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

Fernando Dettoni

TWINBFT: TOLERÂNCIA A FALTAS BIZANTINAS COM MÁQUINAS VIRTUAIS GÊMEAS

Florianópolis

2013

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

TWINBFT: TOLERÂNCIA A FALTAS BIZANTINAS COM MÁQUINAS VIRTUAIS GÊMEAS

Monografia submetida ao Programa de Pós-Graduação em Ciência da Computação como parte dos requisitos para a obtenção do Grau de Mestre em Ciência da Computação.

Orientador: Prof. Dr. Lau Cheuk Lung
Candidato: Fernando Dettoni

Florianópolis

2013

Fernando Dettoni

TWINBFT: TOLERÂNCIA A FALTAS BIZANTINAS COM MÁQUINAS VIRTUAIS GÊMEAS

Esta Monografia foi julgada aprovada para a obtenção do Título de "Mestre em Ciência da Computação", e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

Florianópolis, 08 de Agosto 2013.

Prof Dr. Ronaldo Dos Santos Mello
Coordenador do Curso

Prof. Dr. Lau Cheuk Lung
Orientador

Banca Examinadora:

Prof. Dr. Lau Cheuk Lung
Presidente

Prof. Dr. Carlos Alberto Maziero

Prof. Dr. Mario Antonio Ribeiro Dantas

Prof. Dr. Michelle Silva Wangham

RESUMO

Visando suprir a necessidade de segurança no funcionamento de sistemas computacionais, diversas abordagens tolerantes a faltas bizantinas foram criadas. Apesar de terem fins práticos, a maior parte destas abordagens ainda apresenta um fraco desempenho ou requisitos que limitam seu uso em boa parte dos cenários reais. Neste trabalho de pesquisa é apresentada uma arquitetura e um algoritmo para replicação de máquina de estados tolerante a faltas bizantinas usando virtualização. A virtualização, apesar de existir há mais de 30 anos, vem se tornando cada vez mais comum recentemente, sendo muito utilizada em aplicações de computação em nuvens. São exploradas as vantagens fornecidas pela virtualização para detectar e tolerar réplicas faltosas, de forma a transformar ou reduzir faltas bizantinas em faltas de omissão. Com esta transformação, a abordagem apresentada é capaz de reduzir o número total de réplicas físicas necessárias de $3f+1$, em abordagens tradicionais, para $2f+1$. Esta abordagem se baseia no conceito de máquinas virtuais gêmeas, ou seja, na execução de um conjunto de máquinas virtuais em cada máquina física, cada uma funcionando de uma certa forma como um detector de falhas de sua gêmea, a partir da validação das mensagens enviadas. Neste contexto, um protótipo foi implementado e alguns experimentos foram realizados para obter medidas do desempenho da abordagem em uma execução prática.
Palavras-chave: Tolerância a faltas Bizantinas; Tolerância a intrusões; Replicação Máquina de Estados; Sistemas Distribuídos; Virtualização

ABSTRACT

Aiming to supply the need for security in information systems, a lot of approaches were proposed. Despite of being practical, most part of these approaches still lack in performance or have too strong requirements. We present an architecture and an algorithm for Byzantine fault-tolerant state machine replication using virtualization. Despite of existing for more than 30 years, virtualization is becoming more common, mainly because of cloud computing applications. Our algorithm explores the advantages of virtualization to reliably detect and tolerate faulty replicas, allowing the transformation of Byzantine faults into omission faults. Our approach reduces the total number of physical replicas from $3f + 1$, in traditional approaches, to $2f + 1$. Our approach is based on the concept of twin virtual machines, where there are a set of virtual machines in each physical host, each one acting as failure detector of its twin, by the validation of the messages sent.
Keywords: Byzantine fault tolerance; Intrusion tolerance; State Machine Seplication; Distributed systems; Virtualization

LISTA DE FIGURAS

Figura 1 Modelos de falhas de acordo com a severidade.......... 30
Figura 2 Execução normal do algoritmo PBFT.................. 40
Figura 3 Operação normal proposta em Malkhi e Reiter (1997).. 55
Figura 4 Interação entre réplicas em Kihlstrom, Moser e Melliar-Smith (2003).. 56
Figura 5 Detectores de mudez em operação sem falhas........... 59
Figura 6 Detectores de mudez em operação com falhas.......... 60
Figura 7 Arquitetura da abordagem em Haeberlen, Kouznetsov e Druschel (2006).. 62
Figura 8 TwinBFT - Arquitetura com Máquinas Virtuais Gêmeas. 75
Figura 9 Passos de comunicação do protocolo em execução normal. 79
Figura 10 Passos de comunicação do protocolo em execução com duas falhas.. 85
Figura 11 Arquitetura implementada para análise de desempenho. 93
Figura 12 Latência em execução normal......................... 96
Figura 13 Comparação de throughput em execução normal....... 97
Figura 14 Histograma da latência em execução normal............ 98
Figura 15 Distribuição da latência em execução normal........... 98
Figura 16 *Throughput* em execução normal com múltiplos clientes. 99
Figura 17 Impacto das falhas na latência........................ 100
Figura 18 Histograma da latência com 1% de faltas............... 101
Figura 19 Distribuição da latência com 1% de faltas.............. 101
Figura 20 Distribuição da latência com 5% de faltas.............. 102
Figura 21 Histograma da latência com 5% de faltas............... 103

LISTA DE TABELAS

Tabela 1 Comparação Entre Propriedades dos Protocolos Avaliados 72
Tabela 2 Comparação Entre Propriedades dos Protocolos Avaliados 103

LISTA DE ALGORITMOS

1 Lado Cliente 82
2 Algoritmo em operação normal: Tarefa 1 - rede 83
3 Algoritmo em operação normal: Tarefa 2 - postbox 84
4 Algoritmo de troca de visão: Tarefa 1 - rede 86
5 Algoritmo de troca de visão: Tarefa 2 - postbox 87
6 Algoritmo de troca de visão: Tarefa 3 - timeout 87
7 Algoritmo de coleta de lixo: Tarefa 1 - checkpoint 88
8 Algoritmo de coleta de lixo: Tarefa 2 - postbox 88
9 Algoritmo de coleta de lixo: Tarefa 3 - rede 88

LISTA DE ABREVIATURAS E SIGLAS

SMR State Machine Replication ... 23
BFT Bizantine Fault Tolerance ... 23
PBFT Practical Bizantine Fault Tolerance ... 23
VM Virtual Machine ... 23
CPU Central Processing Unit ... 24
TwinBFT Twin Byzantine Fault Tolerance ... 24
VMM Virtual Machine Monitor ... 36
FIFO First In, First Out ... 46
MinBFT Minimum Bizantine Fault Tolerance ... 48
USIG Unique Sequential Identifier Generator ... 48
SMIT Shared Memory Intrusion Tolerance ... 69
KVM Kernel Virtual Machine ... 94

SUMÁRIO

1 INTRODUÇÃO 23
1.1 MOTIVAÇÃO 23
1.2 PROPOSTA DO TRABALHO 24
1.3 OBJETIVOS 25
1.3.1 Objetivo Geral 25
1.3.2 Objetivos Específicos 25
1.4 ORGANIZAÇÃO DO TEXTO 25
2 CONCEITOS BÁSICOS EM SISTEMAS DISTRIBUÍDOS 27
2.1 AMBIENTE DE COMPUTAÇÃO DISTRIBUÍDA 27
2.1.1 Modelo de Sistema 28
2.1.2 Modelo de Interação 28
2.1.3 Tipos e Modelos de Falhas 29
2.2 TOLERÂNCIA A FALTAS E INTRUSÕES 30
2.2.1 Replicação de Máquina de Estados 31
2.2.2 Problemas de Acordo 32
2.2.2.1 Difusão com Ordem Total 32
2.2.2.2 Consenso 32
2.3 DETECTORES DE FALHAS 33
2.3.1 Detectores de Falhas de Mudez 34
2.4 TECNOLOGIAS DE VIRTUALIZAÇÃO 34
2.4.1 Conceitos Gerais 35
2.4.2 Tipos de Virtualização 36
2.5 CONSIDERAÇÕES FINAIS 37
3 TRABALHOS RELACIONADOS 39
3.1 ABORDAGENS TRADICIONAIS 39
3.1.1 Practical Byzantine fault tolerance - PBFT 39
3.1.2 Separating Agreement from Execution for Byzantine Fault Tolerant Services 43
3.2 ABORDAGENS HÍBRIDAS 46
3.2.1 BFT-TO: Intrusion Tolerance with Less Replicas 46
3.2.2 Efficient Byzantine Fault Tolerance - MinBFT e Min Zyzzyva 48
3.2.3 Zyzzyva: speculative Byzantine fault tolerance 50
3.3 ABORDAGENS BASEADAS EM DETECTORES DE FALHAS 53

3.3.1 Unreliable Intrusion Detection in Distributed Computations . . . 53
3.3.2 Solving Consensus in a Byzantine Environment Using an Unreliable Fault Detector . . . 55
3.3.3 Muteness Failure Detectors . . . 58
3.3.4 The case for Byzantine fault detection . . . 61
3.3.5 Enhanced Fault-Tolerance through Byzantine Failure Detection . . . 64
3.4 ABORDAGENS BASEADAS EM VIRTUALIZAÇÃO . . . 66
3.4.1 VM-FIT: Supporting Intrusion Tolerance with Virtualisation Technology . . . 66
3.4.2 ZZ and the Art of Practical BFT Execution . . . 67
3.4.3 Intrusion Tolerant Services Through Virtualization: A Shared Memory Approach . . . 69
3.5 CONSIDERAÇÕES FINAIS . . . 71
4 REPLICAÇÃO DE MÁQUINA DE ESTADOS TOLERANTE A FALTAS BIZANTINAS USANDO MÁQUINAS VIRTUAIS GÊMEAS . . . 73
4.1 MODELO DO SISTEMA . . . 74
4.2 MEMÓRIA COMPARTILHADA . . . 77
4.3 PROTOCOLO TWINBFT . . . 78
4.3.1 Propriedades . . . 79
4.3.2 Algoritmo Proposto . . . 80
4.3.3 Comportamento do Cliente . . . 81
4.3.4 Execução Normal do Protocolo . . . 81
4.3.5 Troca de Visão . . . 85
4.3.6 Coleta de Lixo . . . 87
4.4 CORREÇÃO DO ALGORITMO . . . 89
4.5 CONSIDERAÇÕES FINAIS . . . 91
5 RESULTADOS EXPERIMENTAIS E ANÁLISE . . . 93
5.1 AMBIENTE DE TESTES . . . 93
5.2 MEMÓRIA COMPARTILHADA . . . 94
5.3 CONSIDERAÇÕES GERAIS . . . 95
5.4 RESULTADOS E ANÁLISE . . . 95
5.5 EXECUÇÃO NORMAL . . . 97
5.5.1 Múltiplos Clientes . . . 97
5.6 EXECUÇÃO COM FALHAS . . . 99
5.6.1 Execução com 1% de Faltas . . . 100
5.6.2 Execução com 5% de Faltas . . . 102
5.7 COMPARAÇÃO QUALITATIVA COM OUTRAS ABORDAGENS . . . 103

5.8 CONSIDERAÇÕES FINAIS 104
6 CONCLUSÕES E PERSPECTIVAS FUTURAS 105
Referências Bibliográficas 107

1 INTRODUÇÃO

1.1 MOTIVAÇÃO

Cada vez mais, sistemas computacionais são usados em aplicações críticas e por isso necessitam operar corretamente apesar da presença de faltas. Estas faltas podem ser acidentais, como faltas de parada (*crash*) ou faltas arbitrárias, também chamadas de faltas bizantinas (LAMPORT; SHOSTAK; PEASE, 1982). Para garantir que o sistema funcione corretamente na presença de faltas arbitrárias é necessário o desenvolvimento de técnicas e algoritmos tolerantes a faltas bizantinas. Uma das técnicas mais utilizadas é a *replicação de máquina de estados* (SMR) (SCHNEIDER, 1990), que utiliza máquinas de estados determinísticas para oferecer um serviço replicado. Muitas abordagens tolerantes a faltas bizantinas (BFT) foram desenvolvidas utilizando SMR (e.g. (CASTRO; LISKOV, 1999b; YIN et al., 2003; KOTLA et al., 2008)). Dentre estas, o algoritmo PBFT (CASTRO; LISKOV, 1999b) é frequentemente considerado um patamar, pois foi o primeiro algoritmo com desempenho suficiente para muitas aplicações práticas.

Outra técnica de tolerância a faltas é o uso de *detectores de falhas não-confiáveis* (CHANDRA; TOUEG, 1996). Apesar de originalmente criada para tolerar faltas de *crash*, algumas propostas posteriores foram capazes de estender a ideia para suportar faltas bizantinas (DOUDOU et al., 1999; KIHLSTROM; MOSER; MELLIAR-SMITH, 2003). Estes detectores de falhas ajudam outro algoritmo fornecendo dicas sobre réplicas que aparentam estar faltosas.

A *virtualização* pode também ser considerada uma técnica de tolerância a faltas bizantinas pois introduz um nível de isolamento entre máquinas virtuais (VM). Diversos trabalhos usam virtualização com o objetivo de proteger alguns componentes de falhas (ou ataques) de outros componentes (JIANG; WANG, 2007; GARFINKEL; ROSENBLUM, 2003; LAUREANO; MAZIERO; JAMHOUR, 2004). Tecnologias de virtualização são bastante aceitas pela indústria, e são largamente utilizadas atualmente, por exemplo por serviços de computação em nuvens como Amazon Web Services e Windows Azure.

O PBFT e vários outros algoritmos de replicação de máquinas de estados BFT têm um alto custo de implementação pois possuem uma resiliência de $n \geq 3f + 1$, ou seja, precisam de $n > 3f$ réplicas para tolerar f réplicas faltosas. Para diminuir esse custo, surgiram

algumas abordagens que usam um *componente confiável* para limitar o comportamento das réplicas faltosas usando apenas $n \geq 2f+1$ réplicas (CORREIA; NEVES; VERISSIMO, 2004; CHUN et al., 2007; VERONESE et al., 2013). Foram propostas também abordagens para executar apenas $f+1$ réplicas, mantendo outras $2f$ *em espera*, ou seja, sem consumir tempo de CPU mas sendo ativadas em caso de falha (DISTLER et al., 2011; WOOD et al., 2011).

1.2 PROPOSTA DO TRABALHO

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

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

A arquitetura e o algoritmo propostos não pretendem oferecer a solução ideal para todos os casos. A utilização de máquinas virtuais é adequada para empresas abertas a esse tipo de tecnologia, como as que usam computação em nuvens. É também indicada quando não se tem acesso à componentes confiáveis ou não se possa esperar pela ativação

de réplicas em espera em caso de falha. Nesses casos, uma solução para replicação de máquinas de estados BFT eficiente – com apenas $2f+1$ réplicas físicas ($4f+2$ virtuais) – usando virtualização pode ser adequada.

1.3 OBJETIVOS

1.3.1 Objetivo Geral

Demonstrar a possibilidade de melhoria na resiliência em algoritmos tolerante a faltas bizantinas a partir da redução do número de réplicas e da redução do número de passos de comunicação entre as réplicas em uma infraestrutura utilizando replicação de máquina de estados e vitualização.

1.3.2 Objetivos Específicos

A partir do objetivo geral, podemos definir alguns objetivos específicos:

- Apresentar uma base teórica e discutir com base em trabalhos relacionados o estado da arte em tolerância a faltas bizantinas.
- Especificar um protocolo de replicação de máquina de estados tolerante a faltas bizantinas, reduzindo o número de réplicas necessárias de $3f+1$ para $2f+1$.
- Comprovar a validade do protocolo proposto a partir da utilização de provas de correção.
- Analisar o protocolo proposto com base no tempo necessário para execução de requisições e na quantidade de requisições executadas em uma unidade de tempo.

1.4 ORGANIZAÇÃO DO TEXTO

Esta dissertação é composta por 6 capítulos. Neste capítulo foi apresentada uma introdução com as motivações para o desenvolvimento desta abordagem, além dos seus objetivos e uma apresentação da pro-

posta do trabalho. Os capítulos restantes deste trabalho estão organizados da seguinte maneira:

- **Capítulo 2 - Conceitos Básicos em Sistemas Distribuídos:** Neste capítulo serão apresentados os conceitos básicos que foram utilizados no decorrer na dissertação, formando assim uma base teórica para o desenvolvimento da abordagem.
- **Capítulo 3 - Trabalhos Relacionados:** Serão apresentados neste capítulo alguns trabalhos que tiveram seus conceitos estudados, e apresentam de alguma forma uma relação com a abordagem apresentada.
- **Capítulo 4 - Replicação de Máquina de Estados Tolerante a Faltas Bizantinas usando Máquinas Virtuais Gêmeas:** Será apresentado neste capítulo todos os detalhes sobre a arquitetura e o algoritmo propostos neste trabalho, sendo exposto o modelo de sistema com as premissas para o funcionamento e os algoritmos relacionados.
- **Capítulo 5 - Resultados Experimentais e Análise:** Neste capítulo estão presentes algumas análises e considerações sobre o desempenho da abordagem, obtidos a partir da implementação de um protótipo da arquitetura proposta. Uma comparação qualitativa é apresentada também entre a abordagem proposta e outras abordagens na literatura.
- **Capítulo 6 - Conclusões e Perspectivas Futuras**: Por fim, este capítulo apresenta as conclusões obtidas, bem como alguns pontos que podem ser melhor trabalhados no futuro.

2 CONCEITOS BÁSICOS EM SISTEMAS DISTRIBUÍDOS

Sistemas distribuídos se dão a partir da utilização de sistemas autônomos com foco na resolução de um problema em comum. Estes sistemas, normalmente interligados por uma rede de comunicação, trocam mensagens entre si para que seja possível a realização de tarefas globais. Neste contexto, a confiança no funcionamento do sistema se torna um problema mais latente já que a natureza distribuída do sistema incrementa a chance de acontecerem falhas.

Este capítulo tem como objetivo apresentar uma visão geral sobre os conceitos utilizados como base no decorrer do trabalho. Na Seção 2.1 serão apresentados conceitos básicos sobre a execução de sistemas distribuídos, bem como noções de falhas relacionadas com este modelo. Na sequência, na Seção 2.2 são apresentados conceitos relevantes para a área tolerância a faltas, especialmente faltas bizantinas que são o foco deste trabalho. Alguns conceitos relevantes à detecção de falhas são apresentados na Seção 2.3. Devido a este trabalho se basear profundamente em técnicas de virtualização, alguns conceitos sobre o tema são introduzidos na Seção 2.4. Por fim, são apresentadas algumas considerações finais na Seção 2.5.

2.1 AMBIENTE DE COMPUTAÇÃO DISTRIBUÍDA

Sistemas distribuídos são definidos como sistemas em que múltiplos dispositivos físicos estão conectados a uma rede de computadores e se comunicam apenas trocando mensagens por esta rede. Devido a esta natureza de dispersão espacial, sistemas distribuídos apresentam diversas características em comum, como (COULOURIS et al., 2011):

- Ausência de relógio global: por estarem geograficamente separados, os participantes de um sistema distribuído não possuem um relógio global, e isso gera desafios na sincronização entre as réplicas.

- Independência de falhas: uma falha em uma parte de um sistema distribuído normalmente não significa uma falha geral no sistema como aconteceria em um sistema centralizado. Isso traz oportunidades para técnicas com a pretensão de manter um sistema funcionando apesar de falhas.

Um sistema distribuído consiste de nodos que possuem uma memória local e executam um ou mais processos. Um processo consiste de uma representação lógica capaz de realizar computações no sistema, sendo a execução de um algoritmo em um determinado processador (ATTIYA; WELCH, 2004; GUERRAOUI; SCHIPER, 1997; LYNCH, 1996). Um processo pode ser entendido como um autômato composto por vários estados, sendo que os estados vão se alterando conforme a execução do programa. A inicialização de um programa distribuído se dá quando todos os processos estão em seu estado inicial e seus canais de comunicação estão vazios.

2.1.1 Modelo de Sistema

Um Modelo de Sistema descreve as propriedades sobre as quais o sistema está fundamentado, desde as premissas básicas até as características fundamentais do sistema. Essas premissas são definidas ainda no projeto de um sistema distribuído e são importantes para delimitar um escopo de funcionamento para o sistema. O modelo de sistema geralmente engloba diversos submodelos que definem hipóteses específicas do funcionamento do sistema, como por exemplo: modelo de interação, modelo de processos e modelo de falhas (COULOURIS et al., 2011).

2.1.2 Modelo de Interação

O modelo de interação tem por objetivo definir todos os parâmetros relacionados com a comunicação entre os processos participantes do sistema distribuído. Esta comunicação é principalmente afetada por dois fatores (COULOURIS et al., 2011): o desempenho da comunicação é usualmente um limitador do sistema e é impossível manter uma noção global do tempo.

Assim, sistemas distribuídos são divididos basicamente entre o modelo de interação síncrono e assíncrono. Sistemas distribuídos síncronos são definidos por terem um limite máximo de tempo para processar e entregar mensagens e por possuírem relógios locais em que a variação para o tempo real esteja dentro de um limite conhecido.

Sistemas assíncronos são aqueles em que não se pode fazer nenhuma afirmação sobre o atraso de processamento ou entrega de mensagens, ou sobre atrasos no relógio local. Nos últimos vinte anos, mode-

los parcialmente assíncronos vem sendo empregados de forma a resolver problemas que requerem sincronismo e ainda assim aproveitar parte das vantagens oferecidas pelos modelos assíncronos (HADZILACOS; TOUEG, 1994).

2.1.3 Tipos e Modelos de Falhas

Modelo de Falhas descreve basicamente sob quais aspectos um sistema pode falhar, ou seja, desviar de sua especificação e quais os efeitos destas falhas no sistema. Os tipos de falhas mais utilizados na literatura são (HADZILACOS; TOUEG, 1994):

- Falhas de omissão:
  - Falha de *crash*: ocorre quando o sistema passa ao um estado em que nenhuma ação será mais executada, como no desligamento da máquina executando o processo.
  - Falha de omissão no envio: ocorre quando o processo envia uma mensagem m, mas esta não é colocada no *buffer* de saída do processo.
  - Falha de omissão de recepção: ocorre quando uma mensagem é colocada no *buffer* de entrada de um processo, porém este processo não recebe a mensagem.
- Falhas arbitrárias (LAMPORT; SHOSTAK; PEASE, 1982):
  - Bizantinas: é uma falha caracterizada pelo comportamento arbitrário de uma entidade, podendo ser causada por situações não-intencionais ou devido a implicações maliciosas em que um atacante mal intencionado tenta tirar proveito do sistema distribuído a partir da manipulação do funcionamento dos processos.
  - Bizantinas com Autenticação: enquanto o modelo de falhas bizantinas não faz nenhuma restrição em relação aos comportamentos que podem ser induzidos, neste modelo considera-se que apesar do processo poder apresentar uma falta arbitrária, está disponível no sistema um mecanismo confiável que autentica cada réplica e não permite que uma se passe por outra.
- Falha de Mudez: são definidas como um tipo de falhas mais genérico do que falhas de omissão, porém mais específicas do que

falhas bizantinas (podendo ser consideradas um subconjunto de faltas bizantinas). Uma falha de mudez ocorre quando um processo é considerado mudo, ou seja, quando este processo para de enviar mensagens antes do previsto, de acordo com o algoritmo sendo executado.

Figura 1: Modelos de falhas de acordo com a severidade.

Na Figura 1 pode-se verificar os modelos de falhas de acordo com a sua severidade. A seta apontada para cima indica um maior grau de severidade, assim sendo, as falhas de parada são as menos problemáticas enquanto as falhas bizantinas se apresentam como as mais severas.

2.2 TOLERÂNCIA A FALTAS E INTRUSÕES

Dependabilidade é definida como a habilidade de um sistema prestar um serviço de forma confiável (LAPRIE, 1995). Apesar deste conceito ser influenciado por muitos fatores, abordagens de tolerância a faltas e intrusões auxiliam na obtenção da dependabilidade de um sistema distribuído. O modelo mais tradicional de tolerância a faltas consiste em replicar um serviço de forma que as réplicas falhem independentemente, para que o conjunto continue operando normalmente mesmo na presença de falhas (BUDHIRAJA et al., 1993).

