View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Repositório Institucional da UFSC

UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

Marcelo Ribeiro Xavier da Silva

TOLERÂNCIA A FALTAS BIZANTINAS ATRAVÉS DE HIBRIDIZAÇÃO DO SISTEMA DISTRIBUÍDO

Florianópolis

2013

Marcelo Ribeiro Xavier da Silva

TOLERÂNCIA A FALTAS BIZANTINAS ATRAVÉS DE HIBRIDIZAÇÃO DO SISTEMA DISTRIBUÍDO

Monografia submetida ao Programa de Pós-Graduação em Ciências da Computação para a obtenção do Grau de Mestre em Ciências da Computação.
Orientador: Prof. Dr. Lau Cheuk Lung
Candidato: Marcelo Ribeiro Xavier da Silva

Florianópolis

2013

Catalogação na fonte elaborada pela biblioteca da
Universidade Federal de Santa Catarina

A ficha catalográfica é confeccionada pela Biblioteca Central.

Tamanho: 7cm x 12 cm

Fonte: Times New Roman 9,5

Maiores informações em:
http://www.bu.ufsc.br/design/Catalogacao.html

RESUMO

A ocorrência de faltas e falhas nos sistemas computacionais pode levar a catástrofes e prejuízos humanos, estruturais e financeiros. Recentemente, as faltas em sistemas computacionais têm aparecido mais frequentemente sob a forma de intrusões, que são o resultado de um ataque que obtém sucesso ao explorar uma ou mais vulnerabilidades. Uma questão recorrente é a discussão de quanto podemos confiar no funcionamento destes sistemas, demonstrando a necessidade de uma melhor aplicação de conceitos como dependabilidade, onde é esperado que o sistema funcione conforme suas especificações, ainda que alguns componentes apresentem problemas.

Replicação de Máquina de Estados é uma técnica comumente utilizada na implementação de serviços distribuídos que toleram faltas e intrusões. Originalmente as abordagens baseadas nesta técnica necessitavam $3f+1$ servidores para tolerar f faltas. Recentemente, através do uso de modelos híbridos, que possuem componentes confiáveis, algumas abordagens conseguiram reduzir este número para $2f+1$. Para construir estes componentes confiáveis é necessário fazer algumas modificações complexas nos servidores, tanto do ponto de vista de *software* quanto de *hardware*.

A arquitetura de sistema proposta neste trabalho é baseada em um modelo, chamado de modelo híbrido, em que as suposições de sincronismo, presença e severidade de faltas e falhas variam de componente para componente. O modelo aqui proposto utiliza uma abstração de compartilhamento de dados - os Registradores Compartilhados Distribuídos - e explora o uso de tecnologias de virtualização para simplificar a criação da componente inviolável de tolerância a faltas. Com esta arquitetura é possível diminuir a quantidade de recursos computacionais necessários de $3f+1$ para $2f+1$, além de alcançar uma latência (em números de passos para comunicação) comparável apenas com algoritmos especulativos.

Palavras-chave: Tolerância a faltas Bizantinas. Tolerância a intrusões. Replicação Máquina de Estados. Sistemas Distribuídos. Virtualização. Modelo Híbrido

ABSTRACT

The occurrence of faults and failures in computer systems can lead to disasters and damages in human, structural and financial meanings. Recently, faults in computer systems have appeared most often in the form of intrusions, which are the result of an attack that succeeds by exploiting one or more vulnerabilities. A recurrent issue is the discussion of how much we can trust in the execution of these systems, demonstrating the need for better implementation of concepts such as dependability, where it is expected that the system works according to their specifications, although some components have problems. State Machine Replication is a technique commonly used in the implementation of distributed services that tolerate faults and intrusions. Originally approaches based on this technique needed $3f+1$ servers to tolerate f faults. Recently, through the use of hybrid models that have reliable components, some approaches have succeeded in reducing this number to $2f+1$. To build these reliable components is necessary to make some complex modifications in the servers, in meanings of software and hardware. The system architecture proposed in this work is based on a hybrid model, in which the assumptions of timing, presence and severity of faults and failures vary from component to component. The proposed model uses an abstraction of data sharing - Distributed Shared Registers - and explores the use of virtualization technologies to simplify the creation of the fault tolerant tamperproof component. With this architecture it is possible to reduce the amount of computational resources needed from $3f+1$ to $2f+1$, and achieve a latency (in terms of number of communication steps) comparable only to speculative algorithms.
Keywords: Byzantine fault tolerance. Intrusion tolerance. State Machine Replication. Distributed systems. Virtualization. Hybrid Model

LISTA DE FIGURAS

Figura 1 Grafo de tipos de falhas de acordo com a severidade......... 30
Figura 2 General traidor... 35
Figura 3 Tenente traidor... 36
Figura 4 Possibilidade de acordo bizantino com $3f+1$ participantes... 37
Figura 5 Visão geral da arquitetura.............................. 74
Figura 6 Fluxo da requisição à resposta 77
Figura 7 Fluxo de uma mudança de líder 81
Figura 8 Fluxo da difusão atômica................................ 88
Figura 9 Camadas dos protótipos 93
Figura 10 RegPaxos e DIFATO - Dispersão com 1 cliente e sem faltas .. 95
Figura 11 RegPaxos e DIFATO - Histograma com 1 cliente e sem faltas 96
Figura 12 RegPaxos - Dispersão com 10 clientes e sem faltas.......... 97
Figura 13 DIFATO - Dispersão com 10 clientes e sem faltas 97
Figura 14 RegPaxos - Histograma com 10 clientes e sem faltas 98
Figura 15 DIFATO - Histograma com 10 clientes e sem faltas 98
Figura 16 RegPaxos - Dispersão com 20 clientes e sem faltas.......... 99
Figura 17 DIFATO - Dispersão com 20 clientes e sem faltas 99
Figura 18 RegPaxos - Histograma com 20 clientes e sem faltas 100
Figura 19 DIFATO - Histograma com 20 clientes e sem faltas 100
Figura 20 Quantidade de clientes e a influência na latência 101
Figura 21 Quantidade de clientes e a influência na vazão 102
Figura 22 Dispersão com 1% de faltas............................. 104
Figura 23 Histograma com 1% de faltas 104
Figura 24 Dispersão com 10% de faltas 105
Figura 25 Histograma com 10% de faltas............................ 105
Figura 26 Dispersão com 25% de faltas 106
Figura 27 Histograma com 25% de faltas............................ 106
Figura 28 Dispersão com 50% de faltas 107
Figura 29 Histograma com 50% de faltas............................ 107
Figura 30 Relação entre a quantidade percentual de faltas e a latência .. 108
Figura 31 Relação entre a quantidade percentual de faltas e a vazão 108

LISTA DE TABELAS

Tabela 1 Distribuição das faltas 103
Tabela 2 Comparação entre as propriedades dos protocolos avaliados (cenário otimista) 109
Tabela 3 Comparação entre as propriedades de protocolos de difusão atômica (cenário otimista). 109

LISTA DE ABREVIATURAS E SIGLAS

SMR .... State Machine Replication

BFT .... Bizantine Fault Tolerance

PBFT .... Practical Bizantine Fault Tolerance

VM ....Virtual Machine

VMM ....Virtual Machine Monitor

DSR .... Distributed Shared Register

DIFATO ... Difusão Atômica Tolerante a Faltas

SUMÁRIO

1 INTRODUÇÃO 17
1.1 MOTIVAÇÃO 17
1.2 PROPOSTA DO TRABALHO 19
1.3 OBJETIVOS 20
1.3.1 Objetivo Geral 20
1.3.2 Objetivos Específicos 20
1.4 ORGANIZAÇÃO DO TEXTO 20
2 CONCEITOS BÁSICOS EM COMPUTAÇÃO DISTRIBUÍDA . 23
2.1 AMBIENTE DE COMPUTAÇÃO DISTRIBUÍDA 23
2.1.1 Processos 23
2.1.2 Sincronismo 24
2.1.3 Modelo de sistema 26
2.1.3.1 Modelo de interação 27
2.1.3.2 Modelo de falhas 28
2.1.3.3 Modelo de segurança 28
2.1.3.4 Tipos de falhas 29
2.1.3.5 Canais de comunicação 31
2.1.4 Memória compartilhada emulada 32
2.2 TOLERÂNCIA A FALTAS E INTRUSÕES 33
2.2.1 Problemas de acordo 34
2.2.1.1 Consenso 34
2.2.1.2 Acordo bizantino 35
2.2.1.3 Difusão com ordem total 36
2.2.2 Replicação de máquinas de estados (RME) 38
2.2.3 Modelo híbrido de tolerância a faltas 39
2.3 TECNOLOGIA DE VIRTUALIZAÇÃO 41
2.4 CONSIDERAÇÕES FINAIS 43
3 TRABALHOS CORRELATOS 45
3.1 ABORDAGENS HOMOGÊNEAS 46
3.1.1 Practical Byzantine Fault Tolerance 46
3.1.2 Zyzzyva 48
3.1.3 Separando o acordo da execução 50
3.1.3.1 Separating agreement from execution for byzantine fault tolerant services 50
3.1.3.2 Espaço aumentado de tuplas e protegido por políticas 51
3.2 ABORDAGENS HÍBRIDAS 54

