização de hardware especializado para virtualização também pode fazer com que o desempenho de nossos protótipos seja aumentado.

Nas subseções subsequentes apresentamos o ambiente e os testes realizados sobre o protótipo de nosso modelo de ME tolerante a intrusões.

5.4 TESTES E ANÁLISES SOBRE O PROTÓTIPO

Com intuito de verificar as potencialidades da nossa abordagem e a conformidade de nosso protótipo, desenvolvemos um conjunto de testes. Estes testes servem também para evidenciar diferenças sobre modelos: o nosso e alguns presentes na literatura. Na sequência começamos introduzindo o ambiente computacional onde foram desenvolvidos os testes.

5.4.1 Ambiente de Testes

Em nossos testes, clientes e servidores se comunicam através de uma rede local com velocidade de 100mbps. Os testes foram realizados em uma rede compartilhada com demais computadores do departamento, porém em horários onde a carga de rede era supostamente baixa, dessa forma, sem muitas interferências.

Para a execução das máquinas virtuais utilizamos um servidor físico com processador Intel Core I7, 8 GB de memória RAM. Cada uma das máquinas virtuais foi configurada de forma a compartilhar o processador da máquina servidora e utilizar 1 GB de memória RAM.

5.4.2 Consideração Gerais sobre os Testes Realizados

Visando verificar o desempenho do suporte de ME tolerante a intrusões e do serviço de aplicação implementado, testes então foram executados sobre o protótipo desenvolvido. Nos testes realizados, foi considerado o limite para réplicas maliciosas como $f = 1$. Este valor de f

foi escolhido pela falta de recursos da máquina física, ou seja, a máquina hospedeira não tem condições de manter um grande número de máquinas virtuais. Os testes realizados foram executados em dois cenários distintos: (i) ambiente sem réplicas maliciosas (caso otimista); e (ii) ambiente com réplicas maliciosas. Além disso, testamos nosso protótipo com a presença de clientes simultâneos.

Todos os testes foram executados em rodadas de 1000 requisições sendo enviadas e recebidas por cada cliente. No total, fizemos para cada um dos testes, 3 rodadas de 1000 requisições e retiramos a média aritmética das 3 rodadas.

Para calcular o tempo de resposta é feita a marcação do tempo local pelo cliente após o envio da requisição e novamente esta marcação é feita no recebimento de uma resposta válida vinda das réplicas de execução (*Round-Trip Time*).

5.4.3 Execuções Sem Replicação

Somente para ter como base de comparação, o tempo médio de resposta quando o serviço é executado em somente uma réplica é de aproximadamente 7 *ms*. Porém nestas condições o serviço não é replicado e não oferece qualquer garantia de segurança e tolerância a intrusão. Na Figura 5.2 é apresentado o comportamento do sistema sem replicação.

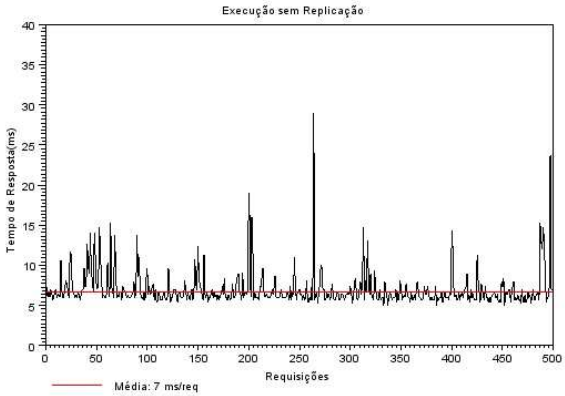

Figura 5.2 – Execução do protótipo sem replicação

5.4.4 Execuções Normais

Neste cenário, apresentamos a execução do protótipo em uma execução otimista, ou seja, sem a presença de falhas. O objetivo deste cenário é mostrar o tempo de resposta de nossa arquitetura e o comportamento do protocolo. Vale salientar que, sempre que utilizamos o termo falha em nosso texto, estamos nos referindo a intrusões ou qualquer atividade que faça com que a réplica de execução comece a apresentar comportamento arbitrário.

É ilustrado na Figura 5.3 o comportamento do sistema neste teste. No gráfico apresentado, utilizamos de 500 amostras das 1000 amostras de dados obtidos na execução dos testes. Podemos perceber que em toda a execução, as respostas são recebidas pelo cliente em tempos próximos à média (tempo médio de resposta que é de 25ms).

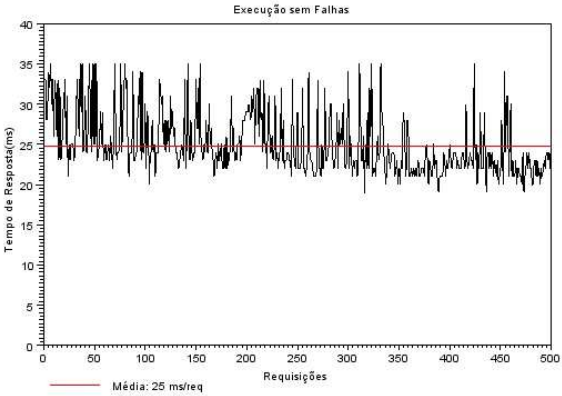

Figura 5.3 – Comportamento do Sistema sem Falhas

Um histograma da distribuição geral dos tempos de resposta das requisições é apresentado na Figura 5.4. No eixo “x”, são apresentados os blocos, que correspondem aos tempos de resposta das requisições em milissegundos (*ms*), enquanto no eixo “y” é apresentada a frequência com que estes tempos de respostas aparecem dentro de uma amostra de 1000 requisições. Podemos observar que há uma variação dos valores entre 18*ms* e 36*ms*, porém, como o esperado, a grande maioria das requisições fica concentrada próxima da média (tempo médio de resposta que é de 25ms).