Dois tipos básicos de replicação são normalmente encontrados: a *replicação primário-backup* e a *replicação ativa*. A replicação *primário-backup* se identifica por manter uma réplica responsável por atender os clientes enquanto outras réplicas redundantes apenas mantêm seu

estado atualizado com a réplica primária a espera de uma falha na réplica primária. Quando isso acontece, uma das réplicas *backup* assume como réplica primária para manter o funcionamento do sistema. Este modelo se encaixa com faltas de *crash*, em que não temos um elemento malicioso (BUDHIRAJA et al., 1993).

A replicação ativa, também conhecida como replicação de máquina de estados é composta de um conjunto de réplicas, sendo que todas as réplicas executam todas as requisições e algum tipo de consenso é empregado para definir a resposta da requisição (SCHNEIDER, 1990). Este modelo é utilizado atualmente como base para diferentes abordagens tolerantes a faltas.

2.2.1 Replicação de Máquina de Estados

O modelo de replicação de Máquinas de Estados, ou *State Machine Replication* (SMR), tornou-se bastante popular sendo utilizado como base para o desenvolvimento de inúmeras abordagens tolerantes a faltas, inclusive faltas maliciosas ou bizantinas. Um exemplo deste uso está na abordagem PBFT (CASTRO; LISKOV, 1999a) que é considerada um marco entre as abordagens tolerantes a faltas bizantinas.

Esta técnica foi formalizada por Schneider (1990) em (SCHNEIDER, 1990) e é definida pela utilização de máquinas de estados determinísticas em cada uma das réplicas empregadas no sistema. Assim, cada uma das réplicas do conjunto se encontrarão em um mesmo estado após a execução de um mesmo conjunto ordenado de requisições, desde que todas tenham partido de um mesmo estado inicial. Estes requisitos formam o conceito definido como *Replica Coordination* que pode ser decomposto em:

- **Acordo:** toda réplica correta recebe todas as requisições.
- **Ordem:** toda réplica correta do conjunto processa as requisições em uma mesma ordem.

De acordo com Poledna (1994), as réplicas são determinísticas se apresentam saídas consistentes assumindo que todas compartilharam um mesmo estado inicial e executaram as mesmas requisições de um serviço em um determinado espaço de tempo (POLEDNA, 1994). Desta forma, o maior desafio da técnica Replicação de Máquina de Estados está na elaboração de protocolos que garantam o Acordo e a Ordem Total. Isto é endereçado neste trabalho, como também em muitas outras abordagens da literatura (CASTRO; LISKOV, 1999a; YIN et al., 2003;

CHUN et al., 2007; KOTLA et al., 2008; VERONESE et al., 2013).

2.2.2 Problemas de Acordo

Problemas de acordo se caracterizam pela necessidade de um sistema distribuído manter um estado global consistente, comportando-se assim como um sistema único. Dentre os diversos problemas de acordo apresentados na literatura, podemos destacar do ponto de vista deste trabalho a Difusão com Ordem Total e o Consenso.

2.2.2.1 Difusão com Ordem Total

Também conhecida como Difusão Atômica, a Difusão com Ordem Total garante uma entrega confiável das mensagens em um grupo de processos, além de garantir que estas mensagens sejam entregues na mesma ordem em que foram enviadas. Uma Difusão Atômica é caracterizada basicamente pelas seguintes propriedades (CHARRON-BOST, 2001):

- Validade: se um processo correto p envia a mensagem m, em algum momento esta mensagem será recebida por algum processo.
- Acordo: se a mensagem m é entregue por um processo p, todos os processos corretos participantes entregarão m em algum momento.
- Integridade: um processo p entrega m no máximo uma vez e apenas se esta foi previamente difundida.
- Ordenação Total: se um processo p recebe uma mensagem m antes de uma segunda mensagem m', então outro processo q receberá m antes de m'.

2.2.2.2 Consenso

O problema do Consenso está fundamentado na necessidade de um grupo de processos concordar com um mesmo valor relacionado a um conjunto de valores propostos (HADZILACOS; TOUEG, 1994). Seu objetivo é garantir que diferentes réplicas de um mesmo algoritmo produzam a mesma computação. Cada processo iniciando com um conjunto inicial de valores, deve concordar com um destes valores de forma

irrevogável a partir da comunicação com outros processos (CHARRON-BOST, 2001).

Formalmente, este problema pode ser definido a partir das seguintes propriedades:

- Acordo: não há dois processos corretos com diferentes decisões.
- Validade: se todos os processos iniciam com um determinado conjunto de valores, então a única decisão possível é algum destes valores.
- Terminação: todo processo chega a uma decisão em algum momento.

O problema do consenso pode ser facilmente resolvido com a Difusão Atômica. Isso se dá pelas seguintes correspondências (DOLEV; DWORK; STOCKMEYER, 1987):

- Para propôr um valor, um processo difunde sua proposta.
- Para decidir um valor, um processo escolhe a primeira mensagem recebida pela difusão atômica. Como a difusão atômica garante que a primeira mensagem recebida será a mesma em todos os processos, haverá consenso entre estes.

Por outro lado, conforme demonstrado em Chandra e Toueg (1996), o problema de Difusão Atômica pode ser reduzido ao problema de Consenso.

2.3 DETECTORES DE FALHAS

Um detector de falhas usualmente corresponde a uma entidade presente em cada réplica do sistema responsável por identificar processos faltosos no sistema. Essa identificação permite ao sistema contornar a impossibilidade de resolver o problema do consenso com pelo menos uma réplica faltosa em sistemas assíncronos (CHANDRA; TOUEG, 1996).

Desta forma, Chandra e Toueg (1996) propuseram uma abordagem de tolerância a falhas de *crash* com a utilização de detectores de falhas não confiáveis. São estes definidos em função de suas propriedades de completude (*completeness*) e exatidão (*accuracy*):

- Completude:

  - *Strong completeness*: em algum momento, cada processo faltoso será suspeito por todos os processos corretos.
  - *Weak completeness*: em algum momento, cada processo faltoso será suspeito por algum processo correto.

- Exatidão:
  - *Strong accuracy*: nenhum processo é suspeito antes de ocorrer uma falha de *crash*;
  - *Weak accuracy*: algum processo correto nunca será suspeito.
  - *Eventual strong accuracy*: existe um momento após o qual qualquer processo correto passa a não ser suspeito por nenhum processo correto.
  - *Eventual weak accuracy*: existe um momento após o qual algum processo correto passa a não ser suspeito por nenhum processo correto.

2.3.1 Detectores de Falhas de Mudez

Sendo falhas de *crash* o modelo mais simples de falhas que pode acontecer em um sistema distribuído, algumas abordagens foram criadas para ampliar a faixa de atuação dos detectores de falhas, mantendo suas vantagens. As falhas de mudez são um submodelo de falhas bizantinas mais amplo que as falhas de crash (DOUDOU et al., 1999).

Detectores de falhas de mudez preservam a modularidade dos detectores de falhas definindo um modelo de interação entre o algoritmo do serviço e o detector de falhas. Desta forma, os algoritmos previstos para serem utilizados em conjunto com os detectores de mudez devem enviar periodicamente alguma mensagem aos outros processos. Não são cobertas pelos detectores as falhas arbitrária, em que um atacante envia uma mensagem incorreta, dentro do conjunto de mensagens possíveis para a aplicação. Não são cobertos também algoritmos em que a mudez pode ser interpretada como um estado válido do sistema. Este assunto será apresentado com mais detalhes na seção 3.3.3.

2.4 TECNOLOGIAS DE VIRTUALIZAÇÃO

Os conceitos de virtualização estão presentes na literatura desde a década de 60, porém nos últimos vinte anos houve uma popularização

com a chegadas dos sistemas de virtualização para a plataforma x86 (NANDA; CHIUEH, 2005), e mais recentemente sua aplicação vem sendo bastante associada com a computação nas nuvens. Seu principal objetivo está na execução de diferentes sistemas operacionais em um mesmo *hardware*.

Dentre as muitas razões pelas quais sistemas virtualizados são utilizados, se destacam (NANDA; CHIUEH, 2005):

- **Consolidação de servidores:** melhor aproveitamento de serviços que subutilizam servidores consolidando em um menor número de máquinas físicas.
- **Isolamento:** máquinas virtuais proporcionam um isolamento entre sistemas executados em uma mesma máquina física (*sandbox*), e este isolamento auxilia no desenvolvimento de soluções de segurança.
- ***Hardware* virtual:** possibilita a utilização simulada de dispositivos inexistentes no sistema.
- **Múltiplos sistemas operacionais:** permite uma diversidade de sistemas operacionais em uma mesma máquina física, algo que também é bastante utilizado em soluções de segurança de funcionamento.

2.4.1 Conceitos Gerais

No que se refere à máquinas virtuais, três conceitos são bastante importantes por serem amplamente utilizados (NANDA; CHIUEH, 2005):

- **Máquina virtual (*Virtual Machine*)**: cada instância de um sistema operacional sendo executada em uma mesma máquina física é chamada máquina virtual, ou sistema convidado (*guest*).
- **Monitor de máquinas virtuais (*Virtual Machine Monitor*)**: é uma camada de software entre o hardware e as máquinas virtuais, gerenciando e dando suporte à execução das mesmas.
- **Sistema hospedeiro (*host*)**: é o sistema físico sobre o qual está sendo executado o Monitor de Máquinas Virtuais.

Quatro atributos principais são compartilhadas por máquinas virtuais modernas (ROSENBLUM, 2004):

- **Compatibilidade de Software**: as máquinas virtuais proveêm uma camada de abstração para que todo software escrito para o sistema virtualizado funcione corretamente.

- **Isolamento**: uma abstração de máquinas virtuais devem fornecer um isolamento de forma que qualquer programa rodando na máquina virtual não possua nenhum acesso à outras máquinas virtuais ou à máquina física, evitando a proliferação de *bugs* ou o acesso de intrusos.

- **Encapsulamento**: como toda a execução de um *software* rodando em uma máquina virtual está encapsulado nesta máquina virtual, pode ser fornecido um melhor gerenciamento dos recursos para o *software.*

- **Desempenho**: como adicionar uma camada intermediária entre o *software* e o *hardware* vai causar um decréscimo no desempenho, a VMM mapeia o os dispositivos virtuais diretamente para os dispositivos reais, diminuíndo o impacto.

2.4.2 Tipos de Virtualização

Várias formas de classificação para tipos de virtualização foram propostas e, dentre estas, algumas categorias se destacam (SAHOO; MOHAPATRA; LATH, 2010):

- **Virtualização Total**: nesta abordagem, a VM é executada pelo sistema operacional do host, geralmente no espaço de aplicação. Com isso, todo o acesso ao *hardware* é simulado e as máquinas virtuais não tem acesso direto ao sistema hospedeiro. Isso possibilita uma maior facilidade no uso mas também prejudica o desempenho da máquina virtual.

- **Virtualização de Sistema Operacional**: nesta abordagem, a VMM virtualiza o núcleo do sistema operacional, compartilhando com as máquinas virtuais. Todas as máquinas virtuais utilizam o mesmo núcleo do sistema hospedeiro, portanto é impossível a virtualização de diferentes sistemas operacionais nesta abordagem mas seu desempenho é superior do que um sistema totalmente virtualizado.

- **Paravirtualização**: na paravirtualização, os sistemas convidados são sistemas operacionais modificados especialmente para ro-

dar em ambientes virtualizados. Assim, as camadas entre a VMM e o *hardware* são diminuídas e as réplicas paravirtualizadas possuem um desempenho próximo a ambientes não virtualizados. isso gera também uma contrapartida em que as máquinas virtuais tem ciência de que são ambientes virtualizados.

- **Virtualização de Aplicação**: neste ambiente é possível executar uma aplicação virtualizada diretamente sem a necessidade de um sistema operacional presente na máquina virtual. Apenas os recursos necessários para a aplicação são virtualizados e disponibilizados para a aplicação.

2.5 CONSIDERAÇÕES FINAIS

Neste capítulo, foram apresentados conceitos básicos presentes na literatura que foram bastante empregados no decorrer do trabalho. Alguns desses conceitos são revisitados ou mais detalhados no decorrer do trabalho conforme necessário. Na Seção 2.1, foram apresentados os conceitos mais básicos relacionados com sistemas distribuídos, com foco nas características necessárias para a tolerância a faltas.

Na Seção 2.2 foram detalhados conceitos específicos voltados para a tolerância a faltas, especialmente no que se refere a faltas bizantinas que são o foco deste trabalho. Também foi apresentado nesta seção, um detalhamento da abordagem de replicação Máquina de Estados que serviu como base para este trabalho. Na seção seguinte, algumas considerações sobre detectores de falhas foram feitas de forma a contextualizar melhor o leitor. Apesar deste trabalho não utilizar diretamente detectores de falhas, os conceitos utilizados serviram de inspiração para o trabalho proposto.

Uma consolidação de definições e conceitos referentes a técnicas de virtualização é apresentada na Seção 2.4, sendo que são apresentadas categorizações e premissas de ambientes virtualizados.

Esta revisão bibliográfica será bastante explorada nos capítulos a seguir e estes conceitos serviram como base para a definição da proposta presente neste trabalho.

3 TRABALHOS RELACIONADOS

De forma a manter serviços funcionando corretamente mesmo sob a ocorrência de faltas, diversas abordagens surgiram. Apesar de as primeiras abordagens terem surgido há bastante tempo, o custo destas é muitas vezes impeditivo. Assim, novas abordagens surgiram com o objetivo de cumprir o mesmo objetivo que as anteriores com um menor custo.

Serão apresentadas neste capítulo, algumas das abordagens que mais influenciaram este trabalho. Inicialmente serão mostrados trabalhos BFT que se utilizam de técnicas de Replicação Máquina de Estados. Nas seções subsequentes são descritos trabalhos agrupados por características específicas como a utilização de detectores de falhas e técnicas de virtualização.

3.1 ABORDAGENS TRADICIONAIS

3.1.1 Practical Byzantine fault tolerance - PBFT

Proposta com o objetivo de fornecer uma abordagem prática na implementação de sistemas tolerantes a faltas bizantinas, o PBFT serve de suporte a uma replicação de máquina de estados. O algoritmo oferece a resiliência de $f = \lfloor \frac{n-1}{3} \rfloor$, sendo n o número de servidores replicados. Mesmo possuindo uma resiliência ótima para sistemas assíncronos, o algoritmo oferece ainda vantagens de desempenho por não precisar utilizar criptografia assimétrica para cifrar todas as mensagens enviadas (CASTRO; LISKOV, 1999a)(CASTRO; LISKOV, 1999b).

O fluxo normal do algoritmo é apresentado na Figura 2, que está separada em cinco quadrantes conforme os passos de execução. Estes passos são descritos a seguir:

1. O cliente envia para o servidor primário a requisição (quadrante a).

2. O servidor primário envia, por *multicast*, uma mensagem *"PRE-PREPARE"* a todas as réplicas (quadrante b).

3. Cada réplica envia, por *multicast*, uma mensagem *"PREPARE"* às outras réplicas do conjunto (quadrante c).

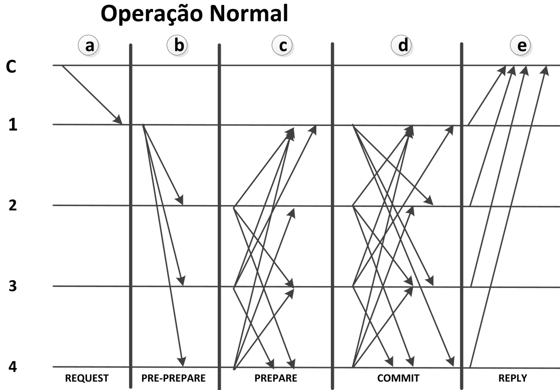

Figura 2: Execução normal do algoritmo PBFT.

4. Ao verificar que recebeu ao menos $2f$ *prepares*, a réplica envia uma mensagem *"COMMIT"* às outras réplicas (quadrante d).

5. Possuindo pelo menos $2f+1$ mensagens *"COMMIT"* para a requisição, esta é executada e o resultado enviado diretamente ao cliente (quadrante e).

6. Cliente aguarda até que $f+1$ respostas válidas de diferentes réplicas tenham sido recebidas para que o resultado seja aceito (quadrante e).

A partir de uma mensagem $\langle$*REQUEST, o,t,c*$\rangle$, o cliente *c* solicita a execução de uma operação *o* com uma assinatura de tempo (*timestamp*) para garantir a unicidade das mensagens. Se a resposta não for recebida de um número satisfatório de réplicas, a requisição é reenviada a todas as réplicas do conjunto que, reenviam a resposta caso esta já tenha sido processada ou, podem iniciar o algoritmo de troca de visão caso haja a suspeita de que a réplica primaria não repassou a requisição aos secundários.

Ao receber uma mensagem *"REQUEST"*, a réplica primária inicia a fase *pre-prepare* enviando a todas as réplicas a mensagem $\langle\langle$*PRE-PREPARE, v,n,d*$\rangle$*, m*$\rangle$, sendo *v* o número da visão atual, *n* o número de sequência atribuído a mensagem pela réplica primária, *m* a requisição enviada pelo cliente, e *d* um sumário *(digest)* calculado sobre a mensagem m. A utilização da mensagem *"PRE-PREPARE"* está relacionada diretamente com a ordenação das requisições, centralizando o

sequenciamento das mensagens em uma réplica primária. Este passo garante que uma mesma mensagem não tenha diferentes números de sequência perante o sistema por duas réplicas terem aceitado a mesma mensagem.

A mensagem $\langle$*PREPARE, v,n,d,i*$\rangle$ contendo o número v da visão atual, o número de sequência da requisição n, o sumário d da requisição e o número i da réplica, é enviada via *multicast* a todas as réplicas assim que a mensagem *"PRE-PREPARE"* é recebida e inicia a fase *prepare* do algoritmo. Esta mensagem é aceita pela réplica apenas se a sua assinatura for válida e o número da visão corresponde à visão atual. O predicado *prepared(m, v, n, i)* torna-se verdadeiro caso a réplica i tenha armazenado em seu log: uma mensagem *"PRE-PREPARE"* com o mesmo *m, v e n*, e ao menos $2f$ mensagens *"PREPARE"* correspondentes.

A validação deste predicado garante a concordância entre as réplicas na ordenação da mensagem, além de garantir que, se *prepared(m, v, n, i)* é verdadeiro então *prepared(m', v, n, i)* é falso para qualquer mensagem *m*. Assim, a utilização deste predicado garante que que diferentes requisições sejam executadas com o mesmo número de sequência.

Caso a fase *prepare* fosse removida do sistema, a fase *commit* entraria diretamente após a fase *pre-prepared* e permitiria, por exemplo, que não fosse atingido um consenso no sequenciamento das requisições entre as réplicas quando um atacante, no controle da réplica primária, atribuísse números de sequência diferentes para mensagens iguais ou números de sequência iguais para mensagens diferentes.

A fase *commit* se inicia assim que o predicado *prepared(m, v, n, i)* se torna verdadeiro e a réplica i envia por *multicast* a mensagem $\langle$*COMMIT, v,n,D(m),i*$\rangle$. Nesta fase, são definidos dois predicados: *commited(m, v, n)* é considerado verdadeiro se *prepared(m, v, n, i)* for verdadeiro para todas as réplicas i em um conjunto de $f+1$ réplicas corretas; *commited-local(m, v, n, i)* é considerado verdadeiro se *prepared(m, v, n, i)* e a réplica i possuir pelo menos $2f+1$ *commits* correspondentes a mensagem m.

O predicado *commited-local(m, v, n, i)* garante que requisições executadas localmente na réplica correta i serão executadas também em pelo menos $f+1$ réplicas corretas, respeitando a sequência acordada entre as réplicas. Após o processamento da mensagem, a resposta é enviada diretamente ao cliente e a réplica passa a não aceitar nenhuma requisição com *timestamp* menor que o da última requisição enviada.

Se a fase *commit* fosse removida do algoritmo, a requisição se-

ria executada logo após a confirmação do predicado *prepared(m, v, n, i)*, porém, não haveria a garantia de que este predicado tornou-se verdadeiro em pelo menos $2f + 1$ réplicas.

Ao enviar uma requisição, o cliente inicia um temporizador e, caso não receba o *quórum* de mensagens dentro de um determinado tempo, reenvia a requisição por *multicast* a todas as réplicas. Estas, por sua vez, reenviam a resposta ao cliente caso já tenha sido processada, ou repassam a requisição à réplica primária caso contrário. Se a réplica primária não repassar ao grupo a requisição após algum tempo, as réplicas inciarão o procedimento de troca de visão para substituir a réplica primária. Este procedimento é necessário não apenas para tolerar faltas de *crash* mas também para tolerar casos em que um atacante no controle da réplica primária tente postergar infinitamente o processamento de requisições.

De forma a não armazenar indefinidamente as mensagens processadas no *log*, cada réplica i deve, periodicamente, enviar às outras por *multicast* uma mensagem ⟨*CHECKPOINT, n,d,i*⟩ contendo o número n da última requisição processada e um sumário d do estado atual da réplica. Ao receber $2f + 1$ *checkpoints* para um mesmo número de sequência n, a réplica descarta todas as mensagens *"PRE-PREPARE"*, *"PREPARE"* e *"COMMIT"* que possuam números de sequência menor que n. *Checkpoints* antigos também podem ser descartados já que este representa o último estado estável do sistema.

Qualquer réplica ao receber uma requisição inicia um temporizador que é cancelado assim que a requisição termina. Caso a requisição não seja finalizada dentro deste tempo, a réplica inicia a troca de visão para $v + 1$. Para tanto, a réplica i para de receber mensagens relacionadas a requisições e envia por *multicast* a mensagem ⟨*VIEW-CHANGE, v+1,n,C,P,i*⟩, sendo $v + 1$ a visão proposta, n o número de sequência do último checkpoint recebido, C um conjunto com $2f + 1$ mensagens do último *checkpoint* recebido e P um conjunto com as mensagens com número de sequência maior do que n e com *prepared(m, v, n, i)* igual a verdadeiro.

Quando a réplica primária da visão $v + 1$ recebe *2f* mensagens de troca de visão para $v + 1$, a réplica primária envia por *multicast* a mensagem ⟨*NEW-VIEW, v+1, V, O*⟩, sendo V o conjunto das mensagens de troca de visão recebidas e O o conjunto de mensagens que devem ser revistas pelas réplicas após a troca de visão.

3.1.2 Separating Agreement from Execution for Byzantine Fault Tolerant Services

O objetivo deste algoritmo é garantir integridade, disponibilidade e confidencialidade, aumentando assim a confiança na execução de requisições mesmo com a ocorrência de faltas maliciosas no sistema. A técnica utilizada se baseia no aprimoramento do algoritmo BFT de forma a diminuir o seu custo e garantir confidencialidade, que não é obtida em técnicas tradicionais (YIN et al., 2003).

Usualmente, o consenso e a execução das tarefas são efetuados pelo mesmo conjunto de réplicas, entretanto, a separação destas etapas traz duas vantagens ao sistema. Enquanto técnicas tradicionais necessitam de $3f+1$ réplicas para tolerar f faltas, o algoritmo proposto necessita para a execução das requisições apenas uma maioria simples, necessitando $3f+1$ réplicas apenas para a ordenação das mensagens.

A diferenciação entre réplicas participantes da ordenação e réplicas participantes da execução é relevante pois as réplicas que executarão as requisições tendem a ser muito mais complexas que as réplicas participantes do consenso já que estas podem, além de utilizar um *hardware* mais simples e barato, utilizar uma biblioteca genérica com custo menor do que o desenvolvimento em N-versões.

Outra vantagem desta abordagem é a possibilidade de utilização de um *firewall*. Isso é possível pois neste modelo um conjunto de nós *firewall* restringem a comunicação das réplicas de execução, filtrando as respostas incorretas antes que estas cheguem ao cliente. Apesar de esta arquitetura inserir uma maior carga na comunicação, em testes de desempenho a abordagem se mostrou próxima de abordagens tradicionais.