3.2.1 Attested append-only memory: Making adversaries stick to their word . . . 54
3.2.2 Componente inviolável através do uso de *Hardware* . . . 55
3.2.2.1 CheapBFT: Resource-efficient Byzantine Fault Tolerance . . . 56
3.2.2.2 Efficient Byzantine Fault Tolerance . . . 57
3.2.3 Abordagens com virtualização . . . 61
3.2.3.1 VM-FIT: Supporting intrusion tolerance with virtualisation technology . . . 61
3.2.3.2 Remus: High Availability via Asynchronous Virtual Machine Replication . . . 63
3.2.3.3 ZZ: Cheap practical BFT using virtualization . . . 64
3.2.3.4 Diverse replication for single-machine byzantine-fault tolerance 65
3.2.4 Usando canal confiável . . . 66
3.2.4.1 How to tolerate half less one byzantine nodes in practical distributed systems . . . 67
3.3 CONSIDERAÇÕES FINAIS . . . 69
4 REPLICAÇÃO DE MÁQUINAS DE ESTADO UTILIZANDO REGISTRADORES COMPARTILHADOS DISTRIBUÍDOS . . . 71
4.1 MODELO DO SISTEMA E ARQUITETURA . . . 72
4.1.1 Tecnologia de virtualização . . . 74
4.1.2 Registradores compartilhados distribuídos . . . 75
4.2 O PROTOCOLO REGPAXOS . . . 76
4.2.1 Propriedades . . . 78
4.2.2 Execução Normal do Protocolo . . . 78
4.2.3 Execução do Protocolo na Ocorrência de Faltas . . . 80
4.2.4 Provas de corretude . . . 82
4.3 DIFATO - DIFUSÃO ATÔMICA TOLERANTE A FALTAS . . . 84
4.3.1 Propriedades da Difusão Atômica . . . 85
4.3.1.1 Execução Normal do Protocolo . . . 86
4.3.2 Execução do Protocolo na Ocorrência de Faltas . . . 87
4.3.3 Provas de corretude . . . 89
4.4 CONSIDERAÇÕES FINAIS . . . 92
5 RESULTADOS EXPERIMENTAIS E ANÁLISES . . . 93
5.1 DETALHES DA IMPLEMENTAÇÃO . . . 93
5.2 TESTES E ANÁLISES SOBRE OS PROTÓTIPOS . . . 94
5.2.1 Ambiente de testes . . . 94
5.2.2 Considerações Sobre os Testes Realizados . . . 95
5.2.3 Execução normal . . . 95
5.2.4 Execuções com falhas . . . 102
5.2.4.1 Experimento 1% faltas . . . 103
5.2.4.2 Experimento 10% faltas . . . 104

5.2.4.3 Experimento 25% faltas 105
5.2.4.4 Experimento 50% faltas 106
5.3 COMPARAÇÃO QUALITATIVA COM OUTRAS ABORDAGENS 108
5.4 CONSIDERAÇÕES FINAIS DO CAPÍTULO 109
6 CONCLUSÃO E PERSPECTIVAS FUTURAS 111
Referências Bibliográficas 113

1 INTRODUÇÃO

1.1 MOTIVAÇÃO

É impossível imaginar como seria nosso cotidiano sem o uso de sistemas computacionais. Eles estão presentes em praticamente todos os aspectos de nossas vidas, na compra de um pão, ao efetuar transações financeiras ou no controle hospitalar. A ocorrência de faltas e falhas nestes sistemas pode levar a catástrofes e prejuízos humanos, estruturais e financeiros. Recentemente, as faltas em sistemas computacionais têm aparecido mais frequentemente sob a forma de intrusões, que são o resultado de um ataque que obtém sucesso ao explorar uma ou mais vulnerabilidades (CORREIA et al., 2005). Uma questão recorrente é a discussão de quanto podemos confiar no funcionamento destes sistemas, demonstrando a necessidade de uma melhor aplicação de conceitos como dependabilidade, onde é esperado que o sistema funcione conforme suas especificações, ainda que alguns componentes apresentem problemas.

A aplicação de conceitos de dependabilidade para construir sistemas distribuídos seguros tem crescido de maneira considerável sob a designação de tolerância a intrusão (VERÍSSIMO; NEVES; CORREIA, 2003; CORREIA; NEVES; VERISSIMO, 2004). Esta área tem sido alvo de muitas pesquisas nas últimas décadas (LAMPORT; SHOSTAK; PEASE, 1982) e dentre as abordagens utilizadas, destacam-se as que fazem uso de redundância. Estas técnicas são implementadas através da replicação de máquina de estados (RME) (LAMPORT, 1978; SCHNEIDER, 1982) que combina uma série de mecanismos que contribuem para a manutenção da disponibilidade e integridade das aplicações, bem como dos ambientes de execução (LUIZ et al., 2008).

A abordagem de RME tem sido utilizada para tolerar faltas bizantinas (arbitrárias) (REITER, 1995; CASTRO; LISKOV, 2002), mantendo o funcionamento correto do sistema ainda que tenha havido intrusões. Os algoritmos tolerantes a intrusão têm como objetivo permitir que os sistemas continuem operando dentro das suas especificações de funcionamento, ainda que alguns de seus componentes apresentem comportamento arbitrário ou, até mesmo, malicioso (AMIR et al., 2006; CASTRO; LISKOV, 2002; REITER, 1995; YIN et al., 2003). Alguns trabalhos comprovaram que através do uso destes algoritmos é possível projetar serviços confiáveis como sistemas de arquivos em rede, *backup* cooperativo, serviços de coordenação, autoridades certificadoras, banco de dados, sistemas de gerenciamento de chaves, etc (VERO-

NESE et al., 2011).

Os algoritmos tolerantes a faltas bizantinas (BFT - acrônimo do inglês *Byzantine Fault Tolerant*), em sua maioria, necessitam de um mínimo de $3f+1$ réplicas (CASTRO; LISKOV, 2002; REITER, 1995; KOTLA et al., 2008) para tolerar f faltosas. Entretanto, quando usada para tolerar faltas de *crash* (parada), a redundância de máquinas diminui para apenas $2f+1$ réplicas (SCHNEIDER, 1990a). O número adicional de réplicas para o primeiro caso se faz necessário para tolerar comportamentos maliciosos e/ou arbitrários. Este custo se torna ainda mais alto se considerarmos que a heterogeneidade é uma premissa importante neste tipo de sistema (GARCIA et al., 2011; OBELHEIRO et al., 2006).

Trabalhos recentes conseguem melhorar a resiliência dos sistemas BFT tolerando faltas com apenas $2f+1$ réplicas, utilizando diferentes abordagens (VERONESE et al., 2011; CHUN et al., 2007; LEVIN et al., 2009; CORREIA; NEVES; VERISSIMO, 2004; REISER; KAPITZA, 2007; JÚNIOR et al., ; STUMM et al., 2010).

Os modelos de sistema destes trabalhos consideram suposições híbridas para as entidades que os compõe (CORREIA et al., 2002; VERÍSSIMO, 2006). Estas suposições baseiam-se na separação do sistema em dois subsistemas, um deles inviolável e sujeito apenas a faltas de *crash* e outro sujeito a qualquer tipo de falta arbitrária. Se formos considerar a viabilidade de se reproduzir alguns destes sistemas, pode-se dizer que são de alta complexidade, pois, para criar o subsistema inviolável, muitos requerem *hardwares* próprios (VERONESE et al., 2011; CHUN et al., 2007; LEVIN et al., 2009) ou então alto acoplamento ao *kernel* do sistema operacional que os suportam (CORREIA; NEVES; VERISSIMO, 2004).