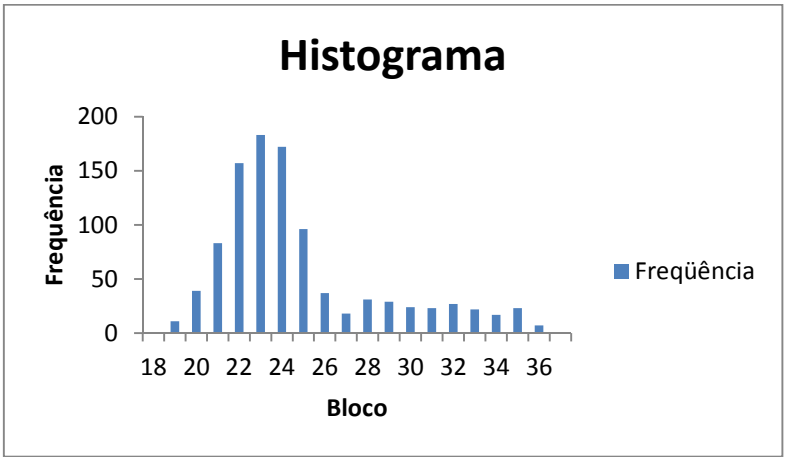

Figura 5.4 – Histograma dos testes sem falhas

5.4.5 Execuções com Falhas

Na seção anterior apresentamos o caso otimista, ou seja, sem falhas. O objetivo do cenário apresentado nesta seção é mostrar o comportamento do nosso protocolo na presença de falhas. Neste cenário, réplicas começam a se comportar de forma arbitrária, apresentando respostas erradas para a execução das operações enviadas pelos clientes. Então, para que o protocolo prossiga, o desacordo das respostas deve ser resolvido e a réplica maliciosa deve ser identificada e removida do sistema.

Assim como em grande parte dos trabalhos tolerantes a faltas bizantinas, o número máximo de réplicas que podem apresentar comportamento malicioso é limitado (*f* é o limite estipulado). Porém, nestas arquiteturas, uma réplica maliciosa pode permanecer no protocolo por toda a execução do sistema. Se considerarmos que a cada invasão de réplica a robustez do sistema é diminuída, então podemos chegar a situações em que este limite *f* é alcançado em seu ciclo de vida e qualquer outra invasão subsequente leva o número de réplicas maliciosas no sistema além do limite de *f*, fazendo com que o serviço replicado deixe de funcionar corretamente. Ou seja, o tratamento de réplicas maliciosas ou ainda o rejuvenescimento das réplicas no sistema devem ser feitos constantemente. Se for rejuvenescimento, os custos são maiores porque o serviço deve parar periodicamente.

Em nossos testes, como as réplicas maliciosas são removidas, o número de réplicas corretas retorna ao seu número mínimo necessário. Dessa forma, o protocolo sempre irá ser executado somente com réplicas corretas. Além disso, apesar da limitação de réplicas maliciosas existirem (*f* em composição de *f + 1*, em nossa abordagem), após a remoção da réplica maliciosa, o protocolo pode voltar a admitir a ocorrência de novas intrusões, porque estará com a sua replicação restaurada na sua máxima robustez. Ou seja, em nossa abordagem a replicação é sempre restaurada em cada caso de intrusão detectado.

Para verificar o comportamento de nossa abordagem, montamos casos de testes com diferentes distribuições de réplicas maliciosas. As distribuições são apresentadas na Tabela 5.1. A coluna $f_{\%}$ apresenta, dentro do conjunto de 1000 requisições, qual a porcentagem de falhas que ocorrem no conjunto. A coluna N_f apresenta o número de vezes que as falhas ocorrerão e a coluna Δr apresenta o intervalo de requisições corretas seguidas pela requisição com falha. Em nossas simulações, optamos pela ocorrência de falhas em requisições pré-determinadas para que os resultados fossem mais visíveis.

Tabela 5.1 - Distribuição das falhas

$f_{\%}$	N_f	Δr
0,1%	1	*
1,00%	10	99
2,50%	25	39
5,00%	50	19
10,00%	100	9
12,50%	125	7
16,60%	166	5
25,00%	250	3
50,00%	500	1
100,00%	1000	0

As subseções abaixo apresentam o comportamento do sistema diante destes cenários.

5.4.5.1 Experimento com Somente uma Falha (0,1%)

No cenário com apenas uma intrusão (Figura 5.5 e Figura 5.6) é possível perceber que, da ocorrência da intrusão até a recuperação, o tempo de resposta neste caso de execução sofre aumento para aproximadamente 1s (1000*ms*). Este impacto é causado principalmente pela inicialização e recuperação de estado pela réplica que está sendo adicionada ao protocolo. O impacto desta falha praticamente não interfere no tempo de resposta do conjunto de requisições do experimento como um todo.