Este algoritmo foi elaborado tendo em vista um sistema distribuído assíncrono em que não há a garantia de recebimento das mensagens, garantindo segurança apesar do *timing*, falhas de *crash*, omissão de mensagens, entrega de mensagens fora de ordem e alteração de mensagens.

Este modelo também assume que as mensagens enviadas entre os participantes são criptografadas e as chaves privadas conhecidas apenas por seus proprietários. Assume-se também que as requisições somente serão aceitas de um universo finito de clientes autorizados. O sistema replicado deve agir como uma máquina de estados determinística, mantendo a consistência entre as réplicas.

Para iniciar a comunicação, o cliente envia à réplica primária uma requisição $\langle REQUEST,\ o,\ t,\ c\rangle$ sendo que o indica a operação a

ser executada, t indica o *timestamp* e c indica o cliente solicitante da mensagem. Após este envio, o cliente aguarda uma resposta do cliente no formato $\langle REPLY\ v,\ n,\ t,\ c,\ \varepsilon,\ r\ \rangle$ sendo v correspondente ao número da visão no *cluster* de consenso quando a requisição foi ordenada, n o número de sequência atribuído na ordenação, t indicando o *timestamp*, c indicando o cliente solicitante, ε indicando o conjunto de réplicas que chegaram na resposta r. Caso o cliente não receba a resposta dentro de um *timeout*, este reenvia por *multicast* a requisição a todas as réplicas do *cluster* de consenso.

Ao receber uma requisição do cliente, o *cluster* de consenso procede da seguinte maneira:

1. Verifica no *cache* local se esta mensagem já foi executada. Em caso positivo retorna ao cliente a resposta já processada.

2. Atribui um número de sequência e gera um certificado de acordo a ser enviado ao *cluster* de execução no formato $\langle COMMIT\ v,\ n,\ d,\ A\ \rangle$ sendo que v indica a visão atual do *cluster* de consenso, n contém o número de sequência atribuído, d contém um sumário calculado sobre a requisição e A corresponde ao conjunto de servidores que assinaram este certificado.

3. Ao receber a resposta das réplicas, esta é repassada ao cliente e, possivelmente, armazenada em um *cache* local.

O *cluster* de consenso é responsável também por reenviar a requisição ao *cluster* de execução caso não receba a resposta dentro de um tempo pré-determinado, já que a comunicação entre o *cluster* de consenso e o *cluster* de execução também não é garantidamente confiável.

O *cluster* de execução, por sua vez, ao receber uma requisição e um certificado de acordo segue o seguinte protocolo:

1. Se o *timestamp* da requisição recebida é maior do que a da última requisição executada, o servidor executa a requisição e retorna uma resposta ao *cluster* de consenso.

2. Se o *timestamp* da requisição recebida for igual a da última requisição executada, o *cluster* gera e envia uma nova resposta com base nos dados em *cache*.

3. Se o *timestamp* da requisição for recebida for menor do que a da última requisição executada, o *cluster* segue o mesmo procedimento do segundo caso porém associa o novo número de sequência à última requisição processada.

Na implementação dos autores, o *cluster* de consenso utiliza a biblioteca *BASE* que é responsável pela execução do algoritmo de *commit* em três fases, atribuição de número de sequência, troca de visão e *checkpoints*. Esta biblioteca, entretanto, não é utilizada em sua totalidade pois a execução das requisições é feita separadamente. Para isto, ao invés de executar a requisição e retornar ao cliente, a biblioteca repassa a requisição para uma fila de mensagens.

Esta fila de mensagens está presente em cada réplica participante do *cluster* de consenso e armazena: o maior número de sequência recebido, as requisições pendentes, uma lista com os certificados de requisição e uma lista com os certificados de acordo. Esta fila trata de enviar, por *multicast*, as requisições recebidas às réplicas do *cluster* de execução e aguarda uma resposta a ser retornada ao cliente.

Cada réplica participante do *cluster* de execução mantém o estado atual da aplicação, uma lista com as requisições pendentes a serem executadas, o maior número de sequência executado, e uma lista com as requisições executadas mais recentemente.

Quando uma réplica do servidor de execução recebe uma requisição válida certificada pelo cliente e com um certificado de acordo, esta é armazenada na fila de requisições pendentes para ser executada assim que as requisições com menor número de sequência forem executadas. Caso o *timestamp* da requisição for maior que a da última requisição processada, a requisição é executada e seu resultado retornado ao *cluster* de consenso. Caso contrário, o sistema inicia o procedimento para retransmissão.

Para garantir a confidencialidade das informações, esta abordagem introduz o *privacy firewall* que consiste basicamente em nós de filtros adicionados entre nós de execução e de consenso, transmitindo apenas informações enviadas por servidores de execução corretos. Os filtros devem ser dispostos em h + 1 linhas de h + 1 filtros tendo na primeira linha cada filtro acoplado com um servidor de consenso. Cada filtro tem apenas uma conexão física com cada um dos filtros na linha acima ou abaixo dele. Corpos de requisição enviadas pelo cliente são cifrados de forma que apenas os servidores de execução podem visualizar seu conteúdo.

Ao receber uma requisição e um certificado de acordo do *cluster* de consenso, a linha do filtro mais abaixo identifica se tem a resposta para a requisição em *cache* e, em caso positivo, retorna a mesma ao *cluster* de consenso. Caso contrário, o filtro envia, por *multicast*, as requisições aos filtro na linha acima. Na linha de filtros mais próxima aos servidores de execução, cada filtro envia as mensagens apenas a um

servidor.

Ao enviar uma resposta, o servidor deve utilizar um esquema de criptografia de limiar para que cada servidor envie apenas parte da resposta aos filtros do nível superior. Quando filtros de uma linha conseguem remontar a mensagem a partir de suas assinaturas, estes reenviam a mensagem à camada inferior de filtros.

A utilização de h + 1 linhas com h + 1 filtros garante, desde que não haja mais do que h falhas, que exista ao menos um caminho correto entre os servidores de execução e consenso, e que existe ao menos uma linha contendo apenas filtros corretos.

3.2 ABORDAGENS HÍBRIDAS

3.2.1 BFT-TO: Intrusion Tolerance with Less Replicas

A arquitetura BFT-TO foi proposta com o intuito de oferecer um algoritmo tolerante a faltas bizantinas similar ao PBFT, reduzindo o número de réplicas no PBFT de $3f + 1$ para $2f + 1$ réplicas. Essa redução é fornecida a partir da utilização de um componente confiável, representando um modelo híbrido em que o sistema replicado (e não confiável) é incrementado com a utilização de um componente mais simples, considerado confiável. Este componente confiável é distribuído e fornece um serviço de ordenação para as réplicas (CORREIA; NEVES; VERÍSSIMO, 2013; CORREIA; NEVES; VERISSIMO, 2004).

O BFT-TO utiliza um sistema de comunicação assíncrono com um número finito de clientes. O componente confiável, chamado *TO wormhole*, funciona de forma parcialmente síncrona, e é baseado nas seguintes premissas:

- O componente *TO wormhole* é à prova de falhas, sendo que a integridade do serviço e a confidenciabilidade das chaves nele armazenadas se mantém mesmo na presença de uma intrusão.

- O componente *TO wormhole* tem síncronia suficiente para resolver o consenso.

É uma presmissa do protocolo que cada par cliente-servidor ou servidor-servidor está conectado por uma rede FIFO que autentica as mensagens trocadas e garante a integridade das mensagens. Essas garantias podem ser obtidas na prática por retransmissão de mensagens e a utilização de criptografia para autenticação das mensagens.

O componente confiável *TO wormhole* auxilia a execução do procolo de *atomic multicast* indicando duas informações sobre cada mensagem m: (a) quando a mensagem m pode ser entregue; e (b) em que ordem a mensagem m deve ser entregue. Cada réplica sempre avisa o *TO wormhole* quando envia ou recebe uma mensagem m. Assim, uma mensagem só é entregue quando o componente confiável detecta que uma mensagem enviada por um servidor foi recebida por f outros servidores.

O protocolo BFT-TO se inicia a partir de uma requisição enviada a partir de um cliente para um dos servidores. Essa requisição tem o formato ⟨*REQUEST, src, dst, num, cmd, vec*⟩, sendo que *src* indica o cliente que enviou a mensagem, *dst* indica o servidor de destino, *num* representa o número da requisição para que o cliente relacione a resposta com a requisição, *cmd* indica o comando a ser executado pelo serviço replicado, e *vec* é a assinatura da mensagem.

Assim que um servidor recebe uma mensagem *"REQUEST"*, esta mensagem é armazenada, a menos que já tenha sido executada anteriormente. Ao mesmo tempo, uma segunda tarefa envia estas mensagens armazenadas, por *atomic multicast*, para as outras réplicas. Uma terceira tarefa é executada nos servidores conforme as mensagens vão sendo entregues. Assim, quando a requisição é entregue pelo *atomic multicast*, o comando contido nesta requisição é executado e uma resposta no formato ⟨*REPLY, src, dst, num, res*⟩ é enviada ao cliente que solicitou a execução. Nesta mensagem, *src* corresponde ao servidor que executou a requisição, *dst* indica o cliente que fez a requisição, *num* corresponde ao número da requisição, e *res* é o resultado da requisição. O cliente espera até que receba $f+1$ mensagens idênticas dos servidores para aceitar a resposta de uma requisição.

O algoritmo de *atomic multicast* executado pelos servidores, possui apenas um tipo de mensagem ⟨*ACAST, src, dst,* S_{req}*, msg_id*⟩ sendo que *src* é o endereço do servidor remetente, *dst* é um conjunto com os endereços dos servidores de destino, S_{req} é um conjunto com as requisições, e *msg_id* um identificador único da mensagem dentro do servidor remetente.

O protocolo de *atomic multicast* encapsulando as requisições em uma mensagem *"ACAST"*. Depois disso é invocada a função *TOW_sent*, indicando para o *TO wormhole* que uma mensagem está sendo enviada e coloca a mensagem em uma lista aguardando que servidores suficientes executem *TOW_received* e o componente *TO wormhole* execute a ação *TOW_decide*.

Uma segunda tarefa, executada quando uma mensagem *"ACAST"*

é recebida, adiciona a mensagem em uma lista de mensagens recebidas. E uma terceira tarefa apenas pega as mensagens armazenadas na segunda tarefa e executa a ação *TOW_received*.

Uma quarta tarefa, executada quando o componente confiável executa a ação *TOW_decide*, armazena os dados da decisão em uma lista local. Uma quinta tarefa, executada quando uma mensagem está aguardando para ser entregue e possui uma ação *TOW_decide* correspondente, entrega a mensagem e reenvia para todos os servidores que não participaram da decisão.

Com isso, Correia, Neves e Veríssimo (2013) foi capaz de contornar a impossibilidade FLP e reduzir o número de réplicas necessárias de $3f + 1$ para $2f + 1$, com a utilização de um elemento confiável que fornece auxílio ao algoritmo de *atomic broadcast*.

3.2.2 Efficient Byzantine Fault Tolerance - MinBFT e Min Zyzzyva

Em (VERONESE et al., 2013), duas abordagens foram propostas com o objetivo de reduzir o custo de replicação para tolerância de faltas bizantinas. Enquanto ambos os algoritmos propostos obtém esta melhor eficiência a partir do uso de um componente simples confiável, o MinBFT oferece uma abordagem mais próxima ao PBFT e o Min Zyzzyva acrescenta técnicas especulativas para reduzir - além do número de réplicas - a complexidade do algoritmo.

Outra abordagem citada anteriormente, o BFT-TO (CORREIA; NEVES; VERISSIMO, 2004), também oferece uma redução no número de réplicas de $3f + 1$ para $2f + 1$ utilizando um componente confiável mas apesar de ter mostrado a possibilidade dessa redução, requer um componente confiável consideravelmente complexo. Os algoritmos propostos por Veronese et al. (2013) utilizam um componente confiável com uma simplicidade considerável. Este commponente confiável, chamado USIG, fornece apenas um contador para as operações executadas em uma réplica, e um verificador para a validade deste contador.

O componente confiável USIG é executado em cada uma das réplicas e atribui um identificador incremental para mensagens. Este identificador deve ser único, monotônico e sequencial. Estas propriedades devem ser garantidas mesmo no caso de a réplica estar comprometida por uma intrusão. Para tanto, é fornecida uma interface com dois métodos:

- *createUI(m)*: retorna um certificado que atribui à mensagem m

um identificador, gerado pelo componente confiável.

- *verifyUI(PK, UI, m)*: verifica que o certificado UI é válido para a mensagem *m*.

No algoritmo MinBFT, as réplicas se intercalam em uma sucessão de configurações chamadas visão que define a réplica primária. Os passos são semelhantes ao PBFT e o algoritmo se inicia a partir do cliente enviando para a réplica primária uma requisição no formato $\langle REQUEST,\ c,\ seq,\ op\rangle_\sigma$, sendo que c corresponde ao cliente que enviou a requisição, *seq* o número de sequência da requisição no cliente e *op* a operação requisitada.

Ao receber a requisição, a réplica primária reenvia para todas as outras réplica no formato de uma mensagem $\langle PREPARE,\ v,\ s_i,\ m,\ UI_i\ \rangle_\sigma$, sendo que v é o número da visão atual, s_i é a éplica primária que está enviando a mensagem, m é a requisição do cliente e UI_i é o identificador da mensagem "PREPARE" gerado pelo USIG.

Assim que cada réplica *backup* s_j recebe a mensagem "PREPARE" da réplica primária s_i, é difundida uma mensagem $\langle COMMIT,\ v,\ s_j,\ s_i,\ m,\ UI_i,\ UI_j\ \rangle_\sigma$. Nesta mensagem, UI_j corresponde ao identificador da mensagem "COMMIT" gerado pelo componente confiável. A cada mensagem recebida a réplica verifica se o identificador atribuído é válido, a partir do seu próprio componente confiável.

Uma réplica correta só irá difundir a mensagem "COMMIT" assim que três condições sejam cumpridas: (1) v é a visão atual e o remetente do "PREPARE" é a réplica primárias desta visão, (2) a requisição possui uma assinatura válida do cliente e, (3) já foi aceita uma mensagem m' em que $cv' = cv - 1$, sendo cv' o contador da requisição m' e cv o contador da requisição m. A terceira condição faz com que as mensagens não apenas sejam executadas na ordem definida, mas também que elas sejam recebidas sempre na ordem correta.

Após executada a requisição pela réplica s, é gerada uma mensagem $\langle REPLY,\ s,\ seq,\ res\rangle_\sigma$, sendo *seq* o da requisição solicitada pelo cliente e *res* a resposta após a execução da requisição. Esta resposta é aceita assim que $f + 1$ respostas idênticas forem recebidas de réplicas distintas.

O algoritmo Min Zyzzyva é bastante semelhante ao MinBFT, porém remove a etapa de "COMMIT" e envia diretamente a resposta ao cliente assim que a mensagem "PREPARE" for recebida e ordenada. Nesta caso, a mensagem só é diretamente aceita caso $2f + 1$ respostas idênticas forem recebidas. Caso uma uma quantidade de respostas entre $f + 1$ e $2f$ seja recebida, o cliente uma mensagem com um certificado

de *commit* para todas as réplicas até que receba uma confirmação de pelo menos $f+1$ réplicas.

3.2.3 Zyzzyva: speculative Byzantine fault tolerance

Apesar de algoritmos de tolerância a faltas bizantinas serem promissores na construção de serviços confiáveis, sua utilização na prática ainda é pouco difundida devido a sua, ao menos teórica, sobrecarga na comunicação. O objetivo do Zyzzyva é oferecer um serviço replicado, tolerante a faltas bizantinas e com alto desempenho, utilizando para isso uma abordagem especulativa (KOTLA et al., 2008).

Similar a uma abordagem tradicional, o algoritmo Zyzzyva define que um cliente envia uma requisição a uma réplica primária que ordena as requisições repassando-as às outras réplicas. Estas, por sua vez, ao invés de executarem o algoritmo de consenso para garantir a corretude da execução, executam a requisição e imediatamente e enviam o resultado, especulativamente, ao cliente. Este, portanto, verifica se há um consenso nas respostas recebidas e exige uma garantia de consenso entre as réplicas apenas se não houve unanimidade entre as respostas recebidas, permitindo maior agilidade no processamento das requisições quando não há faltas. Assume-se que as mensagens enviadas terão sua identidade comprovada por técnicas de criptografia assimétrica e um atacante não tem poder computacional suficiente para quebrar o algoritmo utilizado para tal.

Zyzzyva utiliza um protocolo de replicação de máquina de estados executado por $3f+1$ réplicas. Como outros protocolos BFT, Zyzzyva define um procedimento de consenso, um procedimento para troca de visão e um para *checkpoint*.

O caso básico de funcionamento sem faltas ocorre como segue:

1. O cliente envia uma mensagem *"REQUEST"* para o servidor primário solicitando a requisição de uma execução;

2. O servidor primário envia uma mensagem *"ORDER-REQ"* a todas as réplicas;

3. Cada réplica processa a requisição localmente e envia a mensagem *"SPEC-RESPONSE"* ao cliente;

4. O cliente recebe $3f+1$ respostas idênticas e completa a requisição.

Para solicitar uma requisição o cliente envia à réplica primária uma mensagem <*REQUEST, o, t, c*> tendo *o* indicando a operação

a ser executada, t indicando a assinatura temporal da requisição e c apontando para o cliente que fez a solicitação. Um *timer* é criado pelo cliente para evitar a postergação infinita e será cancelado ao completar a requisição. Caso o cliente não receba um *quórum* de mensagens antes da finalização do *timer*, devido a falha no primário, este inicia o procedimento para troca de visão. A ausência deste procedimento poderia fazer com que um atacante, em posse da réplica primária, retardasse infinitamente o processamento das requisições.

Ao receber a requisição do cliente, a réplica primária atribui um número de sequência à requisição e envia às outras réplicas a mensagem <<*ORDER-REQ, u, n,* h_n*, d, ND*>*, m*> contendo o número da visão u, o número de sequência n atribuído, um resumo criptográfico h_n do histórico, um resumo criptográfico da mensagem d, um conjunto *ND* de valores não determinísticos necessários para a execução da requisição e a própria requisição m. Caso essa mensagem não chegue às réplicas o cliente não receberá a resposta a tempo e efetuará o procedimento para troca de visão, elegendo uma nova réplica primária.

A réplica só pode executar requisições na ordem que elas são enviadas pelo cliente, porém atrasos na rede e falha no primário podem deixar buracos no envio das requisições. Portanto, antes de aceitar uma mensagem da réplica primária, esta deve verificar a sequência da requisição e solicitar à primária as faltantes, caso existam. Ao executar a requisição, a réplica responde ao cliente com uma mensagem <<*SPEC-RESPONSE, u, n,* h_n*, H(r), c, t*>*, i, r, OR*> sendo *H(r)* o resumo criptográfico da resposta, i o identificador da réplica, r a resposta da requisição e *OR* a mensagem *"ORDER-REQ"* recebida da réplica primária.

Ao receber $3f+1$ respostas idênticas de diferentes réplicas, o cliente assume que todas as réplicas são corretas e completa a requisição. Caso, ao acabar o tempo definido pelo *timer* criado no início da requisição, o cliente tenha recebido entre $2f+1$ e $3f$ respostas idênticas, o cliente tem noção que aquela resposta tem o consenso da maioria das réplicas porém as próprias réplicas não. Desta forma, o cliente anexa à resposta um certificado de *commit* e envia a todas as réplicas a mensagem <*COMMIT, c, CC*>, sendo *CC* correspondente ao certificado de *commit*, para que atualizem seu estado de acordo com a última requisição válida. As réplicas recebem este certificado e respondem com uma mensagem <*LOCAL-COMMIT, u, d, h, i, c*> demonstrando que estão cientes da execução da requisição. O cliente completa a execução da requisição somente após receber ao menos $2f+1$ mensagens *"LOCAL-COMMIT"*.

Caso este passo fosse suprimido do algoritmo, seria possível a um atacante em posse de f réplicas, incluindo a réplica primária, empregar um ataque da seguinte forma:

1. Atacante em posse da réplica primária envia uma requisição incorreta para um conjunto C de f réplicas corretas, e a requisição correta para outro conjunto *C'* com $f+1$ réplicas corretas;

2. Um conjunto F composto por f réplicas faltosas e $f+1$ réplicas corretas enviam o resultado para o cliente que aceita a resposta e completa a requisição;

3. Réplica primária faltosa provoca uma troca de visão;

4. Protocolo de troca de visão precisa determinar quais requisições foram executadas naquela visão utilizando para isso dados de $2f+1$ réplicas já que f réplicas podem estar incorretas. Supondo que a troca de visão utilize as informações das réplicas contidas em F, C e apenas uma réplica do conjunto *C'*, então haverá apenas uma réplica em seu estado correto fazendo parte da troca de visão e não é possível ter certeza de que o estado do sistema refletirá o estado correto.

Outra possibilidade está em o cliente não receber $2f+1$ respostas consistentes das réplicas antes de o *timer* expirar. Isto pode acontecer apenas caso haja:

- Uma falha na réplica primária.

- Uma falha na comunicação.

Caso o cliente receba ao menos duas respostas correspondente à mesma requisição, mas com números de sequência divergentes, estas constituem uma prova do comportamento incorreto da réplica primária. Neste caso, uma mensagem <*POM, u, POM*> é enviada a todas as réplicas que iniciam uma troca na visão do sistema.

Quando o número de respostas recebidas for menor que $2f+1$, o cliente assume que houve uma falha na comunicação e reenvia a requisição para todas as réplicas. Estas, por sua vez, retornam o resultado da requisição ao cliente caso este já tenha sido executado, ou repassam a requisição para a réplica primária reordenar a requisição a partir de uma mensagem <*CONFIRM-REQ, u, m, i*>. Se a réplica primária não responder esta requisição em um tempo pré-especificado, a réplica inicia o procedimento para troca de visão.

Caso esta etapa não fosse cumprida, um servidor primário faltoso poderia impedir o cliente de decidir sobre respostas corretas ordenando incorretamente as requisições recebidas.

3.3 ABORDAGENS BASEADAS EM DETECTORES DE FALHAS

3.3.1 Unreliable Intrusion Detection in Distributed Computations

Em sua proposta, Malkhi e Reiter (1997) defendem a utilização de detectores de falhas para ambiente em que estas podem acontecer causadas por intrusão (falhas bizantinas). Para tanto, tenta-se modelar uma falha bizantina como um comportamento incorreto do processo. O desafio maior, entretanto, está em como identificar quais comportamentos devem ser considerados como uma falha no processo.

A abordagem utilizada trata apenas de falhas que impedem o progresso do processo, delegando o tratamento de outros tipos de intrusão para níveis de consenso superiores. Desta forma, partindo do princípio que todas as mensagens enviadas por réplicas corretas são enviadas de forma confiável às outras réplicas corretas, é definido um detector de falhas bizantinas $\diamond S(bz)$. Este detector segue a especificação de que em algum momento todo processo que deixe de enviar mensagens será marcado como faltoso por todas as réplicas corretas *(Strong Completeness)* e, existe um tempo após o qual qualquer processo correto deixa de ser suspeito para todas as réplicas corretas *(Eventual Weak Accuracy)*.

Dado um grupo de processos $p0, \ldots, p_n - 1$ em um sistema assíncrono, esta abordagem assume que há no máximo $f = |\frac{n-1}{3}|$ processos faltosos. A primitiva de *broadcast* confiável garante que as mensagens serão sempre entregues de acordo com as seguintes propriedades:

- Integridade: se *bcast-receive(m, q)* é executado então q executou *bcast-send(m)*.
- Concordância: para duas réplicas corretas, se uma delas executa o recebimento de uma mensagem, a outra também executará.
- Validade: se p e q são correto, e p efetua um envio de m, então q efetua um recebimento de m;
- Ordenação na fonte: se p recebe duas mensagens de q, estas serão recebidas na mesma ordem em que foram enviadas.

- Ordem causal: Se p e q são corretas e p efetua o envio de *m'* após o recebimento de m então q efetua o recebimento de *m'* após o recebimento de m.