Além da resiliência, outro fator importante para avaliação do desempenho de sistemas BFT é o atraso no processamento de uma requisição, ou latência (CORREIA; NEVES; VERISSIMO, 2012). A quantidade de passos de comunicação necessária durante uma execução na ausência de falta é considerada como métrica de latência nestes sistemas (VERONESE et al., 2011). Os algoritmos especulativos, em que os clientes participam ativamente do progresso do sistema, são os que possuem a menor quantidade de passos (KOTLA et al., 2008; VERONESE et al., 2011). Portanto, um bom ponto a ser avaliado é a diminuição da latência sem a necessidade de intervenção dos clientes.

1.2 PROPOSTA DO TRABALHO

A ideia deste trabalho é revisitar o modelo híbrido de sistemas, explorando uma nova abordagem de implementação do subsistema inviolável chamado de Registradores Distribuídos Compartilhados (DSR - acrônimo do inglês *Distributed Shared Register*). Somente as partes cruciais de algoritmos de replicação de máquinas de estados tolerantes a faltas Bizantinas serão executadas neste subsistema. O objetivo é reduzir de $n \geq 3f+1$ para $n \geq 2f+1$ a quantidade de recursos computacionais físicos. A abordagem proposta, utilizando o DSR, tem a vantagem - em relação às anteriores - de não necessitar que haja modificações nos sistemas operacionais nem no *hardware* das máquinas servidoras. Ao invés disto, a arquitetura do subsistema inviolável será embasada em tecnologias largamente disponíveis e de simples configuração, como tecnologias de virtualização e emulação de memória compartilhada.

A proposta apresentada consiste na criação de um subsistema inviolável para ser utilizado em conjunto com outros subsistemas para a criação de um sistema distribuído tolerante a faltas Bizantinas. Este subsistema é responsável por permitir a comunicação de mensagens entre os recursos componentes da replicação de máquinas de estado e seu foco é direcionado para a troca de informações que garantam que estas réplicas entrem em consenso em relação ao estado atual do sistema. A ideia principal por trás deste modelo é permitir que as réplicas recebam as mesmas informações na mesma ordem, de modo que suas transições de estado sejam iguais. Desta forma, evitando a inconsistência de estados do sistema, todas as máquinas podem funcionar corretamente permitindo que qualquer sistema de tolerância a faltas seja criado sobre este modelo híbrido.

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 de computação em nuvens. É também indicada quando não se pode esperar pela ativação de réplicas em espera em caso de falha. Nestes casos, uma solução para replicação de máquinas de estados BFT eficiente - com apenas 2f + 1 réplicas físicas - usando virtualização pode ser adequada.

1.3 OBJETIVOS

1.3.1 Objetivo Geral

O principal objetivo do presente trabalho é investigar e estudar as técnicas existentes de modelos híbridos tolerantes a faltas Bizantinas baseados em Replicação de Máquina de Estados. Em seguida, propor um modelo que visa trazer uma solução simples para a criação de um componente inviolável capaz de servir de base para a solução de problemas de consenso com baixa latência e resiliência. Este modelo deve ser baseado em virtualização e em memória compartilhada emulada.

1.3.2 Objetivos Específicos

De acordo com o objetivo geral acima, alguns objetivos específicos se fazem necessários:

1. Levantamento e estudo de referências teóricas, de modo a formar uma base sólida de conceitos relacionados ao estudo de sistemas híbridos de tolerância a faltas bizantinas.

2. Especificação de uma arquitetura que faça de maneira clara a distinção entre os subsistemas do modelo híbrido.

3. Especificação de um protocolo de difusão atômica que servirá de base para o protocolo de replicação de máquinas de estados.

4. Especificação de um protocolo de consenso baseado na arquitetura descrita acima, bem como a elaboração de suas provas de correção.

5. Testes de execução e avaliação do desempenho obtido com os protótipos.

6. Especificação e implementação de protocolos que, juntamente ao protocolo de consenso, garantam a execução segura de serviços no modelo proposto.

1.4 ORGANIZAÇÃO DO TEXTO

Esta dissertação está divida em 6 capítulos. A introdução foi apresentada neste capítulo demonstrando a motivação e os objetivos do trabalho proposto. O restante da escrita está organizado da seguinte maneira:

- **Capítulo - 2 Conceitos básicos em computação distribuída:** Neste capítulo são apresentados os conceitos que formam uma base teórica da área e que auxiliam no desenvolvimento das ideias apresentadas na dissertação.
- **Capítulo - 3 Trabalhos correlatos:** São apresentados neste capítulo os trabalhos que se relacionam diretamente com o tema abordado na dissertação. Estes trabalhos representam o estado da arte da literatura e servem de base e inspiração para a dissertação.
- **Capítulo - 4 Replicação de máquinas de estado utilizando Registradores Compartilhados Distribuídos:** Este capítulo apresenta os detalhes sobre o modelo de sistema, arquitetura, premissas básicas e algoritmos propostos neste trabalho.
- **Capítulo - 5 Resultados Experimentais e Análises:** Este capítulo detalha os protótipos que são utilizados para a validação do modelo proposto. São abordados os ambientes utilizados, os cenários criados e os testes efetuados, bem como as análises dos dados obtidos. Uma comparação qualitativa é apresentada também entre os protótipos e outras abordagens na literatura.
- **Capítulo - 6 Conclusão e Perspectivas Futuras:** Por fim, este capítulo apresenta as conclusões da dissertação, além de alguns pontos de melhoria para trabalhos futuros.

2 CONCEITOS BÁSICOS EM COMPUTAÇÃO DISTRIBUÍDA

Neste capítulo serão apresentados conceitos básicos em computação distribuída. As informações aqui contidas servem de base para a proposta. Os conceitos apresentados vão desde pontos básicos para o entendimento - como a definição do que são Processos - até o aprofundamento em questões diretamente ligadas a este trabalho, por exemplo ambientes virtualizados. A leitura deste capítulo é de suma importância para o entendimento completo da proposta.

2.1 AMBIENTE DE COMPUTAÇÃO DISTRIBUÍDA

A computação distribuída se dá pela execução paralela e descentralizada de uma tarefa comum, por dois ou mais computadores conectados através de uma rede. Segundo a definição de Andrew Tanenbaum, um sistema distribuído, é uma coleção de computadores independentes que se apresenta ao usuário como um sistema único e consistente (TANENBAUM; STEEN, 2002).

2.1.1 Processos

Um processo corresponde à execução de um algoritmo em um processador (ATTIYA; WELCH, 2004) (LYNCH, 1996). Processos em sistemas distribuídos podem ser abstraídos como unidades capazes de executar computações através da noção de processo (GUERRAOUI; RODRIGUES, 2006a). Isto é, um processo deve ser entendido como uma entidade independente, com seu próprio contador de programa e estado interno (TANENBAUM, 1992). Este estado, ou este conjunto de estados, evolui na medida em que os passos descritos no processo são executados. O estado global do sistema distribuído é composto pelo estado local de cada um dos processos e pelo estado dos canais de comunicação (CHANDY; LAMPORT, 1985). As comunicações interprocessos são feitas através dos *links* ou canais de comunicação. A inicialização de um sistema distribuído ocorre quando os processos se encontram em seus estados iniciais (arbitrários) e os canais estão vazios (LYNCH, 1996).

2.1.2 Sincronismo

Todo computador possui seu próprio relógio interno, o qual pode ser usado pelos processos locais. Dois processos sendo executados em diferentes computadores podem associar indicações de tempo aos seus eventos. Entretanto, mesmo que estes dois processos leiam seus relógios locais ao mesmo tempo, nada garante que os valores serão iguais. Isso ocorre por que seus relógios possuem taxas de desvio diferentes (LAMPORT; MELLIAR-SMITH, 1985). Isto é, mesmo que todos os relógios de um sistema distribuído fossem inicialmente ajustados com o mesmo horário, com o passar do tempo eles variariam entre si significativamente, a não ser que fossem periodicamente reajustados (COULOURIS; DOLLIMORE; KINDBERG, 2005).