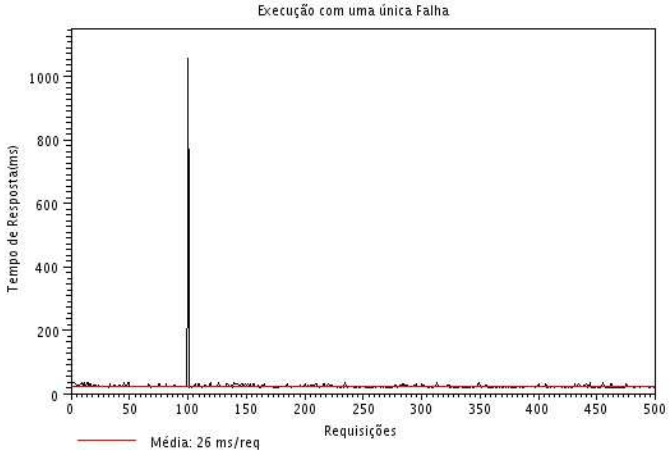

Figura 5.5 – Execução com somente uma falha

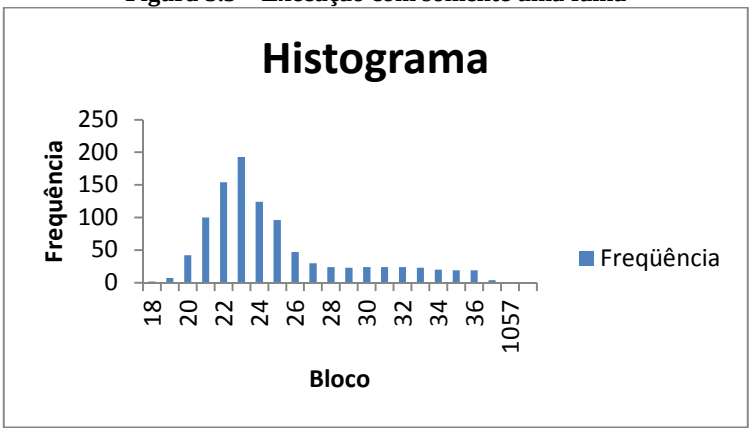

Figura 5.6 – Histograma dos testes com uma única falha

5.4.5.2 Execução com 1% de falhas

Neste cenário, apresentado nas Figura 5.7 e Figura 5.8, os picos já começam a acontecer com mais frequência e o tempo médio de resposta começa a ser afetado devido à presença mais constante das falhas. Como o tempo de recuperação das réplicas é relativamente grande, se comparado à média das respostas sem falhas, esse aumento no tempo médio de resposta diante de falhas é esperado.

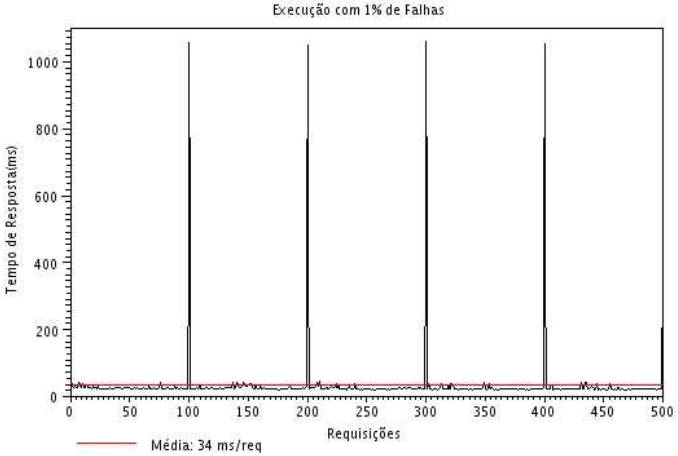

Figura 5.7 – Execução com 1% de falhas

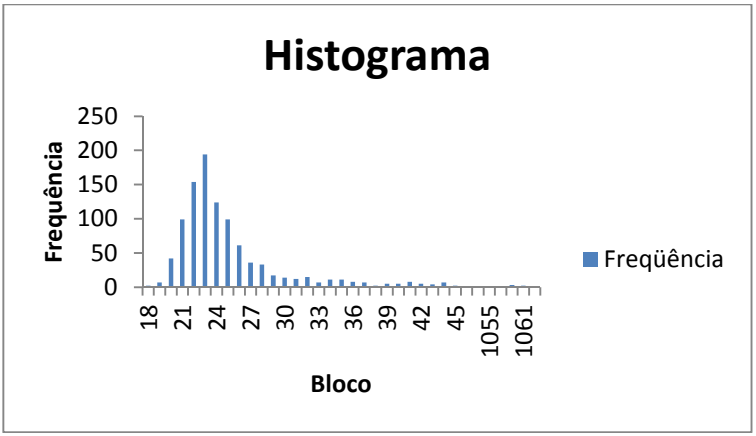

Figura 5.8 – Histograma dos testes com 1% de falhas

5.4.5.3 Execução com 50% de falhas

Neste cenário, a presença de falhas é numerosa, de forma que intrusões ocorrem em metade dos eventos (requisições) no sistema neste experimento. O aumento no tempo médio de respostas é altamente impactante (Figura 5.9) e a distribuição vista no histograma (Figura 5.10) já difere dos histogramas anteriores, contendo uma alta distribuição de tempos de respostas elevados.

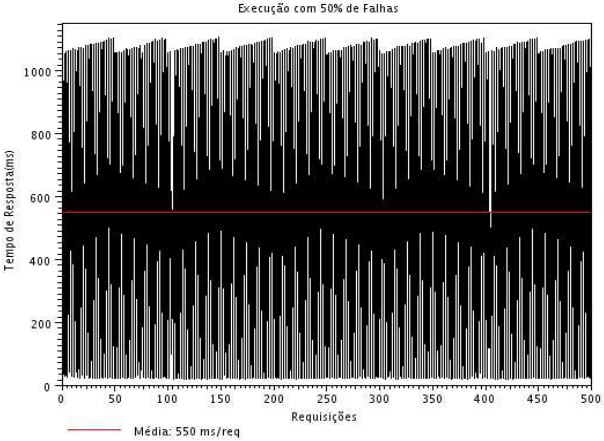

Figura 5.9 – Execução com 50% de falhas

É possível notar também que um efeito dente de serra aparece no gráfico apresentado na Figura 5.9. Em nossos testes, como descrito na seção 5.2, as réplicas efetuam a rotina de *checkpoint* a cada 50 requisições. Definimos N_{exec} como o número de requisições executadas após a execução de um *checkpoint*. Dessa forma, o tempo de atualização de uma réplica é diretamente dependente de N_{exec}, sendo que quanto maior o valor de N_{exec} maior o tempo de atualização da réplica.