Possuindo cada réplica um detector de falhas onde ficam armazenadas quais as réplicas são suspeitas, é definido que réplicas faltosas são aquelas que apresentam apenas um número finito de recebimentos dentro de um tempo de execução infinito. Isso leva, porém, ao problema de que réplicas faltosas são geralmente silenciosas, dificultando sua detecção. Para isso são assumidas as premissas de *(Strong Completeness)* e *(Eventual Weak Accuracy)*.

Um dos protocolos de consenso propostos na abordagem se dá a partir da utilização do detector de falhas $\diamond$*S(bz)* em que cada réplica inicia com um valor binário e deve decidir sobre um dos valores sugerido por uma das réplicas, de forma a garantir as seguintes propriedades:

1. toda réplica correta decide um valor;
2. toda réplica correta decide um mesmo valor;
3. se todas as réplicas corretas mantém um mesmo valor no início do protocolo, então este é o valor do consenso.

O protocolo de consenso se resume basicamente em rodadas assíncronas coordenadas por um único líder. O processo se inicia com cada réplica fazendo *broadcast* do valor escolhido e do número da rodada. Quando o líder coleta $\frac{2n+1}{3}$ destas mensagens, seleciona o valor escolhido por, pelo menos, $\frac{n-1}{3}+1$ réplicas e envia a todos por *broadcast* como uma sugestão. Cada réplica responde às outras com uma concordância ou discordância, suspeitando do líder no último caso. Se um processo recebe $\frac{n-1}{3}+1$ concordâncias, altera seu valor local para o valor recebido e inicia a próxima rodada. Se $\frac{2n+1}{3}$ respostas positivas forem recebidas, a réplica decide sobre este valor e encerra o protocolo. No caso do recebimento de uma resposta mal formada, a réplica emissora passa imediatamente para a lista de suspeitos do receptor. Sua operação normal pode ser observada na Figura 3.

Também é proposto um protocolo de consenso híbrido em que, além da utilização do detector de falhas, são utilizadas técnicas de randomização. Neste contexto, o protocolo tem a finalização garantida se o detector de falhas satisfaz os requisitos de $\diamond$*S(bz)* ou se a propriedade *Strong Completeness* se verifique em certas situações.

Da mesma forma que o protocolo anterior, este opera em rodadas, tendo em cada uma um líder designado. Cada réplica inicia o

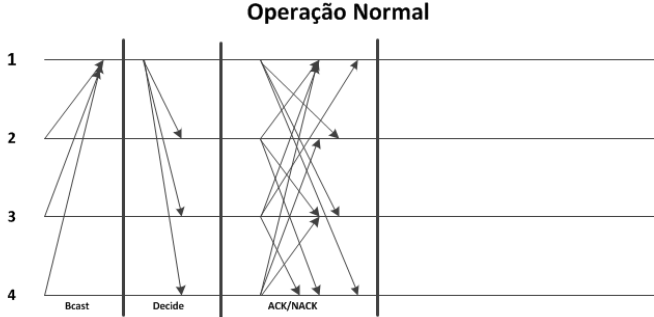

Figura 3: Operação normal proposta em Malkhi e Reiter (1997)

procolo a partir do envio, por *broadcast*, de um valor inicial e o número da rodada. Quando uma réplica coleta $\frac{2n+1}{3}$ mensagens, verifica se todas possuem valores v iguais e atualiza seu valor local v_i para v, caso contrário $v_i = \perp$. Se o valor local $v_j = \perp$ para o líder p_j, este atualiza v_j para um valor randômico, remetendo em ambos os casos este valor por *broadcast*. Cada processo p_i aguarda por um tempo definido a mensagem do líder, alterando seu valor local para o valor recebido, suspeitando do líder caso contrário e atualizando seu valor local para $\perp$. p_i envia então seu valor local para todas as réplicas e aguarda $\frac{2n+1}{3}$ mensagens. Dentre estas, haverá, ao menos, $\frac{n-1}{3}+1$ contendo um valor v que é atribuído a v_i por p_i, iniciando uma nova rodada. Este processo se repete até que cada participante tenha recebido $\frac{2n+1}{3}$, terminando assim o protocolo.

3.3.2 Solving Consensus in a Byzantine Environment Using an Unreliable Fault Detector

De forma a resolver o problema de consenso em sistemas distribuídos assíncronos sujeitos a faltas bizantinas, Kihlstrom, Moser e Melliar-Smith (1997) e Kihlstrom, Moser e Melliar-Smith (2003) definem duas classes de detectores de falha: $\diamond$*S(Byz)* corresponde a detectores de faltas eventualmente fortes no quais existe um momento após o qual todo processo que apresente uma falta bizantina passa a ser suspeito por todo processo correto, enquanto $\diamond$ *W(Byz)* corresponde aos detectores de faltas eventualmente fracos nos quais existe um momento após o qual qualquer processo correto deixa de ser suspeito por

outros processos. Assim, é apresentado um algoritmo que se utiliza de detectores de faltas de ambas as classes para resolver o problema de consenso em sistema com até $\frac{n-1}{3}$ faltas bizantinas. Diferentemente da abordagem utilizada por Malkhi e Reiter (1997) esta não se utiliza de recursos como um serviço *broadcast* confiável e ordenado, possibilitando um protocolo com um menor custo.

No modelo utilizado, o sistema é assíncrono sem nenhuma garantia sobre o tempo máximo de uma computação ou envio de uma mensagem. Cada processo tem acesso apenas ao seu relógio local, não sincronizado com outros processos. As mensagens são transmitidas através de primitivas *send* e *receive*. Processos corretos agem de acordo com a especificação, enquanto processos faltosos tem comportamento arbitrário. Cada processo possui um par de chaves RSA, garantindo a autenticidade das mensagens.

O algoritmo de consenso se utiliza de revezamento de coordenadores e trabalha em rodadas assíncronas, sabendo que durante a rodada r o número do coordenador será sempre dado pela fórmula $(r \ mod \ n) + 1$ sendo n o número de processos do sistema. Utiliza basicamente quatro tipos de mensagens: *estimate*, *select*, *confirm* e *ready*; e três tarefas concorrentes: *Seleção*, *Confirmação* e *Decisão*. O diagrama na Figura 4 mostra a interação entre as réplicas.

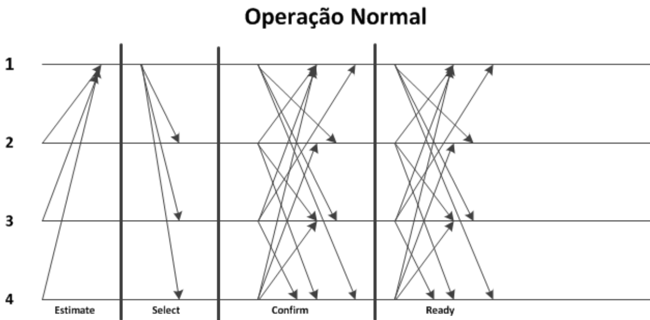

Figura 4: Interação entre réplicas em Kihlstrom, Moser e Melliar-Smith (2003)

1. *Seleção*: consiste de várias rodadas que, por sua vez, se dividem em três fases. Na primeira fase cada processo envia uma mensagem *estimate* contendo sua estimativa local para o coordenador. Na segunda, o coordenador aguarda até receber as estimativas de

$\frac{2n+1}{3}$ réplicas, seleciona uma estimativa baseada nos valores recebidos e envia uma mensagem *select* a todos os processos. Na terceira, cada processo aguarda até receber $\frac{2n+1}{3}$ mensagens *confirm* de diferentes processos utilizando assim o valor recebido para atualizar seus registros locais e envia uma mensagem *ready* para todos. Caso não receba as mensagens previstas na terceira fase, passa a suspeitar do coordenador.

2. *Confirmação*: ao receber uma mensagem *select* do coordenador da rodada, se ainda não enviou uma mensagem *confirm* para aquela rodada, envia uma mensagem *confirm* com o valor contido na mensagem *select* recebida.

3. *Decisão*: cada processo espera por $\frac{2n+1}{3}$ mensagens *ready* de processos distintos com o mesmo valor, decidindo então pelo valor recebido.

Ao decidir por um valor, cada processo trava este valor localmente e nenhum processo correto pode decidir em um valor diferente. Para tanto, nas próximas rodadas um processo que já decidiu por um valor passa a enviar ao coordenador uma mensagem *estimate* com um *timestamp* superior à rodada atual em que o coordenador escolherá o valor contido na mensagem com o maior *timestamp* para encaminhar aos processos. Caso o maior *timestamp* recebido seja zero, o coordenador decidirá por um valor em comum que tenha sido enviado em, ao menos, $\frac{n-1}{3} + 1$ mensagens.

Cada processo, ao receber uma mensagem, deve verificar a sua autenticidade, a partir da assinatura da mensagem pelo processo emissor. Além disso, é necessário que cada processo apenas aceite mensagens bem formadas e propriamente justificadas. Assim, cada mensagem enviada é formada por duas partes: afirmação, que contém a ação correspondente à mensagem, e justificativa que contém as afirmações anteriores que deram origem a esta afirmação. A forma correta das mensagem e suas respectivas justificativas se dá como segue:

- *estimate* tem a forma $<< estimate, p, r_p, e_p, ts_p >_p, confirms_p >_p$, sendo o campo *confirms*$_p$ correspondente à justificativa e contém as mensagens *confirm* que causaram a escolha de e_p caso esta já tenho ocorrido.

- *select* deve ser enviada na forma $<< select, c, r_c, es, ts >_c, estimates_c >_c$send que *estimates*$_c$ contém as mensagens *estimate* que causaram a mensagem.

- *confirm* possui a forma $<< confirm, p, r_p, e >_p, select_p >_p$ dado que a justificativa $select_p$ corresponde à mensagem *select* recebida do coordenador.

- *ready* utiliza o formato $<< ready, p, r_p, e >_p, confirms_p >_p$ sendo que a justificativa $confirms_p$ contém as $\frac{2n+1}{3}$ mensagens *confirm* recebidas que deram origem a mensagem.

A utilização deste algoritmo depende, porém, de um detector de faltas capaz de detectar que coordenador teve um comportamento faltoso devido ao envio de diferentes mensagens *select* em uma mesma rodada ou por tentar bloquear o algoritmo na terceira fase da primeira etapa. Desta forma, é proposto um algoritmo detector de faltas que consiste de cinco etapas concorrentes:

1. quando p envia uma mensagem *estimate*, inicia um *timeout* para a rodada r_p;

2. quando o *timeout* expira, p adiciona o coordenador da rodada na lista de suspeitos, e a rodada na lista de *timeouts* expirados;

3. ao receber $\frac{2n+1}{3}$ mensagens *confirm* de diferentes processos, o *timeout* é cancelado e caso este já tenha expirado, a rodada deve ser removida da lista de *timeouts* expirados e o coordenador removido da lista de suspeitos caso não esteja na lista $bizset_p$;

4. no recebimento de uma mensagem mal formada e devidamente assinada por um processo q, este é adicionado à lista de suspeitos e à lista $bizset_p$ de forma a não ser futuramente removido da lista de suspeitos;

5. Quando p receber, direta ou indiretamente, uma mensagem m que corresponde à outra mensagem *m'* com valor diferente, o emissor é adicionado na lista de suspeitos e na lista $bizset_p$.

3.3.3 Muteness Failure Detectors

Detectores de falhas de mudez[1] foram introduzidos por Doudou et al. (1999), de forma a simplificar os possíveis comportamentos bizantinos a serem reconhecidos pelo detector de falhas. Este modelo define que um processo é dito mudo em relação a um algoritmo A se deixa

[1] Muteness Failure Detectors

de enviar mensagens de A para um ou mais processos (DOUDOU et al., 1999)(DOUDOU; GARBINATO; GUERRAOUI, 2002)(DOUDOU; GARBINATO; GUERRAOUI, 2005).

O detector de falhas de mudez $\diamond M_a$ definido na abordagem se apresenta com as seguintes propriedades:

- *Eventual Mute Completeness*: Existe um momento após o qual todo processo tido como mudo por um processo p, continuará suspeito para sempre.
- *Eventual Weak Accuracy*: Existe um momento após o qual um processo correto p não será mais tido como mudo por nenhum outro processo.

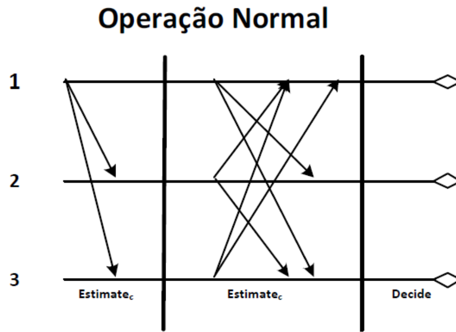

Figura 5: Detectores de mudez em operação sem falhas.

Na Figura 5 é apresentado os passos executados pelo algoritmo no caso de uma execução sem falhas. A implementação do algoritmo do detector de falhas de mudez $\diamond M_a$ inicia com a definição de um timeout $\triangle_p$, uma lista de possíveis suspeitos $critical_p$ e uma lista de suspeitos $output_p$. Sua execução ocorre em três algoritmos concorrentes:

1. Passa por todos os processos que estão em $critical_p$, e adiciona em $output_p$ aqueles processos q que não enviarem a mensagem *"q-is-not-mute"* para p;
2. Ao receber uma mensagem *"q-is-not-mute"*, remove q de $output_p$, sendo que qualquer mensagem recebida de q gera uma mensagem *"q-is-not-mute"* pelo algoritmo;
3. Ao receber do algoritmo uma nova lista $critical_p$, atualiza a lista atual e atualiza o *timeout* $\triangle_p$ do algoritmo a partir de uma função.

A interação entre A_p e $\diamond M_a$ se dá principalmente quando A_p consulta $\diamond M_a$ para obter a lista de processos suspeitos. Esta consulta porém, como mostrado no algoritmo, necessita de informações que são obtidas de A_p. A cada mensagem recebida por p, A_p envia para o detector uma mensagem *"q-is-not-mute"*, utilizada pelas etapas 1 e 2 do algoritmo. A_p também é responsável por enviar a cada início de rodada, uma nova lista com os processos críticos, a ser utilizada pela etapa 3 do algoritmo.

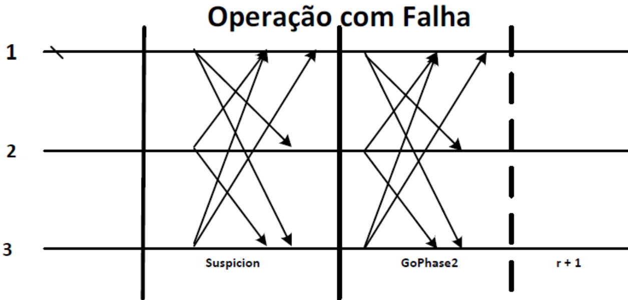

Figura 6: Detectores de mudez em operação com falhas.

Chamado EC[2], o algoritmo de consenso definido para trabalhar juntamente com o detector de falhas $\diamond M_a$ é baseado em rodadas, com um esquema de revezamento de coordenadores, e um modelo de sincronia parcial. Cada rodada é dividida em duas fases:

1. O coordenador da rodada envia sua estimativa a todos os processos que, por sua vez, reenviam a todos os processos com sua assinatura individual. Assim que recebe a mesma estimativa de N - f processos, envia a todos uma mensagem *decision*, comunicando o valor decidido.

2. No caso de um processo suspeitar do coordenador da rodada, este envia a todos uma mensagem *suspicion*. Ao receber esta mensagem, cada processo envia uma mensagem *GoPhase2* a todos os outros, contendo seu valor estimado. Se o processo, ao receber N - f destas mensagens, verificar que uma delas possui o mesmo valor sugerido pelo coordenador, assume este valor e se desloca para a rodada $r + 1$. Este processo pode ser visualizado na Figura 6.

[2] Early Consensus

Na primeira fase de cada rodada, cada processo tem o objetivo de decidir em um valor baseado na estimativa do do coordenador p_c e reenvia este valor, por broadcast, a todos os processos participantes. A decisão, porém, só ocorre quando o processo recebe ao menos N - f estimativas dos processos participantes. A segunda fase serve basicamente para garantir que, se um processo decidir ao final da rodada pelo valor do coordenador, todos os processos devem iniciar a próxima rodada com o mesmo valor. Assim, ao suspeitar do coordenador, cada processo espera por N - f processos enviarem uma estimativa e se ao menos um processo enviar o mesmo valor sugerido pelo coordenador na mensagem *GoPhase2*, então todos iniciam a próxima rodada com este valor.

3.3.4 The case for Byzantine fault detection

Técnicas BFT, apesar de sua eficiência, deixam a desejar normalmente em relação ao seu custo para implementação, exigindo, em sua abordagem tradicional, $3f+1$ réplicas para que seja possível tolerar até f faltas. Além disso, abordagens tradicionais BFT não possuem uma boa escalabilidade, tendo seu *throughput* reduzido conforme aumenta o número de servidores. De forma a otimizar estas deficiências, Haeberlen, Kouznetsov e Druschel (2006) apresenta uma abordagem em que técnicas de detecção são utilizadas no lugar de mascaramento de erros.

Nesta abordagem, cada réplica deve ser equipada com um detector de faltas capaz de monitorar outras réplicas em busca de comportamentos faltosos. Ao detectar algo estranho, as outras réplicas são avisadas e podem decidir sobre como agir em cada caso. A detecção de faltas é menos eficaz do que o mascaramento das mesmas, porém pode oferecer uma alternativa mais eficiente e escalável ao BFT para faltas com efeitos limitados ou recuperáveis.

Para esta detecção é necessário que cada ação possa ser associada inegavelmente com uma réplica. Desta forma o sistema deve garantir ao menos duas propriedades.

- Se uma réplica correta observa um comportamento faltoso, em algum momento será gerada uma evidência contra ao menos uma réplica faltosa.

- O sistema nunca gerará uma evidência válida contra uma réplica correta.

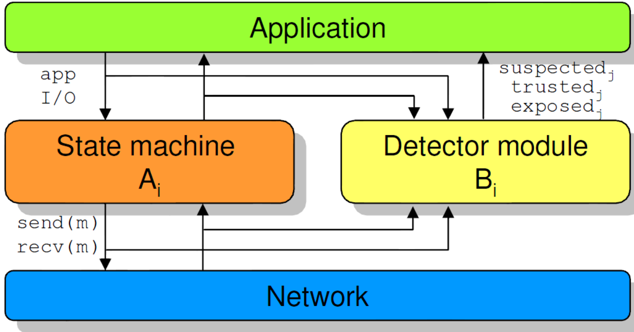

Figura 7: Arquitetura da abordagem em Haeberlen, Kouznetsov e Druschel (2006)

Cada réplica i é modelada como uma máquina de estados A_i e um módulo detector B_i. A arquitetura da abordagem pode ser observada na Figura 7. Sendo assim, pode-se dizer informalmente que uma réplica é correta caso respeite as especificações de A_i e B_i, sendo faltosa em caso contrário.

Uma execução E denota uma sequencia de eventos em que E/A_i é constituído por todos os eventos associados a A_i em E e E/B_i é constituído por todos os eventos associados a B_i. Assim, dizemos que uma réplica é correta se:

1. a sequencia de saídas produzidas em E/A_i e E/B_i são legais em relação à especificação de A_i e B_i e às entradas de E/A_i e E/B_i;

2. caso E seja infinito, E/A_i e E/B_i também são infinitos.

Para uma réplica correta, $\varphi(m)$ é o prefixo da execução do emissor de m até, incluindo, o envio de m. Desta forma, podemos dizer que uma mensagem é observável em E se existe uma réplica correta i e uma sequência de mensagens m_1, ..., m_k que:

1. $\mathrm{m}_1 = \mathrm{m}$;

2. o recebimento de m_k pertence a E/A_i;

3. para todo $j = 2$, ..., k, o recebimento de m_{j-1} pertence a $\varphi(m_j)$.

Assim, uma réplica pode ser assumida como faltosa quando uma mensagem m em E é observável porém: (1) $\varphi(m)$ não é válido; ou, (2)

existe uma mensagem *m'* em que $\varphi(m)$ é inconsistente com $\varphi(m')$. Uma réplica i também pode estar sob suspeita de falta se não existe algum fato que evidencie a falta mas exista alguma mensagem m enviada para i por uma réplica correta e, para nenhuma mensagem *m'* enviada por i, o recebimento da mensagem m não aparece em $\varphi(m')$.

Quando um detector B_i em uma réplica correta i observa um comportamento falto em uma réplica j, este envia uma indicação de falha: $confiável_j$, $suspeita_j$ ou $exposta_j$. Assim, o sistema deve garantir que as seguinte condições se mantenham em cada execução:

- em algum momento, qualquer réplica suspeita será suspeita para todas as réplicas corretas e, se uma réplica foi exposta em relação a uma mensagem m, então qualquer cúmplice (em relação a m) será exposto em algum momento por cada réplica correta;
- nenhuma réplica correta será suspeita eternamente por outra réplica correta e, nenhuma réplica correta será exposta por outra réplica correta.

Como forma de demonstrar a eficiência da abordagem, os autores desenvolveram o algoritmo *PeerReview*, que se apresentou prático e eficiente. De forma a garantir que a identidade de cada réplica, foi utilizada criptografia assimétrica em que cada réplica possui um par de chaves (publica e privada) assinadas por uma autoridade certificadora, assumindo que o atacante não tem poder computacional suficiente para quebrar a chave criptográfica.

Um *log* deve ser mantido por cada réplica do sistema, armazenando todas as entradas e saídas de A_i que deve, de tempos em tempos, apresentar um *hash* autenticado cobrindo todas as entradas até o momento de forma a garantir que uma réplica não altere seu *log*. Além de assinar as mensagens enviadas, cada réplica deve reconhecer o recebimento de todas as mensagens recebidas, caso contrario a réplica será indicada como suspeita pelo emissor da mensagem.

Cada mensagem enviada ou recebida contém um *hash* provando que esta era a última entrada do *log* correspondente. O receptor extrai este *hash* e envia às outras réplicas em algum momento, possibilitando assim que as réplicas interessadas fiquem cientes de todas as mensagens trocadas por réplicas válidas.

De forma a manter a integridade do sistema, cada réplica é auditada pelas outras periodicamente. Neste processo, a réplica auditora j solicita à réplica auditada i um *log* assinado cobrindo todas as entradas desde a última auditoria. Desta forma, j pode comparar com os últimos *hashes* recebidos da réplica, devendo marcar como suspeita

caso se recuse a enviar estes dados em um tempo pré-determinado. No caso de dois ou mais *hashes* serem inconsistentes, estes são enviados para as outras réplicas para que esta seja exposta por todas as réplicas corretas. Após isso, j extrai do log todos os *hashes* correspondentes a este segmento de log e os envia às outras réplicas de forma a que todas fiquem cientes de qualquer informação relevante mesmo se i for faltosa.

Para garantir a conformidade, j finaliza a auditoria inicializando uma cópia local da máquina de estados A_i e alimentando-a com todas as entradas do *log*, confirmando as saídas obtidas com as saídas informadas no log. Qualquer divergência serve como evidência para expôr i como faltosa.

No caso da detecção de uma falta, j obtém um dos dois tipos de evidência: divergências entre *hash* e *log* ou uma falha no teste de conformidade, para o caso de faltas detectáveis; ou uma indicação de que i não está respondendo. Em ambos os casos, a evidência pode ser encaminhada a outras réplicas que farão a verificação independentemente. Na ferramenta proposta *PeerReview*, é garantido que esta verificação sempre vai apresentar um resultado falso para réplicas corretas.

3.3.5 Enhanced Fault-Tolerance through Byzantine Failure Detection

Em sua proposta, Bazzi e Herlihy (2009) propõe um algoritmo capaz de solucionar ambos os problemas clássicos de acordo e *broadcast* desde que o número de processos participantes seja maior que $2f+3F+1$, sendo f o número de processos bizantinos transientes (detectáveis), e F representa o número de processos bizantinos permanentes (indetectáveis). Este modelo foi inspirado pelo *Sybil atack*, em que processos bizantinos podem assumir diferentes identidades e fazer com que o número de processos faltantes seja maior que a realidade.

O modelo considera um sistema síncrono, com N processos que se comunicam por troca de mensagens. Processos com comportamentos bizantinos podem ter um comportamento arbitrário, como o envio de mensagens maliciosas, ou apresentarem uma falha de *crash*. Assume-se que não é possível que um processo faltoso envie uma mensagem apresentando-se com a identidade de um processo correto.