Parte da solução destes problemas está na utilização de relógios com melhor precisão, isto é, com taxa de atraso menores. Existem várias estratégias para corrigir os tempos nos relógios em computadores, como por exemplo, o uso de receptores de rádio que oferecem a precisão em 1 microssegundo (COULOURIS; DOLLIMORE; KINDBERG, 2005) ou então, os relógios atômicos que oferecem precisão de até 1 zepta segundo (1 x 10^{-21}) (BACKE, 2012). Entretanto, quanto mais precisos estes relógios são, mais custosos do ponto de vista de obtenção e manutencão. Uma solução prática é a sincronização periódica dos relógios (LAMPORT, 1978). Esta solução não é simples, pois o problema da sincronização em sistemas distribuídos precisa lidar com a troca de mensagens em canais de comunicação onde não se pode assumir tempo de entrega.

Se em qualquer momento a diferença Δ entre os valores retornados por dois relógios respeitar a regra $\Delta \leq \varepsilon$, então pode-se dizer que eles estão ε-sincronizados (DÉFAGO; SCHIPER; URBÁN, 2003). Se $\varepsilon = 0$, então os relógios estão perfeitamente sincronizados.

A sincronização de relógios tem sido estudada há décadas, e dentre as abordagens mais conhecidas para efetuá-la estão, o método de Cristian (CRISTIAN, 1989), o método de Berkeley (GUSELLA; ZATTI, 1989) e *(*Network Time Protocol) (NTP) (MILLS, 1995). Estes métodos visam ajustar os relógios através de troca de mensagens entre diferentes computadores. Nestes modelos, é preciso entender que o tempo de entrega da mensagem para sincronização e o tempo de processamento influenciam no desvio dos relógios. Por este fato, a precisão atingida nem sempre é satisfatória.

Em (LAMPORT, 1978) é discutido o mecanismo de relógios lógicos que servem para ordenação de eventos ao invés de lidar com a sincronização de relógios das máquinas. Neste algoritmo, cada processo mantém um contador crescente e monotônico C e cada evento a possui uma marca temporal $C(a)$ com a qual todos os processos concordam. Assim, os eventos estão

sujeitos às seguintes propriedades derivadas da relação *happens-before*:

- Se, em um processo, a acontece antes de b, então $C(a) < C(b)$.
- Se a e b representam, respectivamente, o envio e o recebimento de uma mensagem, então $C(a) < C(b)$.
- Sejam a e b eventos quaisquer, então $C(a) \neq C(b)$

Um aspecto importante da caracterização dos sistemas distribuídos está relacionado ao comportamento de seus processos com o passar do tempo (GUERRAOUI; RODRIGUES, 2006a). De maneira sucinta, isto determina quando se pode fazer suposições sobre sincronismo. Usualmente o sincronismo de um sistema distribuído é definido através de três características básicas (HADZILACOS; TOUEG, 1994) (CRISTIAN et al., 1995):

1. Tempo de processamento;
2. Tempo de entrega de mensagem;
3. Desvio do relógio local onde está se executando o processo.

Os sistemas distribuídos podem ser considerados assíncronos ou síncronos, tudo depende das suposições que se pode fazer sobre eles com relação ao tempo. Sistemas assíncronos são aqueles em que não se pode assumir nenhuma hipótese sobre tempo físico com relação aos processos e canais de comunicação. Os sistemas síncronos, por outro lado, permitem que se assuma as seguintes propriedades (GUERRAOUI; RODRIGUES, 2006a):

- Computação síncrona, onde se tem os limites superiores no tempo de processamento. Isto é, dado qualquer processamento, esse limite nunca será superado.
- Comunicação síncrona, onde se tem os limites superiores no tempo de transmissão. Isto é, dado qualquer envio e entrega de mensagem, esse limite nunca será superado.
- Relógio físicos síncronos, onde se tem os limites superiores da taxa de desvio do relógio.

As suposições em sistemas assíncronos são mais fracas que em sistemas síncronos (VERÍSSIMO, 2006). É mais simples supor que não existem limites de tempo para determinadas tarefas do que supor, por exemplo, o limite superior de tempo de entrega de mensagens no sistema. Porém, alguns problemas só podem ser resolvidos em sistemas síncronos. Fischer, Lynch e Paterson (FLP) mostraram que qualquer protocolo para sistemas assíncronos

tem a possibilidade de não terminação se qualquer processo puder sofrer *crash* (FISCHER; LYNCH; PATERSON, 1985). Então, segundo FLP, nenhum protocolo determinístico pode resolver o problema de consenso (vide 2.2.1.1) em um sistema assíncrono.

Existem ainda os sistemas parcialmente síncronos. Em (GUERRAOUI; RODRIGUES, 2006a), os sistemas parcialmente síncronos são definidos como aqueles em que as suposições de tempo ocorrem, em algum momento, sem definir exatamente quando. Sistemas reais são parcialmente síncronos (VERÍSSIMO, 2006), pois, em geral, é fácil de definir limites físicos de tempo, entretanto, existem alguns momentos em que estas suposições não se encaixam. Um exemplo de suposição seria, na ausência de faltas, uma mensagem é entregue de um processo a outro em até 5ms.

2.1.3 Modelo de sistema

Um modelo de arquitetura de sistema define a forma como seus componentes interagem e a maneira pela qual estão mapeados em redes subjacentes. O seu objetivo global é garantir que a estrutura atenda as demandas atuais e, provavelmente, as futuras impostas sobre ela. As maiores preocupações são tornar o sistema confiável, gerenciável, adaptável e rentável.

Para descrever melhor os modelos fundamentais este texto baseia-se em (COULOURIS; DOLLIMORE; KINDBERG, 2005), no qual é feita a subdivisão em três outros modelos:

- O modelo de interação, que trata do desempenho e da dificuldade em lidar com limites de tempo nos sistemas distribuídos.
- O modelo de falha, que especifica detalhadamente quais são as possíveis falhas que os componentes e os canais de comunicação podem sofrer. É neste modelo também que se define a noção de comunicação confiável e da correção de processos.
- O modelo de segurança, que discute as principais ameaças aos processos e aos canais de comunicação. Aqui é definido o conceito de canal seguro.

Todos os modelos de arquitetura de sistemas distribuídos são compostos por processos que se comunicam por meio do envio de mensagens através de uma rede de computadores. Um modelo de sistema precisa definir: (1) quais são as entidades presentes no sistema; (2) como elas interagem, e; (3) quais são as características que afetam seus comportamentos individualmente e coletivamente.

Um modelo precisa tornar explícitas todas as suposições relevantes sobre os sistemas que modela, fazendo generalizações a respeito do que é possível (ou impossível) diante delas. As propriedades garantidas pelo modelo dependem da análise lógica e, em alguns casos, de provas matemáticas.

2.1.3.1 Modelo de interação

No modelo de interação são definidas questões de limite de tempos e sincronismo. As medidas de desempenho em uma rede de computadores são:

- A Latência, que representa o atraso decorrido entre o início da transmissão de uma mensagem no processo p e o início da sua recepção pelo processo p'.

- A Largura de banda, que é o volume total de informações que pode ser transmitido em determinado momento.

- *Jitter*, que é a variação estatística do atraso na entrega de dados.

Um fator inerente aos sistemas distribuídos é a dificuldade em se estabelecer limites para os tempos de execução de um processo, das trocas de mensagens e para as taxas de desvio dos relógios. Dois pontos de vista diferentes fornecem modelos simples, são eles:

- Sistemas distribuídos síncronos - Nos quais, segundo (HADZILACOS; TOUEG, 1994), (i) o tempo para executar cada etapa de um processo tem limites inferior e superior conhecidos; (ii) cada mensagem transmitida em um canal é recebida dentro de um tempo limitado e conhecido; (iii) cada processo tem um relógio local cuja taxa de desvio do tempo real tem um limite conhecido.

- Sistemas distribuídos assíncronos - Nos quais, segundo (COULOURIS; DOLLIMORE; KINDBERG, 2005), não existem considerações sobre: (i) as velocidades de execução de processos - a única afirmação válida é que cada etapa pode demorar um tempo arbitrariamente longo; (ii) os atrasos na transmissão das mensagens - uma mensagem pode ser recebida após um tempo arbitrariamente longo; (iii) as taxas de desvio de relógio - a taxa de desvio de um relógio é arbitrária.

Em 2.1.2 é apresentado o problema de sincronismo.

2.1.3.2 Modelo de falhas

O modelo de falhas define como as falhas podem vir a ocorrer proporcionando um entendimento dos seus efeitos e consequências. Em sistemas distribuídos tanto os processos quanto os canais de comunicação podem divergir do comportamento correto (ou desejável), caracterizando uma falha.