Por exemplo, se uma falha ocorrer na requisição de número 51, a réplica nova irá executar somente 1 requisição além do último *checkpoint* para recuperar o estado de execução da ME. Porém, se a falha ocorrer na requisição de número 99, a nova réplica terá que executar 49 requisições após o último *checkpoint* para atualizar o seu estado, causando, dessa forma, diferentes impactos nos tempos de resposta e consequentemente causando o efeito serra no gráfico apresentado.

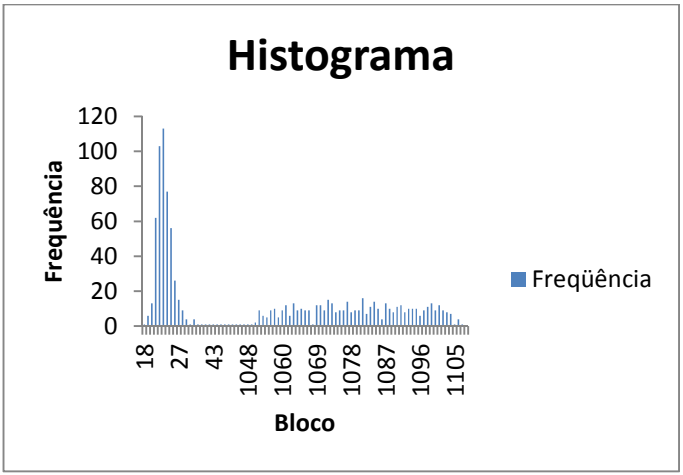

Figura 5.10 - Histograma dos testes com 50% de falhas

5.4.5.4 Execução com 100% de falhas

O pior cenário em nossa abordagem é onde ocorrem falhas no atendimento de todas as requisições do cliente (apresentado nas Figura 5.11 e Figura 5.12).

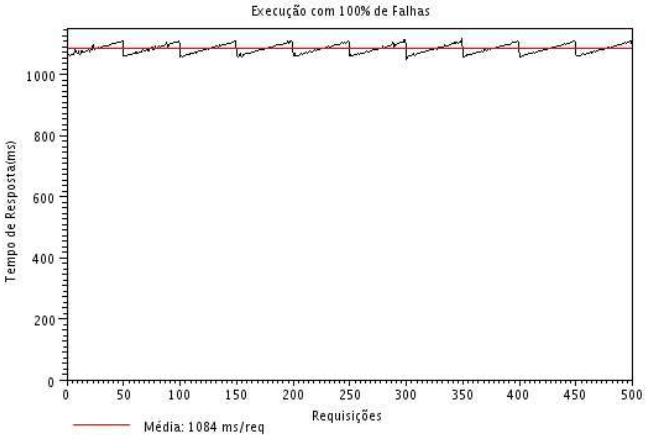

Figura 5.11 – Execução com 100% de falhas

Neste caso, para toda requisição a ser executada pelas réplicas, sempre haverá um desacordo e uma réplica nova deverá ser adicionada ao sistema. Podemos observar que neste cenário o tempo de resposta é

constantemente mantido em um valor alto (1084*ms* em média). O efeito serra também é presente no gráfico, pelos mesmos motivos apresentados no caso anterior.

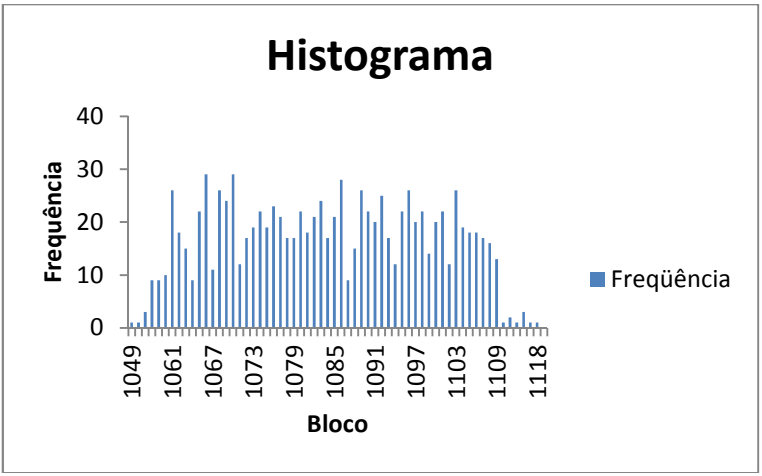

Figura 5.12 - Histograma dos testes com 100% de falhas

5.4.5.5 Impacto das falhas no tempo médio de resposta

Como a apresentação dos gráficos de execução e histogramas de todos os testes efetuados se tornaria repetitiva, optamos por mostrar os resultados de crescimento de tempo de resposta no gráfico apresentado na Figura 5.13.

É possível perceber que, com o aumento do número de falhas, o tempo médio de resposta é profundamente aumentado. Neste gráfico, as variações de tempo de resposta se acentuam significativamente quando o número de invasões (presença de réplicas maliciosas) é incrementado.

Figura 5.13 – Tempo médio de resposta considerando o aumento progressivo de falhas

5.4.6 Clientes simultâneos

Na Figura 5.14 apresentamos os tempos de resposta quando da utilização do sistema por clientes simultâneos.

O tempo de resposta é impactado, pois, com o aumento do número de clientes, a utilização e concorrência sobre os recursos da máquina física são incrementados.

Porém, mesmo em sistemas onde não há qualquer mecanismo de tolerância a intrusão, este efeito de aumento no tempo de resposta a requisições também ocorre.