Uma versão simplificada do protocolo de envio *broadcast*, chamado *one-shot*, ao ser executado, garante que todos os processos decidem no mesmo valor ou falham explicitamente na decisão, enviando $\star$. O protocolo deve satisfazer as seguintes propriedades:

- *Weak Validity*: se o emissor do *broadcast* é correto, o receptor também, e a mensagem recebida é diferente de $\star$, então a mensagem recebida é aquela enviada pelo emissor.

- *Strong Validity*: na ausência de falhas: se o protocolo é invocado em um momento antes de que todos os processos faltoso tenham falhado por *crash* e depois do qual não há novos *crashes*, então nenhum processo receberá $\star$.

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

- *Termination*: todo processo que iniciar o protocolo deve receber um valor ou falhar explicitamente.

No início do algoritmo, o processo b envia, por *broadcast*, o valor v iniciando o processo que segue em três etapas. Na primeira etapa, todos os processo trocam entre si suas visões, consistindo do seu valor atual e sua lista de processos faltosos, por $f + 2$ rodadas. Cada processo atualiza sua lista de faltosos com aqueles dos quais não foram recebidas nenhuma mensagem, atualizando seu valor local a partir do valor recebido de b se este não estiver na lista de faltosos, ou para $\perp$ caso contrário. Uma rodada é considerada limpa se não foram adicionados novos processos faltosos na lista.

Na segunda fase, um processo que passou por $f + 2$ rodadas limpas envia sua decisão, com uma cadeia vazia de testemunhas. Um processo q, ao receber uma cadeia pode tentar decidir neste valor desde que o tamanho desta cadeia não seja menor do que o número de rodadas limpas vistas por q. Para cada cadeia c que o processo q recebe de p, q adiciona p ao topo da cadeia e a armazena. Essas trocas continuam por $f + 1$ rodadas.

Na terceira fase, cada processo constrói uma lista com as cadeias em que o seu processo criador é tido como faltoso, removendo estas da lista de cadeias. Para todas as cadeias na lista de faltosas, se q pertence a uma delas, indicando a concordância com um processo faltoso, e não é conhecido como faltoso, então p remove da lista de cadeias qualquer uma encabeçada por q. Um processo decide em um valor ao encontrar uma cadeia encabeçada por um processo correto, e seguida apenas por processos sabidamente faltosos. Ao decidir por um valor, este é enviado a todos os outros processos que o adicionam a sua lista local de processos decididos.

O protocolo de *broadcast* como um todo é composto de um pipeline de execuções do algoritmo *one-shot*. A cada rodada é criado um *fork* do algoritmo *one-shot* e ao final de sua execução o valor

retornado é armazenado em uma lista a. O valor recebido pelo protocolo será o valor do primeiro round a retornar um valor diferente de $\star$.

O protocolo de consenso proposto se utiliza do algoritmo de *broadcast* apresentado e necessita $N > 2f + 3F$. Cada processo p inicia com uma variável v_d contendo seu valor de entrada e uma lista vazia com os processos sabidamente faltosos. A cada iteração, v_d é enviado, por *broadcast*, aos outros processos e todos os valores recebidos são armazenados. No caso de N - F processos não-faltosos concordarem no mesmo valor, p comunica um *landslide* para aquele valor e o adota localmente. No caso de a maioria dos processos concordarem com o valor, p comunica *majority* e adota o valor localmente. Caso nenhuma das situações anteriores ocorra, p mantém seu valor local.

Em uma segunda etapa, p envia novamente seu valor local para todos os outros processos e: se N - F processos concordam com *landslide*, p decide neste valor; se uma maioria dos processos concorda com *landslide* ou *majority*, p assume o valor mas não decide. Para garantir o término do algoritmo, ao final de cada iteração, p avalia N - F processos potencialmente corretos e atualiza seu valor local de acordo com o valor majoritário neste conjunto. Assim, este algoritmo tem seu término garantido desde que em algum momento, após suficiente número de iterações, p examine o conjunto contendo os N - F processos corretos.

3.4 ABORDAGENS BASEADAS EM VIRTUALIZAÇÃO

3.4.1 VM-FIT: Supporting Intrusion Tolerance with Virtualisation Technology

VM-FIT foi proposto por Reiser e Kapitza (2007) de forma a aproveitar o isolamento oferecido por tecnologias de virtualização para oferecer uma abordagem tolerante a intrusões. A lógica da aplicação fica isolada em um domínio, enquanto a lógica da replicação se dá em outro domínio chamado *NV*. Esta abordagem necessita $2f + 1$ réplicas para tolerar f faltas. Entretanto, apesar de suportar faltas arbitrárias nos domínios da aplicação, suporta apenas faltas de parada nos domínios *NV* (REISER; KAPITZA, 2008). Em uma variação proposta, é possível tolerar faltas arbitrárias mesmo nos domínios *NV*, porém ao custo de $3f + 1$ réplicas (REISER; KAPITZA, 2007).

Outra vantagem nesta proposta está na facilidade e promover

uma recuperação pró-ativa. Enquanto métodos tradicionais de recuperação pró-ativa não podem ser executados durante uma intrusão ou exigem algum dispositivo de *hardware* à prova de falhas para efetuar a reinicialização, a virtualização da abordagem permite que o domínio NV atue como este dispositivo e reinicialize o domínio da aplicação para um estado limpo de uma forma segura. Para efetuar esta reinicialização, o VM-FIT inicia um novo domínio, atualiza o estado deste domínio para o estado atual da réplica a ser reinicializada e substitui a réplica em uso pela nova possibilitando que este processo seja efetuado em um tempo bastante pequeno e sem a necessidade de máquinas físicas adicionais.

Para o funcionamento da abordagem é necessário que as seguintes suposições se mostrem verdadeiras:

- Toda a comunicação entre o cliente e as réplicas pode ser interceptadas em nível de rede.
- O serviço pode ser modelado como uma máquina de estados determinística.
- Réplicas do serviço pode falhar arbitrariamente porém apenas $f < \frac{n-1}{2}$ réplicas falham simultaneamente.
- O VMM e o domínio NV são suscetíveis apenas à falhas de parada.

Toda a comunicação do cliente chega apenas no domínio NV, que intercepta as mensagens antes de enviar às réplicas. Apesar deste domínio permitir apenas faltas de parada, pode ser implementado como um sistema minimalista para evitar sua intrusão. Essa interceptação de mensagens permite uma abordagem transparente para tolerância a intrusões, sem um protocolo específico de consenso. Além disso, a recuperação pró-ativa permite um rejuvenescimento das réplicas e aumenta a resistência do sistema.

3.4.2 ZZ and the Art of Practical BFT Execution

Wood et al. (2011) apresenta uma proposta com o intuito de diminuir a quantidade de máquinas necessárias para tolerar f faltas. A ideia consiste basicamente em manter $f + 1$ réplicas ativas na execução do serviço e ativar réplicas adicionais apenas no caso de suspeita de falhas. Para fornecer estas réplicas adicionais com um menor custo são empregadas técnicas de virtualização. Assim, em servidores que

possuem múltiplas aplicações, um número maior de máquinas virtuais do que a capacidade física do servidor pode estar em estado de espera, já que dificilmente todas estarão ativas simultaneamente (WOOD et al., 2011).

Apesar de necessitar apenas entre $f+1$ (melhor caso) e $2f+1$ (pior caso) réplicas da aplicação, a abordagem ainda necessita $3f+1$ réplicas de acordo, separando as réplicas de acordo e aplicação. Esta separação, também apresentada em Yin et al. (2003), leva em consideração que as réplicas de acordo tem um custo muito menor do que as réplicas da aplicação. Esta abordagem, entretanto, exige que, no máximo, uma réplica de aplicação e uma réplica de acordo sejam executadas simultaneamente em cada máquina física.

Ao iniciar o protocolo o cliente envia a requisição às réplicas de acordo ($3f+1$), que são responsáveis por atribuir um número de sequência para a requisição e a encaminha para as réplicas da aplicação. Ao receber $f+1$ respostas idênticas das réplicas da aplicação, as réplicas do acordo sabem que foi executada por, ao menos, uma réplica correta e encaminham a resposta ao cliente. Ao receber respostas diferentes para a mesma requisição, ou aguardar mais do que um tempo definido, cada réplica de acordo j envia uma solicitação de recuperação para a VMM, que remove uma réplica do estado dormente ao receber $f+1$ solicitações iguais. Ao ser ativada, a réplica deve obter o *checkpoint* mais recente e reexecuta todas as requisições mais recentes do que o checkpoint.

A abordagem garante as propriedades *safety* e *liveness* garantindo que: 1) toda resposta recebida é correspondente a uma requisição válida, 2) todas as réplicas concordam na sequência atribuída à requisição e 3) a resposta recebida é a mesma de um serviço centralizado. A primeira vem das réplicas de acordo apenas gerarem requisições com base em requisições válidas de clientes. A segunda é garantida pelo protocolo de acordo proposto por Castro e Liskov (1999b). A terceira suposição é valida desde que todas as réplicas comecem a execução no mesmo estado inicial.

A abordagem por Wood et al. (2011) se torna interessante por economizar recursos executando as requisições em apenas $f+1$ réplicas em caso normal. Esta abordagem, entretanto, perde um pouco seu brilho ao continuar exigindo que $3f+1$ réplicas de acordo estejam ativas em qualquer momento da execução do protocolo. Outro ponto desfavorável está no armazenamento das réplicas ociosas pois se torna vantajoso apenas no caso do *cluster* executar diversos serviços. Um *cluster* dedicado precisará de ma quantidade de réplicas semelhantes à

proposta de Yin et al. (2003).

3.4.3 Intrusion Tolerant Services Through Virtualization: A Shared Memory Approach

De forma a diminuir o custo de abordagens de tolerância a faltas bizantinas (BFT), a abordagem SMIT se utiliza de técnicas de virtualização para diminuir o número de réplicas N necessárias para tolerar f faltas de $N \geq 3f+1$, em abordagens tradicionais, para $N \geq 2f+1$. Isso se dá principalmente pela utilização da memória compartilhada provida pelo gerenciador de máquinas virtuais (VMM) para a comunicação entre as réplicas, simplificando assim o algoritmo de consenso (STUMM et al., 2010).

SMIT se baseia na utilização de uma máquina física, hospedeira, com uma VMM suportando as réplicas como máquinas virtuais convidadas sendo que estas possuem acesso a um espaço de memória compartilhada no hospedeiro. A VMM deve oferecer às réplicas um total isolamento entre elas de forma que um atacante com acesso a uma réplica não tenha acesso às outras, ou ao hospedeiro. A impossibilidade do acesso ao hospedeiro a partir de uma máquina virtual deve ser garantida pela VMM e o acesso direto a este pode ser facilmente bloqueado com a utilização de um *firewall*. Para garantir a autenticidade das mensagens são utilizadas técnicas criptográficas.

Cada réplica no sistema pode executar dois papéis distintos: réplica primária é responsável por definir a ordem em que as requisições devem ser executadas; réplicas backup, que são responsáveis por executar as requisições na ordem indicada pela réplica primária. O comportamento bizantino no sistema engloba basicamente: parada na execução; omissão de mensagem; mensagem inconsistente com o protocolo do sistema. De forma a não permitir que um atacante explore uma mesma falha em réplicas distintas, é necessário um esquema de diversidade de *software* entre as réplicas em nível de sistema operacional e aplicações.

O componente de memória compartilhada utilizado para a troca de mensagens entre as réplicas, chamada *postbox*, deve ser fornecido pelo hospedeiro a partir do VMM. Possui uma interface simples, com apenas dois métodos:

- *append(value):boolean*

- *read():value*

O primeiro armazena um valor qualquer, juntamente com a identificação do emissor, retornando um valor booleano indicando se a escrita ocorreu com sucesso. O segundo retorna o valor subsequente ao último valor lido pela réplica, como em uma fila. Este componente trabalha no modo *append-only*, de forma que qualquer dado enviado não pode ser alterado posteriormente, impedindo assim que diferentes réplicas leiam diferentes versões de um mesmo valor. Cada entrada será elegível para o *garbage collector* assim que todas as réplicas corretas tiverem lido a informação.

Seguindo a linha de outros algoritmos na abordagem de Replicação Máquina de Estados, o algoritmo deve prover duas propriedades: *Safety* indica que o conjunto de réplicas deve se comportar como se o sistema estivesse sendo executado em uma única máquina; *Livenes* exige a garantia de que toda requisição enviada por um cliente correto terá sua execução completada em algum momento. Para isso é necessário que as mensagens entregues sejam executadas na mesma ordem por todas as réplicas corretas, além de o número de faltas não ser maior do que $(\frac{N-1}{2})$. Esta ordem é definida pela réplica primária e seguida pelas réplicas *backup* sendo que um protocolo de troca de visão é executado quando a réplica primária se torna suspeita.

O cliente envia, por *multicast*, uma requisição para todas as réplicas participantes. A réplica primária s, tal que $s = v \ mod \ |S|$ para a visão v e o conjunto de réplicas S, grava a requisição recebida na *postbox* sendo que qualquer gravação feita por alguma das réplicas backup será simplesmente ignorada. O cliente então aguarda até receber ao menos $f+1$ respostas idênticas.

O algoritmo se divide basicamente em duas tarefas concorrentes. Uma delas trata de receber as mensagens, adicionar a um buffer local e, no caso da réplica primária, gravar um *hash* da mensagem na *postbox*. A segunda tarefa lê o *hash* de uma mensagem da *postbox*, na mesma ordem em que foram gravados, obtém a mensagem a partir de seu buffer local e executa sua requisição, retornando o resultado para o cliente. Caso a mensagem não seja localizada em seu *buffer* local após um *timeout*, passa a suspeitar da réplica primária e tenta executar uma troca de visão. Esta troca de visão será efetuada apenas se $f+1$ suspeitarem da corretude da réplica primária.

3.5 CONSIDERAÇÕES FINAIS

Este capítulo apresentou uma revisão da literatura de trabalhos relacionados com a abordagem proposta. O primeiro trabalho apresentado (CASTRO; LISKOV, 1999a) é considerado um patamar na literatura por ser a primeira abordagem prática tolerante a faltas bizantinas. A partir dele, várias outras abordagens surgiram para com a proposta de oferecer um melhor desempenho a partir de otimizações específicas. Dentre estes, foram apresentados (YIN et al., 2003; CORREIA; NEVES; VERÍSSIMO, 2013; KOTLA et al., 2008; REISER; KAPITZA, 2007; WOOD et al., 2011; STUMM et al., 2010; VERONESE et al., 2013).

Alguns trabalhos relacionados com detectores de faltas foram apresentados pois seus conceitos serviram de base para a elaboração do TwinBFT, apesar de estas abordagens não estarem relacionadas diretamente com RME (MALKHI; REITER, 1997; KIHLSTROM; MOSER; MELLIAR-SMITH, 2003; DOUDOU et al., 1999; HAEBERLEN; KOUZNETSOV DRUSCHEL, 2006; BAZZI; HERLIHY, 2009).

A Tabela 1 apresenta uma comparação entre algumas abordagens presentes atualmente na literatura e que podem ser diretamente comparáveis com a abordagem proposta devido às suas características em comum. São comparados os seguintes atributos:

- Número de réplicas: indica o número de réplicas lógicas do sistema.
- Número de processos: indica o número de processos participates do procolo. Geralmente o número de processos será igual ao número de réplicas.
- Número de máquinas físicas: indica o número de servidores físicos necessários ao protocolo. Normalmente, este número será igual ao número de réplicas.
- Número de passos: apresenta o número de passos necessários para uma execução sem falhas de uma requisição.
- Especulativo: indica se o algoritmo correspondente se aproveita de técnicas especulativas para oferecer o serviço.

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

Tabela 1: Comparação Entre Propriedades dos Protocolos Avaliados

	Número réplicas	**Número processos**	**Número físicas**	**Passos**	**Especulativo Otimista**
PBFT (CASTRO; LISKOV, 1999b)	$3f+1$	$3f+1$	$3f+1$	5	não
Zyzzyva (KOTLA et al., 2008)	$3f+1$	$3f+1$	$3f+1$	3 / 5[1]	sim
Yin (YIN et al., 2003)	$2f+1$	$3f+1$	$3f+1$	4	não
BFT-TO (CORREIA; NEVES; VERÍSSIMO, 2013)	$2f+1$	$2f+1$	$2f+1$	5	não
MinBFT (VERONESE et al., 2013)	$2f+1$	$2f+1$	$2f+1$	4	não
MinZyzzyva (VERONESE et al., 2013)	$2f+1$	$2f+1$	$2f+1$	3 / 5[1]	sim

4 REPLICAÇÃO DE MÁQUINA DE ESTADOS TOLERANTE A FALTAS BIZANTINAS USANDO MÁQUINAS VIRTUAIS GÊMEAS

Diversas abordagens tolerantes a faltas emergiram nas últimas décadas. Devido à necessidade de replicação, entretanto, muitas destas não são viáveis na prática ou se mostram viáveis apenas em situações específicas.

A técnica de Replicação de Máquina de Estados (SCHNEIDER, 1990) fornece uma arquitetura replicada e atua como base em boa parte das abordagens atuais (CASTRO; LISKOV, 1999b; YIN et al., 2003; KOTLA et al., 2008; CORREIA; NEVES; VERISSIMO, 2004; STUMM et al., 2010). Esta replicação permite que faltas sejam mascaradas sem prejuízo para o funcionamento do serviço da aplicação.

Apesar de faltas de *crash* poderem ser contornadas facilmente com o uso de replicação, faltas arbitrárias - ou bizantinas - exigem um maior cuidado devido à sua complexidade.

Com o intuito de oferecer uma proposta mais viável para solucionar o problema de tolerância a faltas, é proposto neste trabalho um algoritmo tolerante a faltas bizantinas a partir da utilização de replicação de máquina de estados, chamado **TwinBFT**. Foram utilizados conceitos de virtualização para desenvolver uma arquitetura em que, cada réplica é dividida em um conjunto de processos virtualizados do serviço, estes processos atuam como um validador de mensagens para outros processos na mesma réplica. Esta validação resulta, na prática, em uma transformação de faltas bizantinas para faltas de omissão, que podem ser mascaradas de forma bem mais simples.

O foco da abordagem está na utilização de virtualização para replicar o serviço dentro de uma mesma réplica física, fornecendo assim duas camadas de replicação do serviço, sendo que em uma das camadas, tem-se um algoritmo para tolerância a faltas de omissão e em outra, mais interna, tem-se a transformação das faltas bizantinas em faltas de omissão. Com isto, a arquitetura proposta pode ser implementada com um conjunto de $2f+1$ máquinas físicas para se tolerar f faltas. Considerando todos os processos envolvidos, o TwinBFT exige pelo menos $4f+2$ processos para tolerar até f faltas arbitrárias.

A execução deste protocolo em um ambiente virtualizado é de suma importância, já que a virtualização é capaz de oferecer, além do isolamento total das máquinas físicas, meios mais diretos para a comunicação entre processos. Outras abordagens surgiram a partir da

utilização de componentes confiáveis para a reduzir o custo de abordagens tolerantes a faltas bizantinas (CORREIA; NEVES; VERISSIMO, 2004; VERONESE et al., 2013). No TwinBFT, este conceito é empregado utilizando as vantagens fornecidas pela virtualização, que é bastante difundida no mercado, como um componente confiável.

Será detalhado na Seção 4.1 o modelo do sistema com as premissas necessárias ao funcionamento do protocolo. A seguir, será apresentado uma definição detalhada da memória compartilhada na Seção 4.2, que atua de forma central na camada de detecção de falhas. Em seguida, será detalhado o protocolo da abordagem, bem como o seu algoritmo, na Seção 4.3. Por fim, na Seção 4.4 serão expostas as provas de correção do algoritmo desta proposta.

4.1 MODELO DO SISTEMA

Uma representação abstrata da arquitetura do sistema é mostrada na Figura 8. O sistema é composto por um conjunto de n máquinas físicas (ou *hosts*) $H = \{h_1, h_2, \ldots, h_n\}$ sendo que $n \geq 2f+1$ e f é o número máximo de máquinas físicas que podem estar faltosas em um momento qualquer da execução do serviço. Cada *host*, conforme mostrado na Figura 8 contém um gerenciador de máquinas virtuais (VMM ou hipervisor) com um conjunto de m máquinas virtuais $\{vm_i, vm_i', \ldots, vm_i^{m-1\prime}\}$, sendo $m \geq 2$. Estas máquinas virtuais são chamadas gêmeas e executam em cada uma, uma réplica do processo, respectivamente p_i, p_i', $\ldots$, $p_i^{m-1\prime}$. Cada um dos processos $\{p_i,\ p_i',\ \ldots,\ p_i^{m-1\prime}\}$ executa o mesmo serviço e se comunica com os outros processos presentes no mesmo *host* para validar cada mensagem de saída antes de enviar para os processos presentes em outras máquinas físicas.

É assumido que até f máquinas virtuais podem falhar de forma bizantina ou arbitrária, mas apenas uma minoria em cada máquina física. Quando uma máquina virtual falha arbitrariamente, o mecanismo de validação transforma essa falha numa omissão. Isso é possível pois cada um dos processos $\{p_i,\ p_i',\ \ldots,\ p_i^{m-1\prime}\}$ em um mesmo host executa o mesmo serviço da aplicação e como cada um destes processos é uma máquina de estados determinística equivalente, todos estes processos devem chegar ao mesmo resultado, dado que cada um iniciou no mesmo estado e recebeu as mesmas entradas. Caso isto não aconteça, a mensagem é ignorada por todos os processos presentes em outros *hosts*, se transformando, portanto, em uma omissão.

O modelo assume que, como uma máquina física contém m

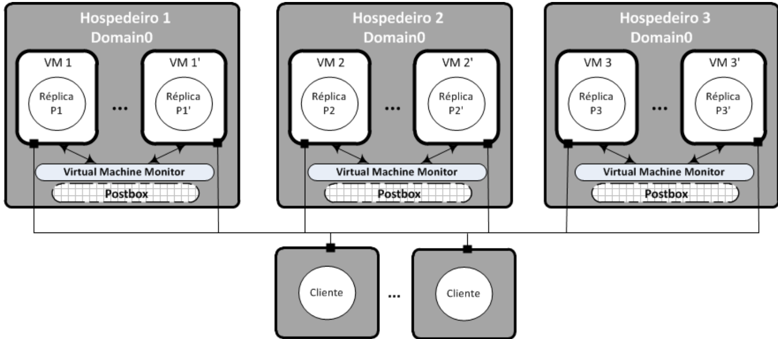

Figura 8: TwinBFT - Arquitetura com Máquinas Virtuais Gêmeas.

máquinas virtuais, uma mensagem é considerada válida quando a condição $m \geq 2fVM + 1$, sendo que fVM é o número máximo de máquinas virtuais faltosas em uma máquina hospedeira. Neste caso, a máquina hospedeira não será considerada faltosa se a maioria das VMs $(fVM + 1)$ fornecerem a mesma resposta.

Além disto, é suposto também que até f máquinas físicas podem falhar por *crash* ou omitindo mensagens acidentalmente ou devido à falha de uma das suas máquinas virtuais. Para substanciar o limite de f falhas, é necessário adotar mecanismos que ajudem a diminuir a chance de ocorrerem faltas simultâneas.

Um mecanismo bastante utilizado neste sentido é a *diversidade de software*, ou seja, implementações diferentes dos processos e máquinas físicas e utilização de diferentes sistemas operacionais em cada réplica (BESSANI et al., 2009; GARCIA et al., 2011; GASHI; POPOV; STRIGINI, 2007; AVIZIENIS et al., 2004). Essa diversidade reduz a chance de máquinas virtuais de um mesmo *host* sofrerem intrusão simultaneamente. Isso se dá porque apesar de todo *software* poder apresentar vulnerabilidades, diferentes versões de um mesmo *software* ou diferentes sistemas operacionais estarão sujeitos a diferentes vulnerabilidades, dificultando o trabalho de um atacante. A diversidade de software, entretanto, é dependente do projeto do serviço replicado e não é o foco deste trabalho.