Em (HADZILACOS; TOUEG, 1994) é fornecida uma taxonomia que distingue as falhas em:

- Falhas por omissão - Casos onde um processo ou um canal de comunicação deixa de executar as ações que deveria.
- Falhas arbitrárias (ou bizantinas) - Descreve uma semântica onde qualquer tipo de erro pode ocorrer.
- Falhas de sincronização (ou temporização) - Aplicáveis aos sistemas distribuídos síncronos onde limites de tempo são estabelecidos para o tempo de execução dos processos. Estas falhas podem ser no:
  - Processo, por exemplo, o relógio local ultrapassa os limites de sua taxa de desvio em relação ao tempo físico.
  - Canal, por exemplo, a transmissão de uma mensagem demora mais do que o limite definido.

Em 2.1.3.4 são discutidos mais profundamente os tipos de falhas.

2.1.3.3 Modelo de segurança

O modelo de segurança, de acordo com (COULOURIS; DOLLIMORE; KINDBERG, 2005), é baseado no princípio de que a segurança de um sistema distribuído pode ser obtida tornando seguros os processos e os canais usados para suas interações e protegendo contra acesso não autorizado os objetos que encapsulam. Em suma, **o modelo de segurança define políticas de acesso e mecanismos de proteção para as entidades e canais do sistema**.

Faz parte do modelo de segurança o detalhamento de como os objetos do sistema serão protegidos, isto é, os mecanismos que serão utilizados para que usuários e processos não acessem regiões do sistema às quais não tem permissão de acesso. Para tal, os usuários e processos precisam de autorizações. A verificação das autorizações deve ser modelada.

O modelo de segurança deve definir claramente como as interações vão ocorrer e quais são os mecanismos de segurança sob o qual residem.

Em sistemas distribuídos a comunicação, em geral, ocorre através da troca de mensagens entre os processos. Estas mensagens carregam informações importantes para o bom funcionamento do sistema. Os sistemas distribuídos costumam ser implantados de maneira que exista acesso externo aos mesmos, o que torna tanto os processos quanto as suas interações vulneráveis a ataques. De forma sucinta, o modelo de segurança traz uma discussão detalhada sobre como o sistema se protege contra invasores, ameaças aos processos e aos canais de comunicação. Deve detalhar quais técnicas serão utilizadas, código de autenticação de mensagem (*message authentication code*), criptografia e compartilhamento de segredos, autenticação, utilização de canais seguros (vide 2.1.3.5), etc.

Os modelos de segurança precisam ter provas do funcionamento correto quando necessário e precisam considerar o custo de processamento e gerenciamento necessário na utilização de algumas técnicas, como por exemplo, criptografia. Estas decisões podem ser importantes para a segurança do sistema, mas podem ser custosas para o resto do modelo, tornando-se impraticáveis.

2.1.3.4 Tipos de falhas

Uma falha de sistema ocorre quando o serviço prestado se desvia de cumprir sua função conforme a especificação do sistema. Falhas são causadas por erros. Erros ocorrem em tempo de execução quando alguma parte do sistema entra em um estado inesperado devido à ativação de uma falta. Faltas são defeitos, um passo incorreto, processo ou definição de dados que faz com que o sistema passe a se comportar de forma não intencional ou imprevista. Faltas podem ser um *bug* em um programa, um problema de configuração e/ou uma interação originada de um sistema externo ou um usuário. Um erro não necessariamente provocará uma falha, por exemplo, uma exceção pode ser lançada e o funcionamento global do sistema continuará em conformidade com a especificação. De maneira geral, uma falta, quando ativada, pode levar a um erro que pode levar ou a outro erro ou a uma falha

Um processo que executa corretamente sua especificação é chamado **correto**, em contrapartida, um processo que não executa corretamente o algoritmo especificado é denominado **não-correto** ou **faltoso**. Guerraoui e Rodrigues (2006) definem que, a menos que falhe, espera-se de um processo que ele execute o algoritmo a ele atribuído, através de um conjunto de componentes que implementam este algoritmo dentro do processo. Guerraoui e Rodrigues ainda afirmam que quando um processo falha assume-se que todos os componentes também falharão, e ao mesmo tempo. As falhas dos processos podem

ocorrer tanto no domínio do tempo quanto no domínio dos valores. Os tipos de falhas no domínio do tempo, de acordo com a literatura(HADZILACOS; TOUEG, 1994; DÉFAGO; SCHIPER; URBÁN, 2004), são:

- **Parada:** o processo para de funcionar de maneira antecipada. Ex.: Desligamento da máquina onde ocorria o processamento;

- **Omissão de Envio:** o processo incorre em omissão, aleatória ou eventual, de envio de mensagens;

- **Omissão de Recepção:** o processo deixa de receber mensagens a ele enviadas de maneira aleatória ou eventual;

- **Falha de temporização:** o processo viola uma das suposições de sincronismo. Este tipo de falha é irrelevante para sistemas assíncronos.

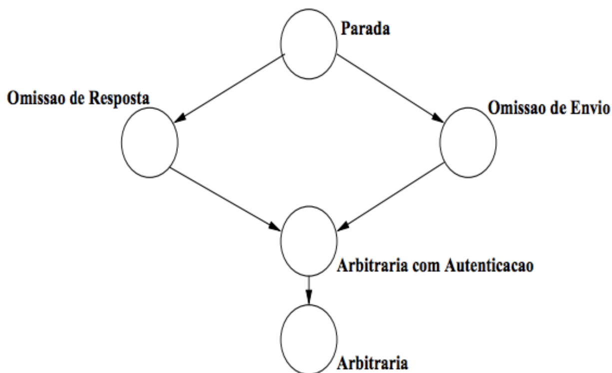


Figura 1 – Grafo de tipos de falhas de acordo com a severidade.

Estas falhas ocorrem no domínio do tempo e são mais simples de se tratar. Falhas que ocorrem no domínio dos valores são chamadas arbitrárias ou bizantinas (LAMPORT; SHOSTAK; PEASE, 1982). À rigor, estas falhas englobam as anteriores, portanto, elas podem ocorrer tanto no domínio do tempo quando no domínio dos valores:

- **Bizantina, maliciosa ou arbitrária**: O processo faltoso pode apresentar qualquer tipo de comportamento. Isto é, pode se comportar como as falhas anteriormente citadas ou apresentar qualquer tipo de comportamento arbitrário. Estas falhas podem ser intencionais ou não;

- **Bizantina com Autenticação:** Semelhante à bizantina, entretanto possui autenticação não forjável que permite a detecção do comportamento bizantino.

As falhas arbitrárias são as mais custosas para se tolerar, mas esta é a única opção aceitável quando se requer uma cobertura extremamente elevada ou quando existe o risco de algum processo ser controlado por usuários maliciosos que deliberadamente tentam corromper o funcionamento correto do sistema (GUERRAOUI; RODRIGUES, 2006a).

É importante ressaltar que um comportamento arbitrário não é necessariamente malicioso e intencional, sua ocorrência pode ser devido a um erro de implementação, da linguagem de programação ou mesmo do compilador (GUERRAOUI; RODRIGUES, 2006a).

A figura 1 representa o nível sequencial da severidade das falhas em um modelo. Por exemplo, falhas arbitrárias são mais severas que falhas arbitrárias com autenticação.

2.1.3.5 Canais de comunicação

A comunicação em sistemas distribuídos ocorre, de maneira geral, através da troca de mensagens entre processos. Para tanto, é necessário que haja coordenação entre os processos (COULOURIS; DOLLIMORE; KINDBERG, 2005). Na comunicação em sistemas distribuídos os canais de comunicação ou *links* (GUERRAOUI; RODRIGUES, 2006a) representam o sistema subjacente de comunicação e fornecem uma topologia de conectividade completa entre os elementos do sistema. A comunicação entre processos é também passível de falhas, por isso podem ocorrer perdas, duplicação e corrupção de mensagens. Com respeito à confiabilidade dos canais de comunicação, duas propriedades podem ser consideradas (CHARRON-BOST; DÉFAGO; SCHIPER, 2002):

- **Sem Perdas**: Se um processo envia uma mensagem a outro processo correto, então a mensagem será recebida;

- **Perda Justa**: Se um primeiro processo envia um número infinito de mensagens a um segundo processo correto, então um número infinito de mensagens do primeiro processo será recebido pelo segundo processo.