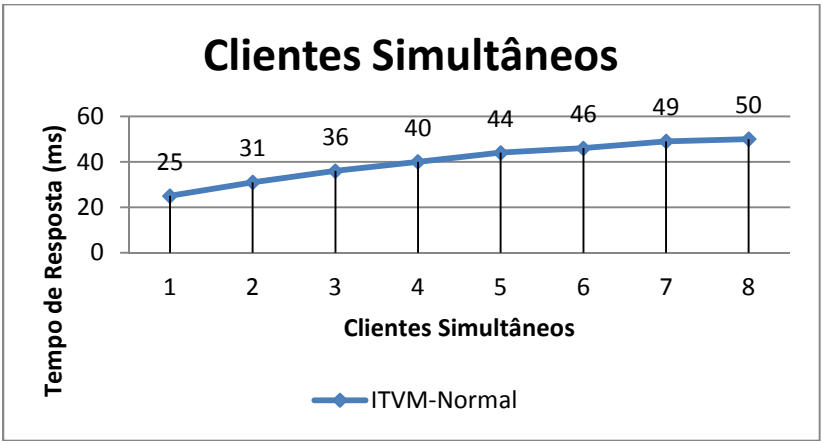

Figura 5.14 – Execução sem Falhas com Clientes simultâneos

5.4.7 Testes Comparativos com Outras Abordagens

Neste cenário comparamos nosso modelo de BFT com a abordagem SMIT que é muito similar ao nosso, por usar máquinas virtuais e memória compartilhada. A comparação prática foi efetuada justamente pela semelhança nas abordagens. A abordagem SMIT é descrita na seção de trabalhos relacionados. No cenário da Figura 5.15 comparamos os tempos médios de resposta dos casos com falhas e sem falhas de ambas as abordagens.

Nossa abordagem tem um melhor desempenho em termos de tempo de resposta. Isso se deve ao fato de envolver configurações com menos réplicas (*f+1*) e, portanto, menos VMs. Com o número de réplicas reduzido, a possibilidade de uma melhor utilização dos recursos do servidor físico melhora o desempenho da replicação.

Outro ponto em que a nossa abordagem apresenta vantagens é na ocorrência de falhas maliciosas. Enquanto o SMIT, na ocorrência de falhas, as réplicas comprometidas continuam na execução do protocolo, influenciando no desempenho do mesmo, em nosso modelo as réplicas maliciosas são continuamente removidas, de forma que o protocolo sempre seja reconfigurado para ser executado com somente $f + 1$ réplicas corretas.

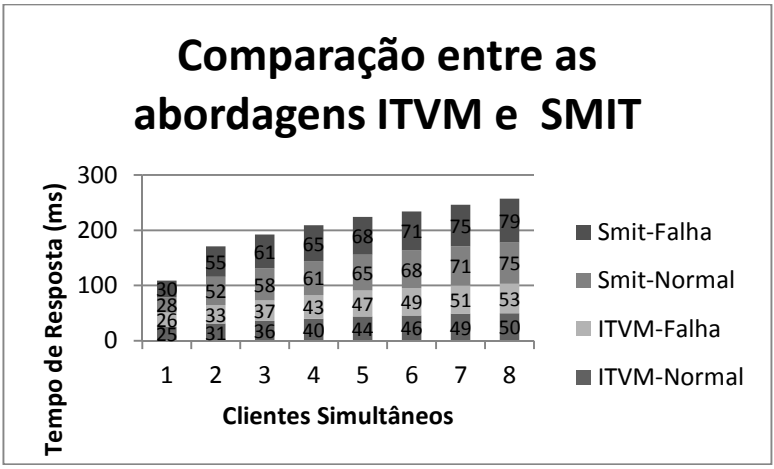

Figura 5.15 – Comparação das abordagens

5.4.7.1 Comparações Qualitativas com Outras Abordagens da Literatura

É ilustrado na Tabela 5.2 a comparação qualitativa de nossa abordagem com as abordagens apresentadas no Capítulo 3.

Tabela 5.2 – Comparações Qualitativas

Protocolo	Rép. Aco	Rép. Exec	Passos N.	Passos F.	Comp. Com.	Virt.
PBFT (CASTRO; LISKOV, 2002a)	*3f + 1*	*3f + 1*	3	6	Não	Não
Zyzzyva (KOTLA *et al.*, 2007)	*2f + 1*	*3f + 1*	1	6	Não	Não
Yin (YIN *et al.*, 2003)	*2f +1*	*3f + 1*	3	6	Não	Não
TTCB (CORREIA, M. *et al.*, 2002)	*f + 2*	*2f + 1*	NA	NA	Sim	Não
MinBFT(VERONESE *et al.*, 2009)	*2f + 1*	*2f + 1*	2	4	Sim	Não
VM-FIT(REISER, Hans P, 2007)	*2f + 1*	*2f + 1*	2	2	Sim	Sim
LBFT (CHUN, B. *et al.*, 2008)	*2f + 1*	*2f + 1*	2	2	Não	Sim
ZZ (WOOD *et al.*, 2011)	*3f + 1*	*f + 1*	3	6	Não	Sim
SMIT (JÚNIOR *et al.*, 2010)	*2f + 1*	*2f + 1*	2	4	Sim	Sim
ITVM	*1 (AS)*	*f + 1*	4	7	Sim	Sim

Embora em termos de número de passos o nosso algoritmo não tenha ganhos em relação aos algoritmos usuais, o uso de memória compartilhada faz com que os custos (se consideramos o *round-trip* requisição/resposta) fiquem extremamente reduzidos quando comparados com implementações em *clusters* ou redes locais.

Obtivemos ainda ganhos na redução do número de réplicas necessárias ($f + 1$), se comparados, por exemplo, aos $3f + 1$ do PBFT (CASTRO; LISKOV, 2002a) e os $2f + 1$ das abordagens MinBFT (VERONESE *et al.*, 2009) e SMIT (JÚNIOR *et al.*, 2010).

Além disso, o uso da virtualização possibilita a redução no número de servidores físicos necessários para a execução do protocolo. Ao invés de se manter um servidor físico para cada réplica de execução do protocolo, em nossa abordagem somente um servidor físico é mantido e este executa as várias réplicas de execução em máquinas virtuais.