É uma premissa do sistema também que a virtualização forneça um completo isolamento, tanto entre as máquinas virtuais quanto do próprio VMM / hipervisor. No entanto, a segurança do hipervisor é essencial para obter isolamento, e algumas diferentes técnicas podem ser usadas para este fim. Técnicas como desagregação propõe *desagregar* o sistema de virtualização como solução para diminuir a *trusted com-*

puting base do sistema (MURRAY; MILOS; HAND, 2008). O sistema *NoHype* vai mais longe retirando o hipervisor do caminho e executando as máquinas virtuais de modo nativo (SZEFER et al., 2011). O sistema *HyperSafe* usa outra abordagem: protege hipervisor de modo a detectar ataques que modifiquem o fluxo de controle, como *buffer overflows* (WANG; JIANG, 2010). A utilização destes mecanismos torna a premissa de isolamento da VMM mais realista. Para tornar o *host* físico totalmente inacessível externamente existem várias opções, como a remoção dos *drivers* de rede do sistema operacional do *host* (STUMM et al., 2010; FRASER et al., 2004), desta forma, apenas as VMs são visíveis à rede (Figura 8).

O VMM deve fornecer uma abstração de memória compartilhada, a *postbox*, com uma comunicação segura, síncrona e FIFO entre as VMs gêmeas. Os detalhes da implementação deste espaço de memória serão apresentados na Seção 4.2. Para evitar que a *postbox* atinja seu limite, é necessário que um mecanismo de *garbage collection* remova as entradas antigas, já processadas pelo sistema.

Processos faltosos podem se comportar arbitrariamente, ou seja, é assumido um modelo de faltas bizantinas. Entretanto, não mais do que $f = \frac{n-1}{2}$ *hosts* podem ser faltosos ao mesmo tempo e $m \geq 2fVM + 1$, sendo que fVM é o número de VMs faltosas dentro de *host* h_i. Dada uma dada entrada, é necessário que haja uma maioria de respostas idênticas em um host para que a máquina física não seja considerada faltosa.

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

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

host é, por definição, p_j.

São utilizadas técnicas de criptografia simétrica para autenticar mensagens e garantir sua autenticidade. Cada par de processos compartilha entre si uma chave secreta usada para gerar um vetor de MACs (*Message Authentication Code*) (TSUDIK, 1992) com um MAC válido para cada processo. A utilização de chaves criptográficas assimétricas também é possível porém a criptografia simétrica foi escolhida por questões de desempenho.

Apesar de ter sido feito um estudo estudo mais detalhado sobre a diferença no desempenho com a utilização de criptografia simétrica ou assimétrica, intuitivamente a utilização de vetores de MACs com criptografia simétrica deve apresentar um melhor desempenho para um conjunto menor de réplica, enquanto a criptografia assimétrica deve apresentar uma melhor escalabilidade. Isso ocorre porque a quantidade de assinaturas a serem processadas cresce exponencialmente com vetores de MACs, em relação ao número de réplicas, enquanto a quantidade de assinaturas a serem processadas com critografia simétrica cresce linearmente.

É previsto que não é possível para um processo se passar por outro devido à utilização de vetores de MACs. Para garantir a identidade de uma mensagem, o processo que a originou e cada processo subsequente dentro de um mesmo host que concorde com o conteúdo da mensagem, i.e., que tenha processado localmente e criado a mesma mensagem, adiciona sua assinatura baseada no *hash* da mensagem dentro de um conjunto de assinaturas denotado pelo símbolo σ.

4.2 MEMÓRIA COMPARTILHADA

O ponto chave no algoritmo TwinBFT está na utilização de uma comunicação confiável entre as máquinas gêmeas. Esta memória compartilhada, chamada *postbox*, deve ser fornecida pela VMM para as máquinas virtuais de forma a oferecer uma comunicação FIFO entre estas. Ao publicar uma nova mensagem na *postbox*, esta deve ser difundida a todas as máquinas gêmeas, que devem efetuar a leitura ordenada. O componente utilizado segue basicamente o mesmo formato apresentado por Stumm et al. (STUMM et al., 2010).

Este componente deve fornecer dois serviços básicos:

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

Ao adicionar uma nova mensagem a ser difundida, utiliza-se o serviço *append* que será responsável por anexar a mensagem na memória compartilhada. Esta mensagem será sempre adicionada ao fim do espaço, após a última mensagem inserida. As mensagens gravadas devem ser identificadas atomicamente e apresentar mecanismos para identificação do seu emissor. Neste caso é utilizado vetores de MAC para autenticação das mensagens. Ao ler uma mensagem, a *postbox* deverá retornar sempre a próxima mensagem não lida pela réplica.

A VMM deve garantir que seja impossível alterar qualquer mensagem enviada para a *postbox*, ou seja, a memória deve ser *append-only*. Após enviada, a mensagem deve poder apenas ser lida e, posteriormente, removida pelo *garbage collector*. O controle de acesso aos dados da *postbox* deve ser coordenado pela VMM no fornecimento do serviço aos sistemas convidados.

4.3 PROTOCOLO TWINBFT

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

Nesta seção, é apresentada a proposta de transformação das faltas bizantinas em faltas de omissão. Neste sentido, será apresentada uma adaptação do protocolo PBFT (CASTRO; LISKOV, 1999b) para o modelo proposto. Em cada visão, apenas uma réplica p_j é primária (líder), e é responsável por definir a ordem das mensagens e encaminhar as requisições para as réplicas do serviço. Se uma mensagem enviada por um *host* qualquer for assinada por uma maioria de VMs deste *host*, ou seja, uma maioria de processos gerou uma mensagem idêntica, assumimos esta mensagem como correta, já que apenas uma minoria de VMs pode falhar ao mesmo tempo em um mesmo *host*.

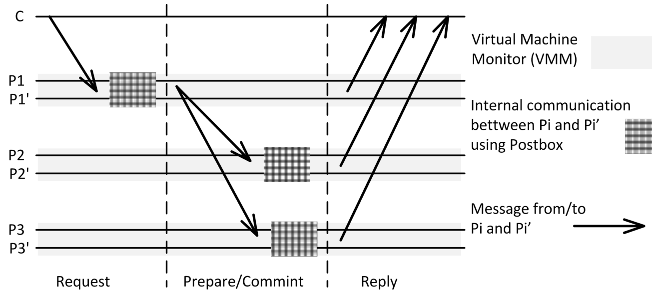

Figura 9: Passos de comunicação do protocolo em execução normal.

4.3.1 Propriedades

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

- **Ordem Total** (*safety*): cada requisição é executada sequencialmente na mesma ordem em cada réplica, i.e., apesar da replicação, as operações são executadas da mesma forma como seriam em um sistema centralizado.

- **Terminação** (*liveness*): qualquer requisição iniciada pelo cliente é terminada em algum momento, mesmo em caso de falha.

O algoritmo proposto fornece tanto *safety* quanto *liveness*, assumindo que não mais do que $f = [\frac{n-1}{2}]$ máquinas físicas, ou *hosts*, são faltosos e, ao menos uma maioria de processos sejam corretos em cada *host* faltoso. Para garantir que as réplicas executarão as requisições na mesma ordem, todas as réplicas seguem a ordem definida pelo líder e esta ordem pode ser considerada correta desde que assinada por uma maioria de processos na réplica primária. O protocolo garante *safety* apesar do tempo levado para o processamento das requisições, porém uma certa sincronia é necessária para garantir *liveness*.

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

4.3.2 Algoritmo Proposto

O algoritmo proposto funciona de forma bastante simplificada, já que a detecção de falhas é tratada pelos detectores existentes em cada réplica, tornando desnecessários passos para um acordo entre as réplicas. Esta ausência do acordo, porém, pode levar a uma inconsistência em que duas réplicas corretas possuem diferentes estados. Portanto, a abordagem se utiliza também de algoritmos de troca de visão e sincronia entre as réplicas. Na Figura 9 é apresentado um diagrama com o caso normal de operação, ou seja, livre de falhas.

Em seguida está apresentado uma definição informal do processo, levando em consideração uma arquitetura em que cada máquina física comporta apenas duas máquinas virtuais, por questões de simplicidade:

1. Cliente envia a requisição para ambos os processos encapsulados nas máquinas virtuais da réplica primária;

2. O líder primário p_i define um número de sequência para a requisição recebida ser executada e insere uma mensagem “ORDER"na *postbox*;

3. O seguidor primário p'_i lê a mensagem da *postbox*, pega o número de sequência, verifica se este número atende aos requisitos para o estado atual e insere na *postbox* a mensagem “ORDER” contendo a requisição original e o número de sequência recebido;

4. Ambos os processos assinam a mensagem “ORDER” lida da *postbox* e enviam para todas as réplicas backup;

5. Assim que cada processo, inclusive os processos na réplica primária, recebem a mensagem “ORDER”, a operação requisitada pelo cliente é executada e uma mensagem “REPLY” assinada pela réplica é inserida na *postbox*;

6. Quando um processo lê da *postbox* uma mensagem “REPLY", compara com a mensagem gerada localmente e, se todos os argumentos forem idênticos, adiciona sua assinatura e encaminha a resposta para o cliente;

7. Se o cliente receber ao menos $f + 1$ respostas corretamente assinadas de diferentes réplicas, aceita a resposta.

Qualquer mensagem recebida por uma réplica que não esteja corretamente assinada por uma maioria de máquinas gêmeas presentes na

réplica será automaticamente descartada pelo protocolo. Uma réplica passa a identificar outra como faltosa quando não recebe uma mensagem prevista, sendo que não faz diferença se esta mensagem nunca foi recebida ou se ela foi recebida e descartada como uma mensagem inválida.

A única ação, entretanto, que se dá devido à uma réplica incorreta é a troca de visão. Isto só acontece caso a réplica faltosa seja a réplica primária pois falhas nas réplicas backup não influenciam no andamento do protocolo.

4.3.3 Comportamento do Cliente

Como um exemplo de cliente, o Algoritmo 1 mostra uma execução normal em que o cliente envia uma requisição (linha 3) para o serviço. Ao mesmo tempo, outra tarefa separada recebe continuamente as respostas enviadas pelas réplicas (linhas 1-8). Assim que se verifica o recebimento de ao menos $f+1$ respostas válidas de diferentes réplicas (linha 4), o cliente cancela o *timeout* existente para a requisição (linha 5) e aceita a resposta recebida (linha 6).

A requisição tem o formato $\langle$REQUEST, c, seq, op$\rangle_\sigma$ sendo que *c* é o identificador do cliente, *seq* é o número de sequência em relação ao cliente, e *op* é a operação a ser executada. Se o cliente não receber $f+1$ respostas em um determinado tempo, reenvia a requisição para todas as réplicas (linha 1), renovando o *timeout* para garantir a retransmissão da mensagem até que uma resposta seja recebida.

4.3.4 Execução Normal do Protocolo

O algoritmo, executado em cada um dos processos possui duas tarefas concorrentes. A Tarefa 1, apresentada no Algoritmo 2 é responsável por ler as mensagens recebidas através da rede. A Tarefa 2, definida no algoritmo 3 é responsável por ler as mensagens recebidas a partir da *postbox*, inseridas por sua gêmea. O estado de cada processo é composto pelo estado do serviço, um *buffer* de mensagens e a visão atual. Este estado é compartilhado entre as tarefas.

Quando qualquer um dos processos $\{p_i, p'_i\}$ no *host* primário recebe a requisição do cliente, p_i gera um novo número de sequência n e cria uma mensagem $\langle\langle$ORDER, p_i, v, n, dm$\rangle_\sigma$, m$\rangle_\sigma$, sendo v o número da visão atual, e *dm* o resumo da mensagem m (linhas 4-6). Assim que

Algoritmo 1 Lado Cliente

```
/* Tarefa 1:  request */
```

1: **procedure** REQUEST ▷ Envia uma nova requisição
2: $\Delta_{seq} \leftarrow$ default *timeout*
3: multi_send($\langle$REQUEST, c, seq, op$\rangle_{\sigma}$) ▷ Envia a requisição para ambos os processos na réplica primária
4: **end procedure**

```
/* Tarefa 2:  receiving */
```

1: **loop**
2: $msg \leftarrow receive()$
3: $buffer \leftarrow buffer \cup msg$
4: **if** $f+1$ replies with $seq = msg.seq$ from different hosts $\exists$ buffer **then**
5: stop($\Delta_{msg.seq}$) ▷ Cancela o timeout da mensagem
6: accept(msg) ▷ Aceita a resposta recebida para a mensagem
7: **end if**
8: **end loop**

```
/* Tarefa 3:  timeout */
```

1: **if** Δ_{seq} expired **then**
2: $\Delta_{seq} \leftarrow$ default *timeout*
3: multi_send($\langle$REQUEST, c, seq, op$\rangle_{\sigma_c}$) ▷ Envia a mensagem para todas as réplicas
4: **end if**

p'_i lê a mensagem "ORDER" inserida na *postbox* por p_i e possui sua respectiva mensagem "REQUEST"no *buffer*, pega o número de sequência proposto por p_i e verifica sua validade a partir dos seguintes critérios:

- A mensagem está corretamente assinada pelo líder primário.
- Não foi executada ainda nenhuma requisição com este número de sequência para este cliente, nesta visão.
- O número de sequência atribuído está entre um mínimo h e um máximo H, apresentados em maior detalhe na Seção 4.3.6.

Caso o número de sequência proposto seja válido, cria uma mensagem "ORDER", assina e insere na *postbox* (linha 6). Quando cada um dos processos lê a mensagem "ORDER" da *postbox*, verifica se todos os parâmetros correspondem aos processados localmente e, em caso positivo, adiciona sua própria assinatura à mensagem de sua gêmea (linha 8). Caso a mensagem possua assinaturas suficientes para ser conside-

Algoritmo 2 Algoritmo em operação normal: Tarefa 1 - rede

```
1: loop
2:     msg ← receive()
3:     if received (REQUEST) then
4:         if é o líder primário then
5:             n ← n+1
6:             postbox.append(⟨⟨ORDER, p_i, v, n, dm⟩_σ, msg⟩_σ)
7:         else if é o seguidor primário then
8:             buffer ← buffer ∪ msg
9:         else
10:            envia mensagem para primários
11:            inicia Δ_p
12:        end if
13:    else if received (ORDER) then
14:        termina Δ_p
15:        postbox.append(⟨REPLY, p_i, v, seq, c, res⟩_σ)
16:    end if
17: end loop
```

rada válida, envia para todas as réplicas (linha 10). Caso contrário, a mensagem é inserida novamente na *postbox*.

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

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

Ao receber uma mensagem “ORDER” de ambos os processos em uma máquina física, cada um dos processos $\{p_i, p'_i\}$ verifica se a mensagem é válida e, em caso positivo, executa operação e cria a mensagem $\langle$REPLY, p_i, v, seq, c, res$\rangle_\sigma$, sendo *res* o resultado da operação executada, e insere na *postbox* (linha 15). Quando um processo lê a mensagem “REPLY” da *postbox*, compara os seus parâmetros e, se forem idênticos aos processados localmente, adiciona sua própria assinatura (linha 17).

Algoritmo 3 Algoritmo em operação normal: Tarefa 2 - postbox

1: **loop**
2: $msg \leftarrow postbox.read()$
3: **if** received (ORDER) **then**
4: **if** todos os parâmetros correspondem aos computados localmente **then**
5: **if** is the primary follower **then**
6: postbox.append($\langle\langle$ORDER, p_i, v, m.n, dm$\rangle_\sigma$, msg$\rangle$)
7: **end if**
8: sign(msg) ▷ Adiciona sua assinatura à mensagem.
9: **if** $|\sigma| > nVM/2$ **then**
10: multicast(msg)
11: **else if**
12: **then**postbox.append(msg)
13: **end if**
14: **end if**
15: **else if** received (REPLY) **then**
16: **if** todos os parâmetros correspondem aos computados localmente **then**
17: sign(msg) ▷ Adiciona sua assinatura à mensagem.
18: **if** $|\sigma| > nVM/2$ **then**
19: reply_to_client(msg)
20: **else if**
21: **then**postbox.append(msg)
22: **end if**
23: **end if**
24: **end if**
25: **end loop**

Caso a mensagem possua assinaturas suficientes para ser considerada válida, envia ao cliente (linha 19). Caso contrário insere novamente na *postbox* (linha 21).

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

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

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

No caso de faltas ocorridas nas réplicas backup, a execução do protocolo não sofre interferências, já que serão enviadas normalmente ao cliente as $f+1$ respostas esperadas. Este comportamento é ilustrado na Figura 10, em que pode-se perceber as réplicas $P2$ e $P4$ não enviando uma resposta baseada na requisição recebida. Esta ausência de envio

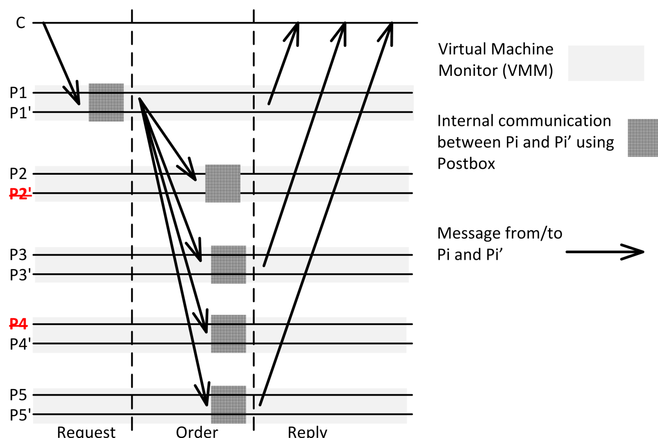

Figura 10: Passos de comunicação do protocolo em execução com duas falhas.

pode ter sido tanto porque a máquina física sofreu um problema de *crash* quanto porque não foi possível definir uma maioria de mensagens iguais entre os processos virtualizados.

4.3.5 Troca de Visão

A principal função do protocolo de troca de visão é manter o serviço progredindo mesmo na presença de um primário faltoso. Se o primário é faltoso, as réplicas backup nunca receberão uma mensagem "ORDER" válida e, portanto, devem definir um novo primário. Se um cliente não receber respostas suficientes em um tempo hábil, envia a requisição para todos os processos do sistema. Se uma réplica backup recebe uma requisição diretamente do cliente, verifica se já processou esta requisição e, em caso positivo, reenvia a resposta ao cliente. Caso contrário, encaminha a requisição aos processos da réplica primária e inicia um contador Δ_p, conforme apresentado no Algoritmo 2 (linhas 10-11).

Ao receber a mensagem "ORDER" correspondente do primário, o contador Δ_p é cancelado (linha 14) e o algoritmo continua normalmente. Se nenhuma mensagem "ORDER for recebida até o contador expirar, o processo inicia o protocolo de troca de visão - apresentado nos Algoritmos 4, 5 e 6 - inserindo na *postbox* a mensagem $\langle$VIEW-CHANGE, p_i, v+1, n, C, P$\rangle_\sigma$, sendo n o número de sequência do último *checkpoint* válido, C um conjunto composto por $f+1$ mensagens

"CHECKPOINT" garantindo o último *checkpoint*, e P um conjunto com todas as requisições processadas após o último *checkpoint* (linha 3). Se sua gêmea concordar com a troca de visão, a partir da mensagem "VIEW-CHANGE" lida da *postbox*, adiciona sua própria assinatura (linha 5). Caso a mensagem possua assinaturas suficientes para ser considerada uma mensagem válida, envia a mensagem para todos os processos (linha 7); caso contrário, insere novamente a mensagem na *postbox* (linha 9).

Algoritmo 4 Algoritmo de troca de visão: Tarefa 1 - rede

```
 1: loop
 2:     msg ← receive()
 3:     if received (VIEW-CHANGE) then
 4:         buffer ← buffer ∪ msg
 5:         if buffer contains at least f + 1 VIEW-CHANGE with n = msg.n ∧ d =
    msg.d then
 6:             envia mensagem aos primários
 7:             if i = msg.v mod |S| then
 8:                 sign(msg)                    ▷ Adiciona sua assinatura à mensagem.
 9:                 postbox.append(⟨NEW-VIEW, p_i, msg.v, V, P⟩_σ)
10:             end if
11:         end if
12:     else if received (NEW-VIEW) then
13:         buffer ← buffer ∪ msg
14:         for all req in msg.P do
15:             ensures req is processed and stored in the log.
16:         end for
17:     end if
18: end loop
```

Ao receber $f+1$ mensagens "VIEW-CHANGE" válidas de diferentes *hosts*, ambos os processos $\{p_i,\ p'_i\}$ no *host* h_i verificam se h_i é primário. Se sim, p aceita a troca de visão criando e inserindo na *postbox* a mensagem $\langle$NEW-VIEW, p_i, v+1, V, P$\rangle_{\sigma_{p_i}}$, sendo V um conjunto contendo as mensagens "VIEW-CHANGE" que originaram a troca de visão, e P um conjunto contendo as mensagens "ORDER" enviadas após o último *checkpoint* válido (linhas 5-9). Quando um processo p lê da *postbox* uma mensagem "NEW-VIEW", verifica se seus parâmetros são idênticos aos processados localmente e, caso sejam, adiciona sua assinatura (linha 14).Caso a mensagem possua a quantidade de assinaturas necessárias, envia a mensagem para todas as réplicas (linha 16); caso contrário, insere novamente a mensagem na *postbox* (linha 18).

Quando qualquer processo p recebe uma mensagem "NEW-VIEW" do novo primário, verifica se: (1) a mensagem está devidamente assi-

Algoritmo 5 Algoritmo de troca de visão: Tarefa 2 - postbox

```
1: loop
2:     msg ← postbox.read()
3:     if received (VIEW-CHANGE) then
4:         if all parameters corresponds the ones locally computed then
5:             sign(msg)                          ▷ Adiciona sua assinatura à mensagem.
6:             if |σ| > nVM/2 then
7:                 multi_send(msg)
8:             else if
9:                 thenpostbox.append(msg)
10:            end if
11:        end if
12:    else if received (NEW-VIEW) then
13:        if all parameters corresponds the ones locally computed then
14:            sign(msg)                          ▷ Adiciona sua assinatura à mensagem.
15:            if |σ| > nVM/2 then
16:                multi_send(msg)
17:            else if
18:                thenpostbox.append(msg)
19:            end if
20:        end if
21:    end if
22: end loop
```

Algoritmo 6 Algoritmo de troca de visão: Tarefa 3 - timeout

```
1: procedure TIMEOUT_EXPIRE                 ▷ Ao expirar o timeout Δp
2:     sign(msg)                              ▷ Adiciona sua assinatura à mensagem.
3:     postbox.append(⟨VIEW-CHANGE, pi, v+1, n, C, P⟨σ)
4: end procedure
```

nada, (2) contém um conjunto V com $f+1$ mensagens "VIEW-CHANGE" válidas. Se as condições forem satisfeitas, reexecuta todas as requisições contidas em P para a nova visão (linhas 12-15).

4.3.6 Coleta de Lixo

Para prevenir que ocorra um estouro na memória, é necessário um mecanismo que descarte as mensagens antigas armazenadas no *buffer*. Para isso, o algoritmo gera periodicamente um *checkpoint*, após algum número constante c de requisições recebidas. O algoritmo de coleta de lixo está apresentado nos Algoritmos 7, 8, 9. Ao chegar em um *checkpoint*, cada processo cria uma mensagem $\langle$CHECKPOINT, p_i, v, n, d$\rangle_\sigma$ (linha 2), sendo que n é o número da última requisição processada e d é um sumário do estado atual de p_i, e insere na *postbox*.