Os canais que não admitem perdas são conhecidos na literatura como canais confiáveis (BASU; CHARRON-BOST; TOUEG, 1996). Já os canais que admitem perda são denominados *Fair-lossy links* (BASU; CHARRON-

BOST; TOUEG, 1996). Estes canais são caracterizados por três propriedades (GUERRAOUI; RODRIGUES, 2006a):

1. Perda justa (*fair-loss*): Se uma mensagem m é enviada infinitas vezes pelo processo p_i ao processo p_j, e dado que nem p_i nem p_j sofreram *crash*, então m é entregue infinitas vezes à p_j.

2. Duplicação finita (*finite duplication*): Se uma mensagem m é enviada finitas vezes pelo processo p_i ao processo p_j, então m não pode ser entregue infinitas vezes à p_j.

3. Nenhuma criação (*no creation*): Se uma mensagem m é entregue a um processo p_j, então m foi anteriormente enviada para p_j por algum processo p_i.

É possível atingir primitivas de difusão e recepção mais robustas semelhantes as dos canais confiáveis a partir dos *Fair-lossy links* , basta empregar mecanismos de reconhecimento e retransmissão (CHARRON-BOST; DÉFAGO; SCHIPER, 2002).

2.1.4 Memória compartilhada emulada

Guerraoui e Rodrigues (2006) definem memória compartilhada emulada como a construção de uma abstração de registro de um conjunto de processos que se comunicam através da troca de mensagens. Ainda segundo esta definição, não existe memória compartilhada física. De acordo com os autores, esta abordagem é bastante atraente porque, geralmente, é mais simples programar sobre uma memória compartilhada do que usar troca de mensagens. Precisamente, o programador pode ignorar os problemas de consistência advindos da distribuição de dados.

Esta definição torna-se muito interessante quando se deseja utilizar modelos híbridos para sistemas distribuídos (vide 2.2.3), já que a memória emulada pode ser construída sob aspectos e suposições diferentes do restante do sistema. Isto é, a memória emulada pode ser construída sob as premissas de ser síncrona o bastante para que se possa conhecer seus limites de tempo inferior e superior, taxas de entrega além de garantir confiabilidade. Estas premissas podem ser asseguradas através da separação da rede da memória emulada das demais redes do sistema, rede controlada, e utilização de difusão confiável com garantia de tempo para entrega da mensagem (CORREIA et al., 2002).

Uma memória compartilhada, emulada ou não, pode ser vista como um *array* de registros compartilhados. Consideramos aqui a definição sob

uma ótica de programação, ou do programador. O tipo do registro compartilhado especifica quais operações podem ser efetuadas e os valores retornados pela operação (ATTIYA; WELCH, 2004). Os tipos mais comuns são registros de leitura/escrita. As operações de um registro são invocadas por processos do sistema para troca de informações. A operação de leitura não precisa de parâmetros de entrada e tem apenas um parâmetro como saída. A operação de escrita por sua vez tem apenas um parâmetro como entrada e a saída é apenas uma confirmação se a operação foi concluída com sucesso.

Se um registro é utilizado por um único processo, e assumindo-se que não há falhas, podemos especificá-lo pelas seguintes propriedades (GUERRAOUI; RODRIGUES, 2006b):

1. *Liveness* - toda operação , em algum momento, termina.
2. *Safety* - toda leitura retorna o último valor escrito.

Mesmo que não haja apenas um processo acessando o registro, se o acesso for sequencial e se nenhum processo sofrer *crash*, então é possível manter a especificação através das propriedades antes citadas. Novamente, através do uso de modelo híbrido, é possível assegurar esta sequencialidade. Uma possível abordagem é garantir que para cada registro, apenas um processo p tem acesso de escrita e apenas outro processo p' tem acesso de leitura. Além disso, garante-se que se houver concorrência de escrita e leitura, a escrita tem precedência sobre a leitura, de modo a assegurar que o registro a ser lido será o último registro já escrito.

2.2 TOLERÂNCIA A FALTAS E INTRUSÕES

Existem certas divirgências na definição de erros, faltas e falhas, portanto é um processo não trivial definir estes termos, entretanto, no escopo deste trabalho seguiremos com as seguintes definições:

- **Falta:**: Falta é um erro latente, um potencial de problema, seja tanto de *software* quanto de *hardware*. Um exemplo seria um arquivo corrompido, é uma falta, mas se nunca for acessado, nunca acontecerá o erro de acesso.
- **Erro:** Um erro é um estado do sistema, ocasionado pela ativação de uma falta. Isto é, é o estado vindo da transição ocasionada pela interação com o objeto faltante.
- **Falha:**: De acordo com o padrão IEEE (1983), uma falha ocorre quando um programa não se comporta conforme o especificado ou apresenta resultados diferentes do planejado.

Já uma intrusão pode ser definida como o ato de introduzir-se (com astúcia ou violência)[1]. Do ponto de vista computacional, uma intrusão é um acesso não autorizado a um sistema, para qualquer fim.

2.2.1 Problemas de acordo

Muitos sistemas distribuídos devem lidar com trocas de informação que dependem de acordos entres os processos, por exemplo, a maneira como dois processos vão se comunicar para enviar mensagens entre si. A distribuição de tarefas em vários computadores também necessita que haja acordo entre todos os envolvidos para que o trabalho seja finalizado com sucesso.

Existem vários tipos de problemas de acordo e, em sua maioria, os processos envolvidos devem entrar em consenso com relação a uma decisão, isto é, deve ser uma decisão coletiva. A decisão do acordo depende da natureza do problema a ser resolvido. Aqui são apresentados os problemas de acordo ligados diretamente com a proposta.

2.2.1.1 Consenso

O problema do consenso (PEASE; SHOSTAK; LAMPORT, 1980) é uma generalização do problema do acordo em sistemas distribuídos. Ele consiste em garantir que os processos corretos em um sistema distribuído entrarão em concordância em relação a um valor proposto por algum destes processos. Formalmente o problema é definido por:

- $propose(G, v)$: o valor v é proposto dentro do grupo G;

- $decide(v)$: o valor v é decidido.

O problema se resume em proposições de valor $v \in V$ e na decisão unânime dos processos em função do v proposto. Em sua definição, as seguintes propriedades precisam ser satisfeitas:

- **acordo** - todos os processos corretos decidirão pelo mesmo v;

- **validade** - se algum processo correto decide por um $v \in V$ então v foi proposto por outro processo;

- **terminação** - todos os processos corretos acabarão por decidir.

[1] Definição segundo http://www.priberam.pt/

A validade, da maneira definida anteriormente, é comumente descartada em sistemas distribuídos sujeitos a faltas bizantinas, pois a mesma permite que um valor proposto por um processo faltoso seja decidido. Em geral implementa-se a **validade fraca** ou **não trivialidade** (CORREIA et al., 2005) (LYNCH, 1996) que estipula que se todos os processos corretos propõe inicialmente $v \in V$, então v é a única decisão possível para os processos corretos. Esta condição evita a implementação de protocolos que decidem sempre o mesmo valor independentemente das proposições dos processos. Entretanto há autores que a descartam na prática, já que quando os processos propõe valores diferentes, o valor decidido não precisa ter ligação com a entrada (BALDONI et al., 2003).

2.2.1.2 Acordo bizantino

Em ambientes onde é possível a ocorrência de faltas bizantinas, o problema de consenso ou acordo, do ponto de vista teórico, necessita que mais de dois terços dos participantes entrem em acordo (LAMPORT; SHOSTAK; PEASE, 1982) considerando-se f faltas e $3f+1$ participantes.

Em (LAMPORT; SHOSTAK; PEASE, 1982), usando o conceito de generais bizantinos, os autores provam que esta proporção é válida, e é em função deste trabalho que surge o termo faltas bizantinas. As figuras 2 e 3 mostram a impossibilidade de acordo bizantino descrita no artigo.

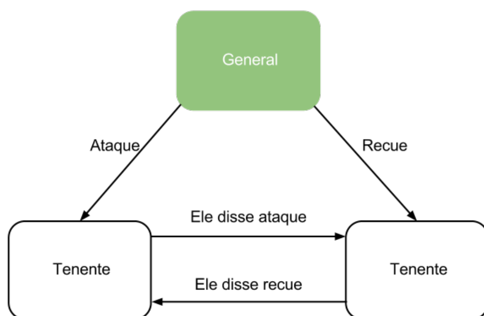


Figura 2 – General traidor.