5.5 CONSIDERAÇÕES FINAIS

Neste capítulo apresentamos a implementação de um protótipo a fim de avaliar nosso protocolo. Para isso, um serviço replicado foi implementado. Neste serviço, clientes enviam requisições e os servidores de aplicação devem executa-las na mesma ordem em uma replicação ME.

Efetuamos testes de nosso protótipo com diferentes cenários. Em cenários otimistas, onde não ocorrem falhas, assim como em cenários onde as falhas estão presentes. O foco maior de nossos testes foi em cenários com falhas. O objetivo foi sempre de verificar o comportamento de nossa abordagem na presença de diferentes cenários de intrusões no sistema. Como resultados, observamos que com o aumento do número de intrusões, o impacto nos tempos médios de resposta das requisições aos clientes é crescente e dependente do cenário construído. Isso se deve em grande parte ao fato da necessidade da ativação de réplicas *standby* para recompor a replicação da ME em situações de desacordos.

Ainda que o tempo de resposta seja altamente impactado quando o número de intrusões no sistema é frequente, nosso sistema continua funcionando corretamente. Mesmo em situações hipotéticas, como o caso do sistema ser invadido em frequências de períodos menores que 1

segundo, nossa abordagem se manteve em funcionamento, respondendo corretamente às requisições do cliente.

Outro cenário efetuado em nossos testes foi a comparação de nosso protótipo com uma abordagem similar à nossa. Fizemos estas comparações tanto em ambientes com falhas como sem falhas. Observamos que nossa arquitetura é levemente mais eficiente se compararmos os tempos de resposta, porém, em nosso caso, utilizamos somente $f + 1$ réplicas (VMs), ao invés de $2f + 1$ réplicas (VMs) necessárias na abordagem com a qual fizemos nossas comparações.

Em nossa abordagem, as réplicas maliciosas são identificadas e removidas do sistema, de forma que novas intrusões podem ocorrer no sistema ao longo da execução. Em SMIT, somente f servidores podem ser comprometidos e estes permanecerão presentes no ciclo de vida, até que uma reparação seja feita no sistema.

6 CONCLUSÃO E PERSPECTIVAS FUTURAS

Nesta dissertação apresentamos uma abordagem que faz uso de virtualização com finalidade de fornecer suporte de tolerância a intrusões (faltas maliciosas ou bizantinas) para provedores de serviço. Além disso, nossa abordagem de BFT utiliza uma abstração de memória compartilhada no sentido de conseguir custos mais baixos na execução de protocolos de replicação Máquina de Estado (ME) em contexto bizantino.

A abordagem (ITVM), por enfatizar a separação de faltas de *crash* e faltas maliciosas, limita os protocolos de replicação ME que tratam da tolerância a intrusões à execução das réplicas em máquinas virtuais. A separação dos tipos de faltas e também o uso da virtualização, permitem que as requisições de clientes possam ser executadas com um número variável (entre $f+1$ e $2f+1$) de réplicas.

O algoritmo de BFT que definimos nesta abordagem tem uma dependência muito forte em relação ao componente confiável introduzido no nosso modelo: o *Agreement Service* (AS). Se este componente for violado, todo o sistema deixa de funcionar corretamente. A correção deste componente pode ser verificada facilmente devido à simplicidade de suas funções. O isolamento deste componente confiável é garantido na nossa abordagem porque o mesmo fica num domínio isolado das máquinas virtuais que executam as réplicas e inacessível através da rede. Na implementação com o *hypervisor* XEN utilizamos o *domain0* para garantir seu isolamento.

Em nosso modelo, não houve uma preocupação inicial com possíveis ocorrências de falhas de *crash* na máquina física. Porém, pela separação dos modos de falhas (*crash* e intrusões), a simples extensão do nosso modelo para também suportar *crashes* é facilmente concretizada. Basta estabelecer dois níveis no nosso modelo. No nível mais interno estariam as execuções do nosso BFT em máquinas virtuais. E no nível mais externo, com a inclusão de mais uma ou mais máquinas físicas, estariam executando protocolos mais brandos e menos caros, usando uma rede isolada que interligaria estas máquinas para tratar o problema de *crash* na máquina física.

Os objetivos propostos pelo trabalho foram alcançados. Reduzimos o número de servidores físicos através do uso da virtualização e, além disso, diminuímos o número de réplicas

necessárias para a execução do serviço para $f + 1$ com a utilização de um componente confiável.

Os resultados obtidos com os testes efetuados mostraram que nossa abordagem continua sendo executada corretamente mesmo na presença constantes de intrusões. A renovação da replicação fez com que essa possibilidade fosse explorada e alcançada. Com o aumento da quantidade de intrusões no sistema, há o aumento do tempo de resposta necessários para a execução da requisição, porém a segurança do serviço continua sendo preservada.

Ao final do desenvolvimento deste trabalho, identificamos pontos que poderão ser otimizados em trabalhos futuros:

- Extensão do suporte algorítmico proposto para atender os requisitos de tolerância a faltas de parada do servidor físico. No trabalho a preocupação ateve-se apenas a faltas maliciosas que são as mais custosas em termos de algoritmos. Acreditamos que extensões para atender faltas de paradas possam facilmente ser desenvolvidas a partir do suporte já desenvolvido;
- Melhora de desempenho das operações em memória compartilhada, implementando em outros formatos ou mesmo utilizar o formato atual através das interfaces que o *hypervisor* XEN oferece;
- Aprofundar os estudos com relação a execução de *checkpoints* e melhorias nos tempo de transferência de estados;
- Aprofundar os estudos com relação a memória compartilhada, tal como memória distribuída, memória compartilhada distribuída, etc.