Cada máquina gêmea lerá a mensagem da *postbox* (linha 3) e assim que atingir o mesmo *checkpoint*, confirma se o estado recebido

Algoritmo 7 Algoritmo de coleta de lixo: Tarefa 1 - checkpoint

1: **procedure** CHECKPOINT ▷ Chamada a cada c mensagens processadas.
2: postbox.append(⟨CHECKPOINT, p_i, v, n, d⟩$_\sigma$)
3: **end procedure**

Algoritmo 8 Algoritmo de coleta de lixo: Tarefa 2 - postbox

1: **loop**
2: $msg \leftarrow receive()$
3: **if** all parameters corresponds the ones locally computed **then**
4: sign(msg) ▷ Adiciona sua assinatura à mensagem.
5: **end if**
6: **if** $|\sigma| > nVM/2$ **then**
7: multi_send(msg)
8: **else if**
9: **then**postbox.append(msg)
10: **end if**
11: **end loop**

Algoritmo 9 Algoritmo de coleta de lixo: Tarefa 3 - rede

1: **loop**
2: $msg \leftarrow receive()$
3: **if** received (CHECKPOINT) **then**
4: $buffer \leftarrow buffer \cup msg$
5: **if** buffer contains at least f+1 CHECKPOINT with $v = msg.v \wedge n = msg.n \wedge d = mdg.d$ **then**
6: remove all messages with n $\langle msg.n$
7: **end if**
8: **end if**
9: **end loop**

é o mesmo estado local e, em caso positivo, adiciona sua própria assinatura à mensagem "CHECKPOINT" (linha 4). Caso a quantidade de assinatura da mensagem seja maior do que $m/2$, ou seja, a mensagem está assinada por uma maioria de réplicas, envia para as outras réplicas (linha 7). Caso a quantidade de assinatura ainda não for suficiente, adiciona a mensagem novamente na *postbox* (linha 9). Quando um processo recebe $f+1$ mensagens "CHECKPOINT" corretamente assinadas de diferentes réplicas (linha 5) , para o mesmo número de visão v, o mesmo número de sequência n e o mesmo estado d, aceita como último *checkpoint* válido e remove do *buffer* todas as mensagens

com número de sequência menor do que n (linha 6).

Com o avanço do *checkpoint*, vão sendo atualizados o número mínimo h e máximo H de sequência que podem ser definidos pelo líder primário. A cada *checkpoint* efetuado, h é atualizado para o número de sequência da última requisição executada antes do *checkpoint* e H é atualizado de forma que $H = h + 2c$.

4.4 CORREÇÃO DO ALGORITMO

Na Seção 4.3.1 foi definido que o algoritmo deve apresentar as propriedade de Ordem Total (*safety*) e Terminação (*liveness*). A propriedade de Ordem Total exige que todas as requisições enviadas pelo cliente sejam executadas por todas as réplicas, na mesma ordem. A Terminação, por sua vez, exige que todas as requisições iniciadas pelo cliente devem ser executadas em algum momento pelo serviço.

Dados estes requisitos, serão declaradas algumas suposições feitas anteriormente que auxiliarão na definição das provas formais da proposta:

Axioma 1. *Apenas uma minoria de máquinas virtuais gêmeas pode falhar em qualquer momento da execução do serviço, com exceção para faltas de crash no host.*

Axioma 2. *Uma réplica física é considerada faltosa quando, para uma determinada mensagem, não é possível aferir uma maioria de mensagens idênticas entre as réplica gêmeas.*

Axioma 3. *Qualquer mensagem enviada sem a concordância de uma maioria de processos gêmeos, é considerada inválida e ignorada para todos os efeitos por todas as réplicas.*

Axioma 4. *Para cada f réplicas físicas faltosas, existe pelo menos $f + 1$ réplicas físicas corretas.*

Axioma 5. *Todas as máquinas gêmeas em uma máquina física possuem uma visão homogênea da postbox.*

Axioma 6. *A requisição será retransmitida pelo cliente periodicamente até o recebimento de uma resposta válida, conforme apresentado no Algoritmo 1.*

Axioma 7. *Sendo um modelo parcialmente síncrono, existe um tempo máximo Δ para entrega das mensagens na rede porém este tempo pode variar e não é conhecido.*

Axioma 8. *Não é possível a falsificação de mensagens por uma réplica maliciosa.*

Axioma 9. *O serviço executado pelas réplicas é determinístico e todas as réplicas iniciam o protocolo no mesmo estado.*

A partir de declaradas as hipóteses iniciais para o correto funcionamento do sistema, podemos apresentar as provas de correção.

Lema 1. *Toda requisição r correta enviada por algum cliente será, em algum momento, entregue, executada, e respondida pelas réplicas.*

Demonstração. Prova (esboço). Devido aos Axiomas 6 e 7, sabemos que a requisição r será recebida em algum momento por todas as réplicas. Caso a réplica primária não seja correta as réplicas backup iniciarão uma troca de visão até que a réplica atuando no papel de réplica primária seja uma réplica correta.

Desta forma, vemos que em algum momento uma réplica primária irá receber a requisição do cliente. Assim, conforme o Algoritmo 2 e 3, uma réplica primária correta irá difundir a requisição para as outras réplicas (linhas 6 do Algoritmo 2, 6 do Algoritmo 3 e 10 do Algoritmo 3) e todas as réplicas corretas irão executar a requisição em algum momento (linhas 15 e 19). Com isso, mostramos que toda requisição enviada por algum cliente será respondida pelas réplicas. □

Lema 2. *Se um processo correto qualquer p_i processa uma requisição r, então r será processada em todos os processos corretos na mesma ordem que p_i.*

Demonstração. Prova (esboço). Se um processo correto recebe uma requisição da réplica primária, esta sempre será considerada correta, caso contrário a requisição seria ignorada de acordo com o Axioma 3. Assim, todos os processos recebem a mesma definição sobre a ordem de execução das mensagens pois caso dois processos corretos recebam diferentes ordens para a mesma mensagem, esta condição estaria em contradição com os Axiomas 1, 2 e 3. □

Lema 3. *Se um processo correto p_i executa uma requisição r fornecendo uma resposta q, então pelo menos $f+1$ réplicas físicas fornecerão a mesma resposta q;*

Demonstração. Prova (esboço). Levando em consideração o Axioma 9 e o Lema 2, uma mesma requisição terá a mesma execução em todas as réplicas corretas que for executada. Se não houver pelo menos $f+1$ réplicas físicas corretas executando simultaneamente o protocolo, haverá uma contradição com o Axioma 4. □

Teorema 1. *As requisições enviadas por clientes são todas executadas pelas réplicas corretas em ordem total.*

Demonstração. Prova (esboço). Conforme o Lema 1, o Lema 2 e o Lema 3, todas requisições são recebidas e respondidas em algum momento pelas réplicas corretas, sendo executadas na mesma ordem por, pelo menos, $f+1$ réplicas físicas corretas. Assim, todas a requisições são executadas em ordem total por uma maioria de réplicas corretas, mantendo o estado do protocolo consistente. Assim, demonstramos que o algoritmo TwinBFT apresenta a propriedade ***safety***. □

Teorema 2. *O algoritmo TwinBFT sempre faz progresso.*

Demonstração. Prova (esboço). De acordo com o Axioma 7 temos que nem o serviço sendo executado nem a rede podem atrasar indefinidamente. Caso uma réplica backup maliciosa tente atrasar a resposta e deixe de enviar mensagens, o sistema ainda assim fará progresso com outra $f+1$ réplicas corretas, de acordo com o Lema 3. No caso de a réplica primária ser maliciosa e deixar de enviar mensagens, ainda assim haverão $f+1$ réplicas corretas solicitando uma troca de visão, de forma a continuar o progresso, de acordo com o Lema 1. Assim, demonstramos que o algoritmo TwinBFT apresenta a propriedade ***liveness***. □

4.5 CONSIDERAÇÕES FINAIS

Neste capítulo foi introduzido o algoritmo TwinBFT como uma técnica de replicação de máquina de estados tolerante a intrusões. A partir do uso de virtualização e ideias de detectores de falhas, o TwinBFT oferece uma abordagem de tolerância a faltas bizantinas com resiliência de $f = \frac{N-1}{2}$ tendo uma vantagem sobre abordagens tradicionais que costumam apresentar uma resiliência de $f = \frac{N-1}{3}$, considerando N como o número de máquinas físicas necessárias para a tolerância de f faltas.

Outras abordagens atualmente possuem a mesma resiliência, geralmente obtida através da utilização de um componente confiável. De certa forma, o uso de virtualização para o espelhamento de máquinas virtuais do serviço, ou máquinas gêmeas, substitui este componente confiável pelo uso da virtualização que é uma tecnologia bastante acessível. Assim, a abordagem essencialmente transforma faltas bizantinas em faltas de omissão.

5 RESULTADOS EXPERIMENTAIS E ANÁLISE

5.1 AMBIENTE DE TESTES

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

Os experimentos foram executados em três servidores com processador Intel®Core^{TM}i7 3.8Ghz com Debian 7.0 “wheezy” (Kernel 3.2.0 x86-64) e VMM Xen Hypervisor 4.1.3. Cada máquina virtual foi configurada com 2GB de memória e duas unidades de processamento virtuais, equipadas com SUN’s JDK 1.6.0_29.

A rede utilizada para a comunicação entre as réplicas físicas e os clientes foi uma rede local LAN de 100Mbps.

A arquitetura de sistema utilizada empregou duas máquinas virtuais em cada máquina física, cada uma com um processo do sistema replicado. A memória compartilhada ficou presente em um diretório compartilhado pelo sistema operacional hospedeiro, chamado no XEN de *Domain0*. A arquitetura, conforme implementada pode ser visualizada na Figura 11.

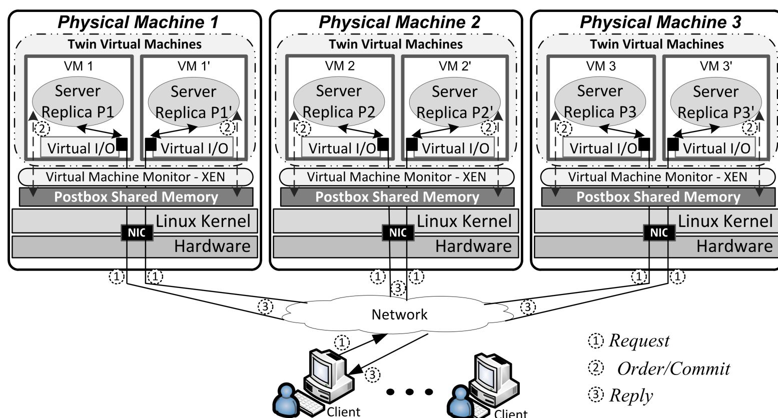

Figura 11: Arquitetura implementada para análise de desempenho.

5.2 MEMÓRIA COMPARTILHADA

A implementação da memória compartilhada é de suma importância para o algoritmo. A utilização de técnicas de virtualização fornece diversos meios para a implementação de uma memória compartilhada entre as máquinas virtuais. A alternativa mais óbvia está no compartilhamento de um diretório localizado no host. Em alguns sistemas operacionais, como o GNU Linux, este diretório pode inclusive ser mapeado diretamente em memória utilizando alguma técnica como o sistema de arquivos *tmpfs*.

Algumas técnicas para passagem de mensagem específica entre máquinas virtuais foram propostas. Um exemplo é a abordagem proposta por Diakhaté et al. (2009) que cria uma forma eficiente para passagem de mensagem usando *buffers* compartilhados em memória no KVM(DIAKHATé et al., 2009). Outra proposta semelhante, voltada à passagem de mensagens no *XEN Hypervisor*, é o *XenSocket* que oferece uma abstração da camada de comunicação de rede para a comunicação entre domínios sem a necessidade das mensagens passarem pela pilha de rede (ZHANG et al., 2007).

Atualmente, alguns bancos de dados oferecem um armazenamento em memória, como é o caso do Memcached (FITZPATRICK, 2004). Com este tipo de banco de dados é possível a implementação da memória compartilhada executando este servidor no hospedeiro, e acessando as tuplas a partir das máquinas virtuais. Como essa abordagem utiliza a rede, entretanto, é necessário que sejam configuradas interfaces de rede virtuais para a comunicação entre as máquinas virtuais e o hospedeiro sem causar um impacto no desempenho ou na segurança.

Como a implementação desta memória compartilhada não é o foco principal deste trabalho, foi utilizada uma técnica de armazenamento em arquivos compartilhados em memória. Para substituir uma possível implementação manual da comunicação foi utilizada a biblioteca *Java Chronicle*(LAWREY, 2012) focada na comunicação entre processos com baixa latência. Em teste realizado pelo autor da ferramenta, foram obtidas latências de até 80ns na gravação de uma mensagem. Em testes preliminares com uma mensagem típica trocada no protocolo TwinBFT, entretanto, a transferência de uma mensagem obteve uma latência média na casa de 100 μs.

A implementação se deu pela criação de dois canais entre cada par de máquinas virtuais em um host. Cada processo na máquina virtual possui permissão para adicionar dados em um destes canais e ler os dados do outro. Assim, a comunicação entre cada par de máquina

virtual ocorreu com uma baixa latência.

5.3 CONSIDERAÇÕES GERAIS

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

De forma a avaliar a sua performance, o algoritmo foi executado em condições normais. Foram enviadas dez mil requisições de um único cliente, em três diferentes cargas: 0/0, 0/4 e 4/0. Eles representam, respectivamente, uma requisição e resposta nula, uma requisição nula e uma resposta de 4KB, e uma requisição de 4KB e resposta nula. Todos os tempos foram medidos pelo cliente, a partir da leitura de seu relógio local antes do envio da requisição e após o recebimento de uma resposta válida.

5.4 RESULTADOS E ANÁLISE

As primeiras avaliações do algoritmo foram feitas com base na medição de sua latência e *throughput* na execução de dez mil requisições sendo enviadas por um único cliente em diversas condições.

Para a medição da latência, uma sequência de dez mil requisições foram enviadas a partir de um único cliente com um atraso entre os envios suficiente para garantir o processamento da cada requisição individualmente nas réplicas. Esta abordagem visa medir o tempo de processamento das requisições em um caso ótimo, sem interferência entre requisições e sem a ocorrência de falhas.

Na Figura 12, são mostradas as diferentes latências em cada carga. Para este teste, foram executadas três baterias de requisições sendo cada uma delas com uma carga diferente. Com isso é apre-

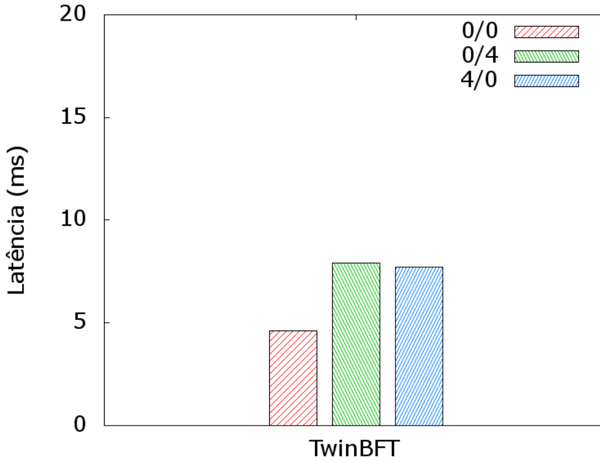

Figura 12: Latência em execução normal.

sentado o impacto no tempo de resposta causado pela mudança no tamanho das mensagens. Este gráfico mostra basicamente o melhor tempo possível para a execução de uma requisição no protocolo. Como todas as medições são feitas pelo cliente, está incluída neste tempo a comunicação entre o cliente e as réplicas. Esta métrica, entretanto, não apresenta nenhuma noção sobre o funcionamento dos servidores em situações com uma maior carga.

O *throughput* mede a capacidade do servidor em atender as requisições simultaneamente, medindo quantas requisições conseguem ser atendidas pelo servidor em uma determinada unidade de tempo. A Figura 13 apresenta a quantidade de requisições atendidas por segundo em uma execução com dez mil requisições sendo enviadas pelo cliente sem nenhum tempo de atraso entre elas.

No primeiro grupo, são apresentadas as medidas para o serviço em seu caso normal, sem falhas. Logo em seguida, em TwinBFT-f, é apresentado o *throughput* do algoritmo em um caso de faltas, sendo que cada requisição enviada possuía 1% de chance de ser recusada pelas réplicas primárias, forçando assim uma troca de visão para que seja mantida a consistência.

O caso de testes TwinBFT-f não foi utilizado para uma medição de latência pois as requisições faltosas apenas alterariam levemente o tempo médio de resposta como pontos fora da curva. Adiante será apresentada uma avaliação mais detalhada de tempos de resposta em casos também de falhas. Serão também apresentados casos de *through-*

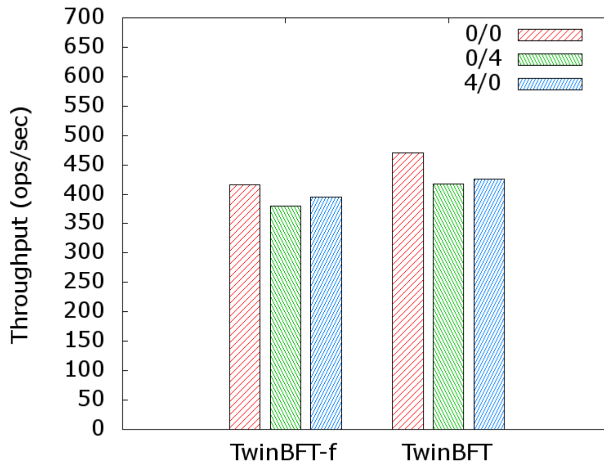

Figura 13: Comparação de throughput em execução normal.

put com múltiplos clientes para um melhor apreciação da performance do algoritmo com uma carga superior de requisições.

5.5 EXECUÇÃO NORMAL

Para demonstrar o funcionamento do algoritmo de uma forma mais detalhada, efetuamos medições sobre uma execução otimista do protocolo, em um cenário livre de faltas. Foram executadas quinhentas requisições pelo cliente, de forma sequencial, com um atraso entre as requisições.

Na Figura 14 é possível observar que a maior parte das requisições se encontra abaixo de 10ms porém algumas chegam próximas ao dobro disso. Esta variação pode se dar por atrasos na rede, ou outras variáveis no processamento da execução. Mesmo assim, podemos perceber na Figura 15 que a grande maioria das requisições são executadas na casa de quatro e cinco milissegundos.

5.5.1 Múltiplos Clientes

De forma a avaliar a performance do protocolo sob situações de alta carga, foi feita uma análise utilizando vários clientes enviando requisições simultamente. Desta forma, pode-se perceber o volume máx-

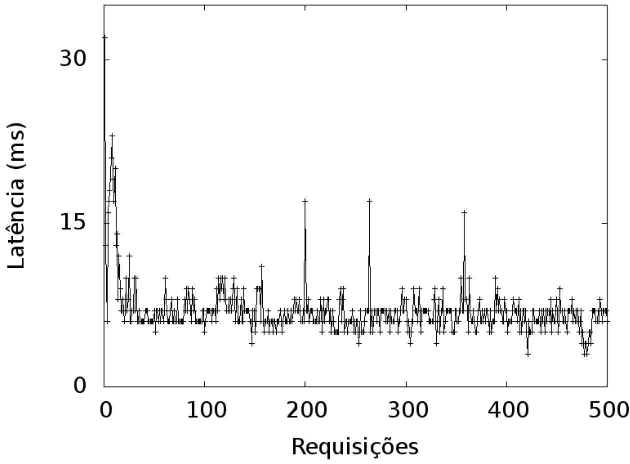

Figura 14: Histograma da latência em execução normal.

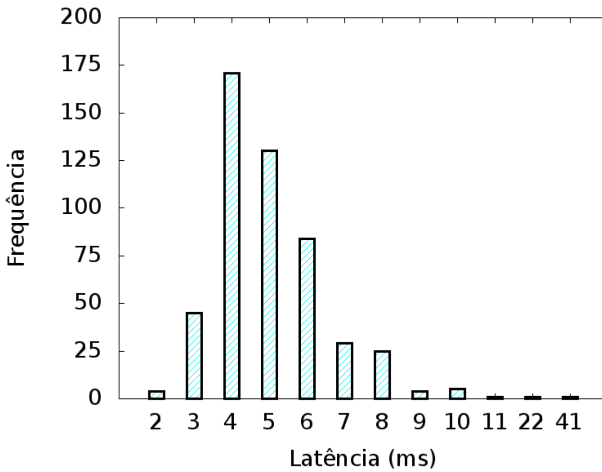

Figura 15: Distribuição da latência em execução normal.

imo de requisições atendidas pelo protocolo.

A Figura 16 apresenta o *throughput* da execução desde um único cliente enviando requisições até vinte clientes enviando as requisições simultaneamente.

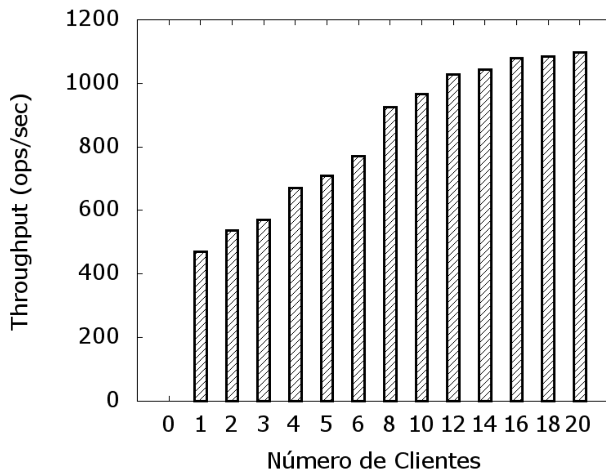

Figura 16: *Throughput* em execução normal com múltiplos clientes.

5.6 EXECUÇÃO COM FALHAS

Sendo um protocolo voltado à tolerância a faltas, se faz necessária uma análise da performance obtidas em casos em que ocorrem falhas na execução de parte das requisições. Levando em consideração que este é um serviço replicado tolerante a $2f+1$ faltas em máquinas físicas, podemos verificar que, na arquitetura de testes utilizada, podemos tolerar, no máximo, uma réplica faltosa.

Devido à natureza do protocolo, qualquer réplica *backup* faltosa não apresentará impacto no tempo de resposta das requisições do cliente, já que este considera válida uma resposta que foi recebida de $f+1$ réplicas. Ou seja, mesmo se houver uma réplica *backup* faltosa entre as réplicas, esta não influenciará em nada na verificação da performance do protocolo. Por este motivo, todas as faltas apresentadas neste estudo ocorreram em uma das máquinas virtuais da réplica primária.

Ao receber uma requisição do cliente, a réplica primária aplica uma probabilidade específica para aceitar a requisição e continuar com o protocolo ou simular uma falta e não dar procedimento à requisição. Com isso, o cliente reenvia a requisição para as outras réplicas que induzem à uma troca de visão. A resposta só será enviada ao cliente quando a troca de visão for encerrada e um novo líder oferecer continuidade ao protocolo.

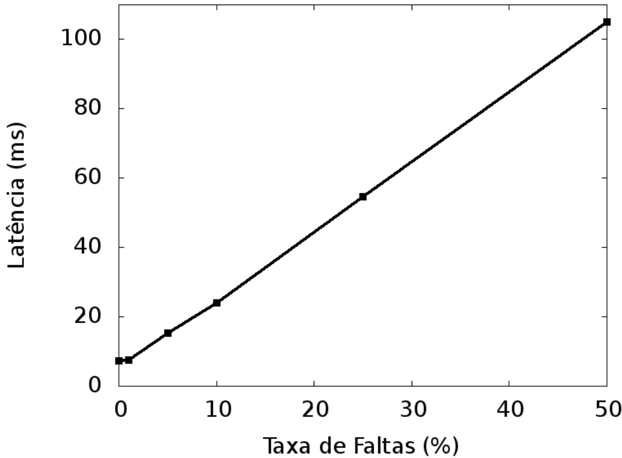

Figura 17: Impacto das falhas na latência.

Para efeitos de continuidade mesmo com mais faltas do que o número de réplicas, em momentos distintos, assim que a réplica deixa de ser primária ela volta a responder normalmente o protocolo, permitindo assim que a nova réplica primária sofra uma falta sem infringir a premissa de f réplicas faltosas.

De uma forma geral, é comparada a latência média das requisições na Figura 17. Neste cenários foram executadas dez mil requisições, em diferentes probabilidades de falhas. Assim, pode-se perceber o crescimento do tempo de resposta conforme a taxa de faltas cresce no protocolo.

5.6.1 Execução com 1% de Faltas

Com a probabilidade de 1% das requisições falharem ao serem recebidas pela réplica primária, foi feita uma amostragem do tempo de respostas, conforme detalhado na Figura 18 e distribuído na Figura 19. Para esta medição foi executado o protocolo com quinhentas requisições sendo enviadas sequencialmente a partir do cliente, com tempo suficiente entre as requisições para que o processamento de cada uma ocorresse sem interferência, mesmo em casos faltosos. Isso explica a consistência nos tempos de resposta das requisições corretas e das requisições com falta.