O problema é descrito da seguinte maneira: Dado um exército com um general e seus tenentes, o general envia uma ordem aos tenentes. Esta ordem pode ser atacar ou recuar, os tenentes trocam entre si a informação que cada um recebeu do general para validá-las. No primeiro caso (figura 2) o general

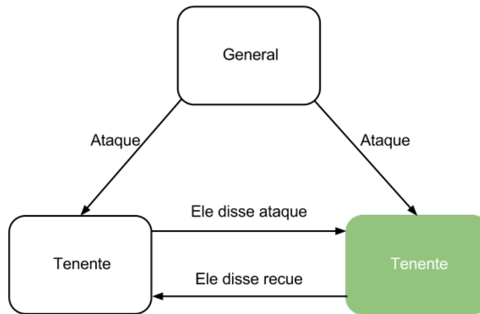


Figura 3 – Tenente traidor.

traidor emite ordens diferentes aos tenentes. Ao trocarem informações, os tenentes não chegarão a um consenso, desta forma, não saberão como agir. No segundo caso (figura 3) quem forja uma ordem é um dos tenentes, gerando o mesmo impasse. Como não existe nenhuma maneira de validar quem é o traidor, o processo simplesmente estagna. A impossibilidade de acordo bizantino com n menor que $3f+1$ é válida tanto para sistemas síncronos e assíncronos.

A figura 4 demonstra que com acréscimo de mais um participante, isto é, aumentando-se de $2f+1$ para $3f+1$ e, neste exemplo considerando-se $f=1$, é possível prosseguir com a ordem, mesmo que haja até f traidores no exército. Isto é possível em função da segunda etapa do processo, onde os tenentes trocam informações entre si. Pois, segundo a figura, ao final do processo, cada tenente terá pelo menos duas ordens para atacar conseguindo uma predominância da ordem para atacar, isto é, entrando em consenso. Este fato prova que com pelo menos $3f+1$ participantes é possível resolver o problema de acordo bizantino.

A literatura apresenta soluções práticas que conseguem diminuir o número de participantes (DOLEV; STRONG, 1983) (JÚNIOR et al.,) (VERONESE et al., 2011), entretanto ainda existe bastante espaço para pesquisas.

2.2.1.3 Difusão com ordem total

A difusão com ordem total (ou difusão atômica) força a confiabilidade na difusão de mensagens e também que todos os processos receptores entreguem as mensagens na mesma ordem. O problema pode ser definido através de duas primitivas básicas:

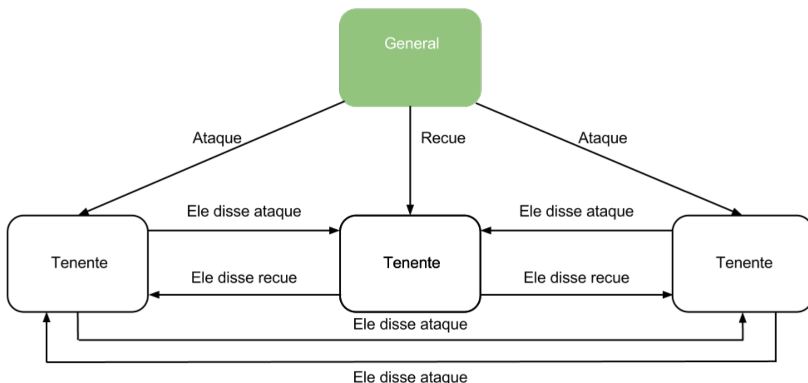


Figura 4 – Possibilidade de acordo bizantino com $3f+1$ participantes.

1. $TO\text{-}multicast(G,m)$: A mensagem m é difundida para todos os processos pertencentes ao grupo G;

2. $TO\text{-}deliver(m)$: A mensagem m é entregue pelo processo p_i para a aplicação com ordem total.

Para que se obtenha a difusão com ordem total deve-se satisfazer as seguintes propriedades (DÉFAGO; SCHIPER; URBÁN, 2004):

- Validade: Se um processo correto difunde uma mensagem em seu grupo, então algum processo correto pertencente ao mesmo grupo entregará a mensagem;

- Acordo: Se um processo correto em determinado grupo entrega uma mensagem, então todos os processos corretos pertencentes ao mesmo grupo entregarão esta mensagem;

- Integridade: Para qualquer mensagem enviada dentro de um determinado grupo, cada processo correto pertencente ao mesmo grupo a entregará apenas uma vez;

- Ordenação total local: Se dois processos corretos p e q entregam as mensagens m e m' difundidas em G, então ambos entregarão m e m' na mesma ordem.

No escopo de sistemas distribuídos, em função da possibilidade de um processo poder participar de mais de um grupo, são verificadas duas primitivas na validação da difusão, isto é, além da ordenação total local é verificada

a ordenação total global (HADZILACOS; TOUEG, 1994), que garante que a ordem de entrega de mensagens é correta mesmo sob a gerência de múltiplos grupos.

Em suma, a difusão atômica é um produto da difusão confiável com ordem total

2.2.2 Replicação de máquinas de estados (RME)

O modelo mais simples para tolerância a faltas bizantinas em sistemas distribuídos é replicação ativa ou replicação por máquinas de estado (LAMPORT, 1978). Neste modelo, o sistema é composto por réplicas que oferecem o mesmo serviço, estas máquinas possuem estados deterministas em que processam as requisições como suas entradas e tem o resultado deste processamento como as suas respostas ou saídas (SCHNEIDER, 1990a).

Uma máquina de estados está sempre em único estado, chamado de estado atual, e este estado representa a memória da máquina. Isso significa que uma máquina de estados depende apenas do seu estado inicial e das entradas para definir seu estado atual (ANDERSON, 2006). Um modelo de máquinas de estados não pode representar processos não deterministas, uma vez que nestes casos o estado agrega mais informação ou depende de fatores mais complexos do que apenas o estado inicial e a sequência de entradas. Esta característica está relacionada ao determinismo de réplicas (SCHNEIDER, 1990a), isto é, réplicas iniciadas no mesmo estado e submetidas às mesmas entradas devem obrigatoriamente atingir o mesmo estado final.

O modelo de replicação de máquinas de estado tem como requisito a **coordenação de réplicas** (SCHNEIDER, 1990a) que pode ser dividida da seguinte maneira:

1. Acordo - Todas as réplicas recebem o mesmo conjunto de requisições.

2. Ordem - Todas as réplicas corretas executam as requisições na mesma ordem.

Em resumo, todas as réplicas corretas em um modelo de máquina de estados iniciam no mesmo estado e processam a mesma sequência de requisições. Este requisito pode ser atendido se utilizarmos algum protocolo de difusão com ordem total (vide 2.2.1) no envio de requisições dos clientes para as réplicas.

O funcionamento de sistemas baseados neste modelo é simples. A requisição do cliente chega a todas as réplicas que processam e enviam suas respostas. Quando o cliente recebe um determinado número de respostas iguais entre si e de diferentes servidores o cliente aceita a resposta como

correta. Ressaltando que o número de respostas iguais depende do modelo e da quantidade de faltas toleradas pelo sistema.

Um conceito bastante empregado no uso da abordagem de RME é a diversidade (OBELHEIRO et al., 2006) (GARCIA et al., 2011). Esta técnica consiste em criar ambientes diferentes para cada réplica fazendo com que suas especificações sejam a mesma, porém seus desenvolvimentos totalmente independentes das demais, variando sistemas operacionais, *hardware*, linguagem de programação, equipe de desenvolvimento, paradigmas, etc. Isso diminui a probabilidade das réplicas apresentarem as mesmas falhas de vulnerabilidades, diminuindo não somente as chances de erros, mas também as chances de um ataque ser bem sucedido, além de contribuir na independência de faltas (RODRIGUES; CASTRO; LISKOV, 2001) (OBELHEIRO; BESSANI; LUNG, 2005).

2.2.3 Modelo híbrido de tolerância a faltas

Antes de falarmos do modelo híbrido de tolerância a faltas, cabe uma breve discussão sobre sistemas síncronos e assíncronos.