O desenvolvimento deste trabalho resultou em duas publicações referentes ao mesmo:

- Lau, J., Barreto, L., & Fraga, J. Infraestrutura Baseada em Virtualização para Serviços Tolerantes a Intrusões. XXX Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos. 2012.
- Lau, J., Barreto, L., & Fraga, J. An Infrastructure based on Virtualization for Intrusion Tolerant Services. IEEE International Conference on Web Service. 2012.

REFERÊNCIAS

ALPERN, B.; SCHNEIDER, F. B. Defining Liveness. Information Processing Letters, v. 21, p. 181-185, 1985.

AVIZIENIS, A.; LAPRIE, J.-C.; RANDELL, B.; LANDWEHR, C. Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable and Secure Computing, v. 1, n. 1, p. 11-33, jan 2004.

AVIZIENIS, Algirdas; KELLY, J. P. J. Fault Tolerance by Design Diversity: Concepts and Experiments. IEEE Computer Society, n. August, p. 67-80, 1984.

BARHAM, P.; DRAGOVIC, B.; FRASER, Keir; *et al.* Xen and the Art of virtualization. ACM SIGOPS Operating Systems Review, v. 37, n. 5, p. 164, dez 2003.

BESSANI, A. N.; OBELHEIRO, R. R.; SOUSA, P.; GASHI, I. On the Effects of Diversity on Intrusion Tolerance. Technical Report, 2008.

CACHIN, C.; GUERRAOUI, R.; RODRIGUES, L. Introduction to Reliable and Secure Distributed Programming. [S.l.]: Springer-Verlag New York Inc, 2006. .

CASTRO, M.; LISKOV, B. Proactive recovery in a Byzantine-fault-tolerant system. Symposium on Operating System Design, v. 4, 2000.

CASTRO, M.; LISKOV, B. Practical Byzantine Fault Tolerance and Proactive Recovery. ACM Transactions on Computer Systems, v. 20, n. 4, p. 398-461, 2002a.

CASTRO, M.; LISKOV, B. Practical byzantine fault tolerance and proactive recovery. ACM Transactions on Computer Systems, v. 20, n. 4, p. 398-461, nov 2002b.

CHARRON-BOST, B. Uniform Consensus is harder than consensus. Journal of Algorithms, v. 51, n. 1, p. 15-37, abr 2004.

CHUN, B.; MANIATIS, P.; SHENKER, S. Diverse replication for single-machine byzantine-fault tolerance. USENIX 2008 Annual Technical, p. 287-292, 2008.

CHUN, B.-gon; MANIATIS, P.; SHENKER, S.; KUBIATOWICZ, J. Attested Append-Only Memory: Making Adversaries Stick to their Word. Proceedings of twenty-first ACM Symposium on Operating Systems Principles SIGOPS, v. 41, n. 6, 2007.

CLARK, C.; FRASER, Keir; HAND, Steven; *et al.* Live migration of virtual machines. Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2. Anais... [S.l.]: USENIX Association. Disponível em: <http://dl.acm.org/citation.cfm?id=1251223>. Acesso em: 14 jan. 2012. , 2005

CORREIA, M; NEVES, N.; LUNG, L.; VERISSIMO, P. Worm-IT – A wormhole-based intrusion-tolerant group communication system. Journal of Systems and Software, v. 80, n. 2, p. 178-197, fev 2007.

CORREIA, M. From Consensus to Atomic Broadcast: Time-Free Byzantine-Resistant Protocols without Signatures. The Computer Journal, v. 49, n. 1, p. 82-96, 1 jul 2005.

CORREIA, M.; NEVES, N. F.; VERISSIMO, P. Efficient Byzantine-resilient reliable multicast on a hybrid failure model. 21st IEEE Symposium on Reliable Distributed Systems, 2002. Proceedings., v. 11583, p. 2-11, 2002.

CORREIA, M. P. Serviços Distribuídos Tolerantes a Intrusões: resultados recentes e problemas abertos. Technical Reports, p. 1-58, 2005.

DOLEV, D.; DWORK, C.; STOCKMEYER, L. On the Minimal Synchronism Needed for Distributed Consensus. Journal of the ACM, v. 34, n. I, p. 77-97, 1987.

DWORK, C.; LYNCH, N.; STOCKMEYER, L. Consensus in the Presence of Partial Synchrony. Journal of the ACM, v. 35, n. 2, p. 288-323, 1988.

DÉFAGO, X.; SCHIPER, A.; URBÁN, P. Total order broadcast and multicast algorithms: Taxonomy and survey. ACM Computing Surveys (CSUR), v. 36, n. 4, p. 372–421, 2004.

FERNANDEZ, A.; JIMENEZ, E.; RAYNAL, M. Electing an Eventual Leader in an Asynchronous Shared Memory System. 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07), p. 399-408, jun 2007.

FISCHER, M. J.; LYNCH, N A; PATERSON, M. S. Impossibility of Distributed Consensus with One Faulty Process. Journal of the ACM (JACM), v. 32, n. 2, p. 374-382, 1985.

FRAGA, J.; POWELL, D. A Fault and Intrusion-Tolerant File System. Proceedings of the 3rd International Conference on Computer Security, v. 203, p. 203-218, 1985.

FRASER, K; HAND, S; NEUGEBAUER, R.; PRATT, I. Safe hardware access with the Xen virtual machine monitor. 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure, 2004.

GUERRAOUI, R.; SCHIPER, André. The Generic Consensus Service. IEEE Transactions on Software Engineering, v. 27, n. 1, p. 29-41, 2001.

HADZILACOS, V.; TOUEG, S. A Modular Approach to Fault-Tolerant Broadcasts and Related Problems. Journal On The Theory Of Ordered Sets And Its Applications, p. 1-84, 1994.

ISHIKAWA, H.; NAKAJIMA, T.; OIKAWA, S.; HIROTSU, T. Proactive Operating System Recovery. Proceedings of the twentieth ACM symposium on Operating systems principles, 2005.