O tempo de resposta das requisições com falta ficam bastante

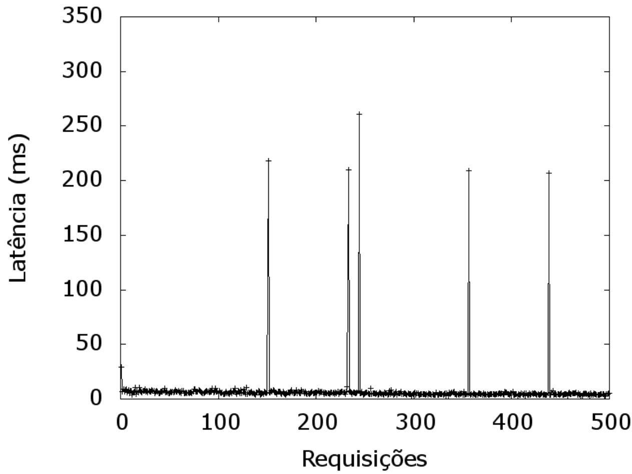

Figura 18: Histograma da latência com 1% de faltas.

visíveis, pois levam cerca de vinte vezes mais tempo do que as requisições sem faltas. Este tempo se dá principalmente pelo tempo necessário ao *timeout* que lança a troca da visão. Este *timeout*, apesar de fixo nestes experimentos pode ser ajustado dinamicamente ao longo do funcionamento do protocolo para um melhor desempenho.

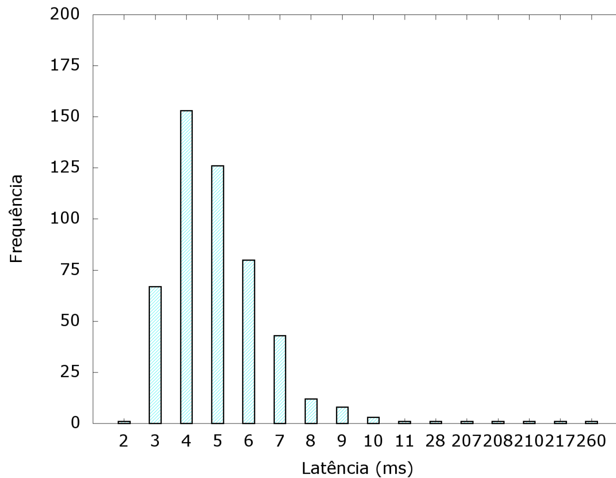

Figura 19: Distribuição da latência com 1% de faltas.

5.6.2 Execução com 5% de Faltas

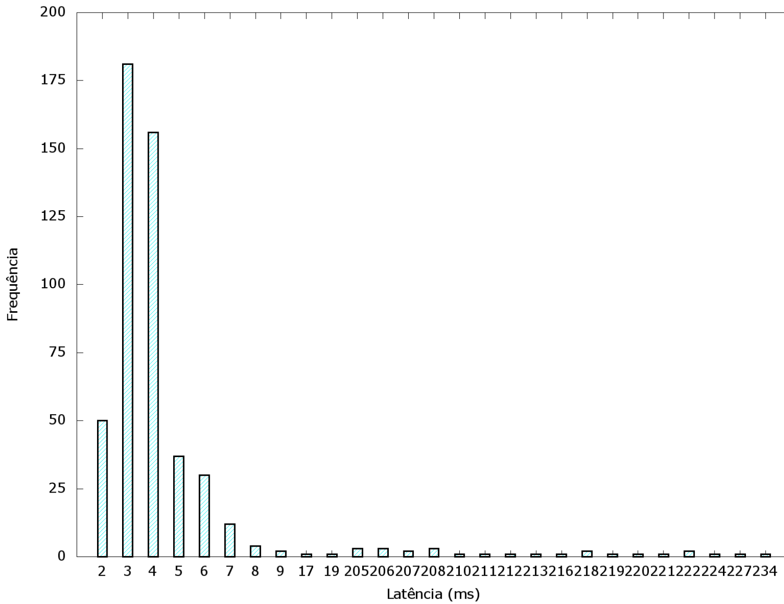

Figura 20: Distribuição da latência com 5% de faltas.

Com 5% de faltas, temos um cenário em que podemos extrapolar o impacto das faltas no tempo de resposta. Conforme apresentado na Figura 17, a partir de 5% de chance de uma mensagem ocasionar uma troca de visão, já temos um impacto bastante significativo no tempo de resposta médio. Este comportamento pode ser visualizado em detalhes nas Figuras 21 e 20.

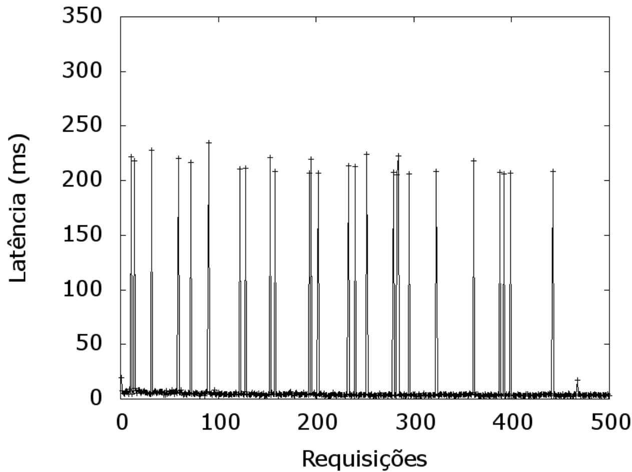

Figura 21: Histograma da latência com 5% de faltas.

5.7 COMPARAÇÃO QUALITATIVA COM OUTRAS ABORDAGENS

Tabela 2: Comparação Entre Propriedades dos Protocolos Avaliados

	Número réplicas	**Número processos**	**Número físicas**	**Passos**	**Especulativo Otimista**
PBFT (CASTRO; LISKOV, 1999b)	$3f+1$	$3f+1$	$3f+1$	5	não
Zyzzyva (KOTLA et al., 2008)	$3f+1$	$3f+1$	$3f+1$	3 / 5[1]	sim
Yin (YIN et al., 2003)	$2f+1$	$3f+1$	$3f+1$	4	não
BFT-TO (CORREIA; NEVES; VERÍSSIMO, 2013)	$2f+1$	$2f+1$	$2f+1$	5	não
A2M-PBFT-EA (CHUN et al., 2007)	$2f+1$	$2f+1$	$2f+1$	5	não
MinBFT (VERONESE et al., 2013)	$2f+1$	$2f+1$	$2f+1$	4	não
MinZyzzyva (VERONESE et al., 2013)	$2f+1$	$2f+1$	$2f+1$	3 / 5[1]	sim
TwinBFT	$2f+1$	$4f+2$	$2f+1$	3	não

Na Tabela 2 é mostrada uma comparação entre a abordagem proposta e outros algoritmos BFT na literatura. Todos os número consideram apenas execuções sem faltas. Os benefícios no uso de máquinas virtuais gêmeas são visíveis no número de réplicas e passos de comuni-

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

cação. Enquanto a abordagem proposta tem o menor número de réplicas, juntamente com (CORREIA; NEVES; VERISSIMO, 2004; CHUN et al., 2007; VERONESE et al., 2013), ela tem o mesmo número de passos de comunicação de algoritmos especulativos (KOTLA et al., 2008; VERONESE et al., 2013), mesmo em caso de faltas. Algoritmos especulativos, entretanto requerem um número maior de passos em casos com faltas, além de envolverem o cliente no protocolo, perdendo transparência.

Como a abordagem proposta utiliza duas máquinas virtuais em cada réplica, possui um número maior de processos, apesar do número de máquinas físicas ser o mesmo de (CORREIA; NEVES; VERISSIMO, 2004; CHUN et al., 2007; VERONESE et al., 2013).

5.8 CONSIDERAÇÕES FINAIS

Neste capítulo foi apresentado um estudo sobre a performance do algoritmo em um ambiente real. Para este fim, um serviço foi implementado e replicado de forma a cumprir os requisitos do protocolo. Com esta execução foi possível analisar o comportamento do sistema sob uma ótica mais prática.

O objetivo básico do TwinBFT é servir como uma arquitetura de tolerância a faltas bizantinas a partir da utilização de ambientes replicados. Esta arquitetura deve apresentar uma execução ordenada das requisições. A execução do protocolo foi efetuada sob diversos cenários com o intuito de simular situações práticas sempre pensando na tolerância a intrusões. Com isso foi possível observar o aumento gradual da latência em resposta a diferentes probabilidades de faltas.

Também foram apresentadas métricas em casos mais complexos, como a execução do protocolo sob a presença de múltiplos clientes. Com estes números foi possível analisar uma tendência de estabilização indicando o máximo de requisições que podem ser servidas pelo protocolo nas configurações utilizadas. Testes efetuados também com mensagens de diferentes tamanhos mostraram a variação no tempo de resposta em detrimento da quantidade de dados trafegada pelo protocolo.

6 CONCLUSÕES E PERSPECTIVAS FUTURAS

O presente documento apresentou em detalhes uma arquitetura proposta com o objetivo de fornecer tolerância a faltas bizantinas a partir da replicação de máquina de estados. A abordagem proposta utiliza técnicas de virtualização para espelhar o serviço de forma a detectar e ignorar réplicas faltosas, transformando assim faltas bizantinas, ou intrusões, em faltas de omissão. Essa transformação possibilitou a redução $N = 3f + 1$ para $N = 2f + 1$ no que N se refere à quantidade de máquinas físicas para o correto funcionamento do protocolo, além de possibilitar a um protocolo simples, com uma quantidade menor de passos de comunicação em cada requisição.

A partir disto foi desenvolvida uma arquitetura segundo o modelo de sistema apresentado na Seção 4.1 de forma a satisfazer as propriedades de funcionamento apresentadas na Seção 4.3.1. Isso foi possível a partir do aproveitamento e da união de conceitos apresentados em outras abordagens na literatura como: *PBFT* (CASTRO; LISKOV, 1999a) e Replicação de Máquina de Estados (SCHNEIDER, 1990), a utilização de virtualização para uma memória compartilhada (STUMM et al., 2010), as noções de detectores de falhas (MALKHI; REITER, 1997; KIHLSTROM; MOSER; MELLIAR-SMITH, 2003; HAEBERLEN; KOUZNETSOV; DRUSCHEL, 2006; DOUDOU; GARBINATO; GUERRAOUI, 2005), e a utilização de componentes confiáveis para a redução do número de réplicas na abordagem (CORREIA; NEVES; VERISSIMO, 2004; VERONESE et al., 2013).

Assim, o trabalho apresentado cumpriu todos os objetivos estipulados na Seção 1.3. Uma revisão bibliográfica, apresentada no Capítulo 2 foi efetuada para formar uma base teórica para a elaboração deste trabalho, a partir do estudo de conceitos como: confiança de funcionamento, sistemas distribuídos e técnicas de virtualização, e técnicas de deteceção de falhas. Uma análise do estado da arte foi apresentada no Capítulo 3, a partir da síntese de alguns dos trabalhos mais relacionados com a abordagem proposta.

Os detalhes da arquitetura proposta, bem como o algoritmo, seu modelo de sistema e suas provas de correção foram apresentados no Capítulo 4. Neste ponto foram apresentados todos os detalhes presentes na arquitetura como o modelo de falhas, as técnicas de troca de visão e *checkpoint* utilizadas, as deficiências do protocolo, etc.

Cumprindo os objetivos definidos na Seção 1.3.2 o algoritmo foi implementado e seu funcionamento analisado em diversos cenários de

uso, feito assim um levantamento sobre seu desempenho tanto em casos de funcionamento sem faltas quanto em casos em que ocorrem faltas maliciosas. Este levantamento, apresentado em detalhes no Capítulo 5, mostrou que a abordagem proposta continua sendo executada corretamente mesno na presença de faltas e possibilitou um melhor entendimento do algoritmo proposto. Também foi possível verificar com esta análise o impacto causado pelas faltas no desempenho do algoritmo.

No decorrer da elaboração deste trabalho foram identificadas oportunidades que podem exploradas futuramente, representando possíveis trabalhos futuros. Estas oportunidades seguem listadas abaixo:

- Implementaçao e comparação prática da abordagem proposta com outras abordagens da literatura, utilizando uma arquitetura padrão.
- Testes da abordagem com diferentes gerenciadores de máquinas virtuais, e com diversidade de software pois apesar da independência teórica de VMM, todos os testes práticos foram efetuados apenas utilizando o *XEN*.
- Analisar possibilidades do isolamento do serviço replicado de forma a espelhar apenas a parte funcional da arquitetura, diminuindo assim o ônus da utilização de máquinas gêmeas para a replicação de serviços complexos.
- Estudo sobre o ônus sobre o desempenho em relação à escalabilidade do sistema.

Além deste documento, o desenvolvimento deste trabalho resultou na publicação de três artigos em eventos e periódicos da área:

- *"Replicação por Máquina de Estados Tolerante a Faltas Bizantinas usando Máquinas Virtuais Gêmeas"* - XXXI Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos - SBRC 2013 (DETTONI et al., 2013b).
- *"Byzantine Fault-Tolerant State Machine Replication with Twin Virtual Machines"* - 18th IEEE Symposium on Computers and Communications - IEEE ISCC 2013 (DETTONI et al., 2013a).
- *"Using Virtualization Technology for Fault-Tolerant Replication in LAN"* - 8th International Conference on Dependability and Complex Systems - publicado no periódico "Advances in Intelligent and Soft Computing" - 2013 (DETTONI; LUNG; LUIZ, 2013).

REFERÊNCIAS BIBLIOGRÁFICAS

ATTIYA, H.; WELCH, J. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. [S.l.]: John Wiley & Sons, 2004. ISBN 0471453242.

AVIZIENIS, A. et al. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, v. 1, n. 1, p. 11–33, 2004. ISSN 1545-5971.

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

BESSANI, A. et al. Enhancing fault / intrusion tolerance through design and configuration diversity. In: *Proceedings of the 3rd Workshop on Recent Advances on Intrusion-Tolerant Systems*. [S.l.: s.n.], 2009.

BUDHIRAJA, N. et al. Distributed systems (2nd ed.). In: MULLENDER, S. (Ed.). New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1993. cap. The primary-backup approach, p. 199–216. ISBN 0-201-62427-3. <http://dl.acm.org/citation.cfm?id=302430.302438>.

CASTRO, M.; LISKOV, B. *Authenticated Byzantine Fault Tolerance without Public-Key Cryptography*. 1999.

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

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

CHARRON-BOST, B. Agreement problems in fault-tolerant distributed systems. In: *Proceedings of the 28th Conference on Current Trends in Theory and Practice of Informatics Piestany:*

Theory and Practice of Informatics. London, UK, UK: Springer-Verlag, 2001. (SOFSEM '01), p. 10–32. ISBN 3-540-42912-3. <http://dl.acm.org/citation.cfm?id=647011.712835>.

CHUN, B.-G. et al. Attested append-only memory: making adversaries stick to their word. In: *Proceedings of the 21st ACM Symposium on Operating Systems Principles*. [S.l.: s.n.], 2007. p. 189–204.

CORREIA, M.; NEVES, N. F.; VERISSIMO, P. How to tolerate half less one Byzantine nodes in practical distributed systems. In: *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems*. [S.l.: s.n.], 2004. p. 174–183. ISBN 0-7695-2239-4.

CORREIA, M.; NEVES, N. F.; VERÍSSIMO, P. Bft-to: Intrusion tolerance with less replicas. *Comput. J.*, v. 56, n. 6, p. 693–715, 2013.

COULOURIS, G. et al. *Distributed Systems: Concepts and Design*. 5th. ed. USA: Addison-Wesley Publishing Company, 2011. ISBN 0132143011, 9780132143011.

DETTONI, F. et al. Byzantine fault-tolerant state machine replication with twin virtual machines. In: *18th IEEE Symposium on Computers and Communications (IEEE ISCC 2013)*. [S.l.: s.n.], 2013.

DETTONI, F. et al. Replicação por máquina de estados tolerante a faltas bizantinas usando máquinas virtuais gêmeas. In: *XXXI Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*. Brasília, DF, Brasil: [s.n.], 2013.

DETTONI, F.; LUNG, L. C.; LUIZ, A. F. Using virtualization technology for fault-tolerant replication in lan. In: *8th International Conference on Dependability and Complex Systems*. [S.l.: s.n.], 2013.

DIAKHATé, F. et al. Efficient shared memory message passing for inter-vm communications. In: CéSAR, E. et al. (Ed.). *Euro-Par 2008 Workshops - Parallel Processing*. [S.l.]: Springer Berlin Heidelberg, 2009, (Lecture Notes in Computer Science, v. 5415). p. 53–62. ISBN 978-3-642-00954-9.

DISTLER, T. et al. SPARE: Replicas on hold. In: *Proceedings of the 18th Network and Distributed System Security Symposium*. [S.l.: s.n.], 2011. p. 407–420.

DOLEV, D.; DWORK, C.; STOCKMEYER, L. On the minimal synchronism needed for distributed consensus. *J. ACM*, ACM, New

York, NY, USA, v. 34, n. 1, p. 77–97, jan. 1987. ISSN 0004-5411. <http://doi.acm.org/10.1145/7531.7533>.

DOUDOU, A.; GARBINATO, B.; GUERRAOUI, R. Encapsulating failure detection: From crash to byzantine failures. In: *Proceedings of the 7th Ada-Europe International Conference on Reliable Software Technologies*. London, UK, UK: Springer-Verlag, 2002. (Ada-Europe '02), p. 24–50. ISBN 3-540-43784-3. <http://dl.acm.org/citation.cfm?id=646581.697916>.

DOUDOU, A.; GARBINATO, B.; GUERRAOUI, R. Tolerating arbitrary failures with state machine replication. In: *Dependable Computing Systems: Paradigms, Performance Issues, and Applications*. [S.l.]: Wiley, 2005. p. 27–56.

DOUDOU, A. et al. Muteness failure detectors: Specification and implementation. In: *Proceedings of the Third European Dependable Computing Conference on Dependable Computing*. London, UK: Springer-Verlag, 1999. (EDCC-3), p. 71–87. ISBN 3-540-66483-1. <http://dl.acm.org/citation.cfm?id=645332.649834>.

FITZPATRICK, B. Distributed caching with memcached. *Linux J.*, Belltown Media, Houston, TX, v. 2004, n. 124, p. 5–, ago. 2004. ISSN 1075-3583. <http://dl.acm.org/citation.cfm?id=1012889.1012894>.

FRASER, K. et al. Safe hardware access with the xen virtual machine monitor. In: *In 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS*. [S.l.: s.n.], 2004.

GARCIA, M. et al. OS diversity for intrusion tolerance: Myth or reality? In: *Proceedings of the IEEE/IFIP 41st International Conference on Dependable Systems and Networks*. [S.l.: s.n.], 2011. p. 383 –394.

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

GASHI, I.; POPOV, P. T.; STRIGINI, L. Fault tolerance via diversity for off-the-shelf products: A study with SQL database servers. *IEEE Transactions on Dependable and Secure Computing*, v. 4, n. 4, p. 280–294, 2007.

GUERRAOUI, R.; SCHIPER, A. Software-based replication for fault tolerance. *Computer*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 30, n. 4, p. 68–74, abr. 1997. ISSN 0018-9162. <http://dx.doi.org/10.1109/2.585156>.

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

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

JAIN, R. K. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. [S.l.]: John Wiley & Sons, 1991. ISBN 0471503363.

JIANG, X.; WANG, X. Out-of-the-box monitoring of VM-based high-interaction honeypots. In: *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection*. [S.l.: s.n.], 2007.

KIHLSTROM, K. P.; MOSER, L. E.; MELLIAR-SMITH, P. M. . Byzantine fault detectors for solving consensus. *The Computer Journal*, v. 46, p. 2003, 2003.

KIHLSTROM, K. P.; MOSER, L. E.; MELLIAR-SMITH, P. M. Solving consensus in a byzantine environment using an unreliable fault detector. In: *In Proceedings of the International Conference on Principles of Distributed Systems (OPODIS*. [S.l.: s.n.], 1997. p. 61–75.

KOTLA, R. et al. Zyzzyva: speculative byzantine fault tolerance. *Commun. ACM*, ACM, New York, NY, USA, v. 51, p. 86–95, November 2008. ISSN 0001-0782. <http://doi.acm.org/10.1145/1400214.1400236>.

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

LAPRIE, J.-C. Dependable computing and fault tolerance : Concepts and terminology. In: *Fault-Tolerant Computing, 1995, Highlights from*

Twenty-Five Years., Twenty-Fifth International Symposium on. [S.l.: s.n.], 1995. p. 2–.

LAUREANO, M.; MAZIERO, C.; JAMHOUR, E. Intrusion detection in virtual machine environments. In: *Proceedings of the 30th Euromicro Conference.* [S.l.: s.n.], 2004. p. 520–525.

LAWREY, P. *Java Chronicle.* 2012. <https://github.com/peter-lawrey/Java-Chronicle>.

LYNCH, N. A. *Distributed Algorithms.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996. ISBN 1558603484.

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

MURRAY, D. G.; MILOS, G.; HAND, S. Improving Xen security through disaggregation. In: *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments.* [S.l.: s.n.], 2008. p. 151–160.

NANDA, S.; CHIUEH, T. cker. *A survey of virtualization technologies.* [S.l.], 2005.

POLEDNA, S. Replica determinism in distributed real-time systems: a brief survey. *Real-Time Syst.*, Kluwer Academic Publishers, Norwell, MA, USA, v. 6, n. 3, p. 289–316, maio 1994. ISSN 0922-6443. <http://dx.doi.org/10.1007/BF01088629>.

REISER, H. P.; KAPITZA, R. VM-FIT: supporting intrusion tolerance with virtualisation technology. In: *Proceedings of the 1st Workshop on Recent Advances on Intrusion-Tolerant Systems (in conjunction with Eurosys 2007, Lisbon, Portugal, March 23, 2007).* [S.l.: s.n.], 2007. p. 18–22.

REISER, H. P.; KAPITZA, R. Fault and intrusion tolerance on the basis of virtual machines. In: *Tagungsband des 1. Fachgespräch Virtualisierung (Feb 11-12, 2008, Paderborn, Germany).* [S.l.: s.n.], 2008.

ROSENBLUM, M. The reincarnation of virtual machines. *Queue*, ACM, New York, NY, USA, v. 2, n. 5, p. 34–40, jul. 2004. ISSN 1542-7730. <http://doi.acm.org/10.1145/1016998.1017000>.

SAHOO, J.; MOHAPATRA, S.; LATH, R. Virtualization: A survey on concepts, taxonomy and associated security issues. In: *Computer and Network Technology (ICCNT), 2010 Second International Conference on*. [S.l.: s.n.], 2010. p. 222–226.

SCHNEIDER, F. B. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 22, n. 4, p. 299–319, dez. 1990. ISSN 0360-0300.

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

SZEFER, J. et al. Eliminating the hypervisor attack surface for a more secure cloud. In: *Proceedings of the 18th ACM Conference on Computer and Communications Security*. [S.l.: s.n.], 2011. p. 401–412.

TSUDIK, G. Message authentication with one-way hash functions. *SIGCOMM Comput. Commun. Rev.*, ACM, New York, NY, USA, v. 22, n. 5, p. 29–38, out. 1992. ISSN 0146-4833.

VERONESE, G. S. et al. Efficient Byzantine fault tolerance. *IEEE Transactions on Computers*, v. 62, n. 1, p. 16–30, 2013.

WANG, Z.; JIANG, X. HyperSafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In: *Proceedings of the IEEE Security and Privacy Symposium*. [S.l.: s.n.], 2010. p. 380–395.

WOOD, T. et al. ZZ and the art of practical BFT execution. In: *Proceedings of the 6th ACM SIGOPS/EuroSys European Systems Conference*. [S.l.: s.n.], 2011. p. 123–138.

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

ZHANG, X. et al. *XenSocket: A high-throughput interdomain transport for VMs*. [S.l.], 2007.