JUNQUEIRA, F. P.; MARZULLO, K. Synchronous Consensus for Dependent Process Failures. Proceedings of the 23rd International Conference on Distributed Computing Systems, 2003.

JÚNIOR, V. S.; LUNG, L. C.; CORREIA, Miguel; FRAGA, J. D. S.; LAU, J. Intrusion Tolerant Services Through Virtualization: A Shared Memory Approach. 24th IEEE International Conference on Advanced Information Networking and Applications, p. 768-774, 2010.

KIHLSTROM, K. P.; MOSER, L. E.; -SMITH, P. M. M. Solving Consensus in a Byzantine Environment Using an Unreliable Fault Detector. In Proceedings of the International Conference on Principles of Distributed Systems, p. 61--75, 1997.

KOTLA, R.; ALVISI, L.; DAHLIN, M.; CLEMENT, A.; WONG, E. Zyzzyva: speculative byzantine Fault Tolerance. ACM SIGOPS Operating Systems Review, v. 41, n. 6, p. 45–58, 2007.

LAMPORT, L. Proving the Correctness of Multiprocess Programs. IEEE Transactions on Software Engineering, v. SE-3, n. 2, p. 125-143, mar 1977.

LAMPORT, Leslie. Paxos Made Simple. ACM SIGACT News, v. 32, n. 4, p. 18–25, 2001.

LAMPORT, Leslie; SHOSTAK, R.; PEASE, M. The Byzantine Generals Problem. ACM Transactions on Programming Languages and Systems, v. 4, n. 3, p. 382-401, 1982.

LAPRIE, J.-C. Dependable Computing and Fault Tolerance: Concepts and Terminology. Twenty-Fifth International Symposium on In Fault-Tolerant Computing, v. 2, 1995.

LITTLEWOOD, B.; STRIGINI, L. Redundancy and Diversity in Security. European Symposium on Research Computer Security, p. 423--438, 2004.

LYNCH, Nancy A. Distributed Algorithms. [S.l.]: Morgan Kaufmann Publishers Inc, 1996.

MARSHALL PEASE, ROBERT SHOSTAK, L. L. Reaching Agreement in the Presence of Faults. Journal of the ACM, v. 27, n. 2, p. 228-234, 1980.

NANDA, S.; CHIUEH, T. A survey of virtualization technologies. Technical Report, p. 1-42, 2005.

POLEDNA, S. Deterministic Operation of Dissimilar Replicated Task Sets in Fault-Tolerant Distributed Real-Time Systems. Research Report, 1996.

POPEK, G. J.; GOLDBERG, R. P. Formal Requirements for Virtualizable Third Generation Architectures. Communications of the ACM, v. 17, n. 7, p. 412-421, 1974.

RAYNAL, M. A Short Introduction to Failure Detectors for Asynchronous Distributed Systems. SIGACT News, v. 36, n. 1, p. 53-70, 2005.

REISER, H.; HAUCK, F.; KAPITZA, R.; SCHRÖDER-, W. Hypervisor-based redundant execution on a single physical host. Proc. of the 6th European Dependable Computing Conference, 2006.

REISER, Hans P. VM-FIT: Supporting Intrusion Tolerance with Virtualization Technology. Workshop on Recent Advances on Intrusion-Tolerant Systems, 2007.

REISER, Hans P; LISBOA, U. D. Fault and Intrusion Tolerance on the Basis of Virtual Machines. Proceedings of the 1st Virtualization Expert Meeting, 2008.

REISER, H.P.; KAPITZA, R. Hypervisor-Based Efficient Proactive Recovery. 2007 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007), p. 83-92, out 2007.

ROBERT P GOLDBERT. Survey of Virtual Machine Research. IEEE Computer Society Press, v. 7, n. 6, p. 34-45, 1974.

ROSENBLUM, M. The Reincarnation of Virtual Machines. Queue Magazine - Virtual Machines, v. 2, n. 5, p. 34-40, 2004.

SAHOO, J.; MOHAPATRA, S.; LATH, R. Virtualization: A Survey on Concepts, Taxonomy and Associated Security Issues. Second International Conference on Computer and Network Technology, p. 222-226, abr 2010a.

SAHOO, J.; MOHAPATRA, S.; LATH, R. Virtualization: A Survey on Concepts, Taxonomy and Associated Security Issues. 2010 Second International Conference on Computer and Network Technology, p. 222-226, abr 2010b.

SCHNEIDER, F. B. Implementing Fault-Tolerant Approach: A Tutorial Services Using the State Machine. ACM Computing Surveys, v. 22, n. 4, p. 299--319, 1990.

STALLINGS, W. Cryptography and network security: principles and practices. 4th. ed. [S.l.]: Prentice Hall, 2006.

VERONESE, G. S.; CORREIA, Miguel; BESSANI, A. N.; LUNG, L. C.; VERISSIMO, Paulo. Minimal Byzantine fault tolerance: Algorithm and evaluation. Technical Reports, 2009.

VERÍSSIMO, P.; NEVES, N.; CORREIA, M. Intrusion-tolerant architectures: Concepts and design. Architecting Dependable Systems, v. 11583, p. 3–36, 2003.

WOOD, T.; SINGH, R.; VENKATARAMANI, A.; SHENOY, P.; CECCHET, E. ZZ and the Art of Practical BFT Execution. Proceedings of the sixth conference on Computer Systems EuroSys '11, p. 123-138, 2011.

XEN. The Xen® hypervisor. Disponível em: <Http://xen.org/>.

YIN, J.; MARTIN, J.-P.; VENKATARAMANI, A.; ALVISI, L.; DAHLIN, M. Separating agreement from execution for byzantine fault tolerant services. ACM SIGOPS Operating Systems Review, v. 37, n. 5, p. 253, 1 dez 2003.

ZIELIŃSKI, P. Paxos at war. In Proceedings of the 2001 Winter Simulation Conference, n. 593, 2004.