Fischer, Lynch e Paterson (FLP) mostraram que qualquer protocolo para sistemas assíncronos tem a possibilidade de não terminação se qualquer processo puder sofrer *crash* (FISCHER; LYNCH; PATERSON, 1985). Isto trouxe à tona questionamentos sobre os modelos assíncronos em que são baseadas muitas das soluções de sistemas distribuídos (VERÍSSIMO, 2006). Alguns autores contornaram o FLP fazendo com que seus sistemas fiquem síncronos o bastante para suportar o consenso (CHANDRA; TOUEG, 1996) (CRISTIAN; FETZER, 1999).

Outro fator importante é que o sincronismo de um sistema varia na dimensão do tempo e do espaço (VERÍSSIMO; CASIMIRO, 2002). Isto significa que durante o tempo de vida de um sistema, ele pode estar ou não síncrono, considerando a janela de tempo no qual ele é observado. Além disso, existe uma variação de acordo com a dimensão do espaço, isto é, considerando trechos do sistema, é mais simples predizer quais componentes são mais rápidos ou tem limites de tempo menores que outros na execução das tarefas.

Levanto em conta estes fatores, um modelo de faltas com hipóteses de falhas híbridas é aquele em que se assume que a presença e severidade de vulnerabilidades, ataques e intrusões variam de componente para componente (CORREIA; VERISSIMO; NEVES, 2002). Portanto, os princípios que embasam sistemas híbridos são (VERÍSSIMO, 2006):

- Sistemas podem ter domínios com diferentes propriedades não-funcio-

nais, como sincronismo, comportamento defeituoso, qualidade de serviço, etc.

- As propriedades de cada domínio são obtidas pela construção do(s) subsistema(s) em que se encontram.

- Estes subsistemas tem encapsulamento bem definido e interfaces por meio das quais as propriedades anteriores manifestam-se.

Este princípios reforçam a ideia de que as dimensões de tempo e espaço dentro dos sistemas tem variações e definições dependentes do(s) subsistema(s) que se analisa.

Em (VERÍSSIMO, 2006) o autor afirma que estes modelos permitem-nos tirar o melhor de ambas as dimensões. Ele mostra que modelos híbridos são:

- Expressivos, por considerarem cada componente isoladamente, isto é, diferentes velocidades e premissas de funcionamento. Em modelos homogêneos não é possível explorar essas vantagens dado que o pior caso vai nivelar todo o sistema.

- Simples para provas de correção, já que o sistema é avaliado por cada trecho que o compõe. Isto nos obriga a fazer provas para cada parte integrante dele.

- Naturalmente suportados por arquiteturas híbridas. Estas arquiteturas consideram a existência de componentes ou subsistemas com características diferentes. Modelos e arquiteturas híbridas fornecem condições de se alcançar abstrações poderosas que, em grande parte, não podem ser implementadas em modelos canônicos (homogêneos) assíncronos, por exemplo, detectores de falhas, canais ad-hoc síncronos, *triggers* disparados por tempo, etc.

- Viabilizadores de conceitos para a construção de algoritmos totalmente novos. Como nos modelos híbridos as premissas dos componentes são diferentes, podemos ter trechos síncronos implementados em ambientes assíncronos, permitindo a criação de novos algoritmos.

Em (CORREIA et al., 2002) foi apresentada uma abordagem de modelo híbrido de falhas que descreve um sistema distribuído que possui um componente inviolável local aos servidores. Este componente está sujeito apenas a falhas de *crash* e está interconectado aos componentes locais de outros membros através de uma rede controlada. Através deste componente são realizadas operações para o estabelecimento de acordo e consenso. O sistema

é dividido entre aqueles componentes que estão sujeitos as faltas bizantinas e aqueles que não estão. Este trabalho abriu precedentes para outros trabalhos com abordagem semelhante (VERONESE et al., 2011; CHUN et al., 2007; LEVIN et al., 2009; CORREIA; NEVES; VERISSIMO, 2004). Em 3.2, é feita uma discussão sobre alguns destes trabalhos.

Em (VERÍSSIMO, 2006) é discutido o modelo de *wormholes*. Este modelo é bi-modal com uma rede de *payload* e um subsistema *wormhole*. Em termos práticos, *wormhole* é um artefato privilegiado para ser usado somente quando necessário, e, supostamente, implementa funcionalidades difíceis de alcançar sobre o sistema de *payload*. O sistema de *payload*, por sua vez, deve executar a maior parte da atividades de computação e comunicação. O subsistema do *wormhole* segue um conjunto de suposições de falhas e sincronismo normalmente mais fortes do que suposições do subsistema de *payload*, tais como processamento e comunicação sendo síncronos, e comportamento faltoso de *crash* apenas.

2.3 TECNOLOGIA DE VIRTUALIZAÇÃO

O conceito de máquina virtual (VM - do inglês *Virtual Machine*) surgiu em meados da década de 1960 (GOLDBERG, 1974) e sua concepção inicial visava o particionamento lógico de *mainframes* em vários sistemas virtuais. A discussão teve retorno quando foram lançadas tecnologias de virtualização capazes de funcionar sob plataforma x86 (ROSENBLUM, 2004). Atualmente sistemas virtualizados são bastante aplicados até mesmo em computadores pessoais. Popek e Goldberg (1974) define máquina virtual como uma cópia isolada e eficiente de uma máquina real e, Parmelee (1972) como um sistema de computação no qual as instruções emitidas por um programa podem ser diferentes daquelas realmente executadas pelo hardware. Essas definições estão fortemente ligadas ao seu objetivo inicial de diminuir a subutilização de recursos de *hardware*, o que era comum nos antigos *mainframes*.

Processadores atuais já vem preparados para suportar virtualização. Aliando-se isto às capacidades computacionais oferecidas por eles, fez-se com que o mercado de virtualização de sistemas voltasseaa ser bastante atrativo (ROSENBLUM, 2004). As aplicações de virtualização tornaram-se mais abrangentes do que apenas otimizar os recursos oferecidos, vindo a ser aplicadas a outras áreas como gerenciamento de servidores de rede (PADALA et al., 2007) e segurança de sistemas computacionais (LAUREANO; MAZIERO; JAMHOUR, 2004; ROSENBLUM; GARFINKEL, 2005).

Existem duas categorias de virtualização, Smith e Nair (2005) as definiram como:

1. Máquina Virtual de Processo: é uma plataforma que executa um processo individual. Tal máquina virtual é instanciada unicamente para suporte ao processo, sendo criada e terminada juntamente ao processo. Exemplo: Java Virtual Machine (LINDHOLM; YELLIN, 1999).

2. Máquina Virtual de Sistema: provê um ambiente completo e persistente que suporta um sistema operacional completo e seus processos. Fornece ao sistema convidado um conjunto de recursos que compõem o hardware virtual. Exemplo: Xen (BARHAM et al., 2003).

Aqui discutiremos apenas a segunda categoria.

O processo ou sistema que é executado dentro da VM é chamado de convidado (*guest*), enquanto o sistema que suporta a VM é chamada de anfitrião (*host*) (SMITH; NAIR, 2005). Entre o *hardware* e o sistema operacional do convidado existe uma camada de abstração chamada de monitor da máquina virtual (VMM - *Virtual Machine Monitor*). O VMM ou *hypervisor* basicamente esconde os recursos físicos da plataforma computacional dos sistemas operacionais convidados (SAHOO; MOHAPATRA; LATH, 2010).

De acordo com Popek e Goldberg (1974) um VMM deve possuir as seguintes propriedades:

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

- Controle de Recursos: o VMM possui total controle sobre os recursos a serem oferecidos para as máquinas virtuais.

- Equivalência: qualquer programa K, executando com um VMM, deve apresentar comportamento idêntico à sua execução quando não há a presença do VMM.

Posteriormente, em (GARFINKEL; ROSENBLUM et al., 2003), foram descritas outras propriedades desejáveis:

- Isolamento: O VMM não deve ser acessível nem modificável por *softwares* executados em uma VM.

- Inspeção: Todo o estado das VMs deve estar disponível ao VMM.

- Interposição: A VMM deve intervir em determinadas operações realizadas por máquinas virtuais, como a execução de instruções privilegiadas.

Em (KING; DUNLAP; CHEN, 2003), os monitores de máquinas virtuais são categorizados como:

- Tipo I: Quando são executados diretamente sobre o *hardware*.
- Tipo II: Quando são executados sobre um sistema operacional anfitrião comum.

A maior vantagem dos VMMs do Tipo I em relação ao Tipo II está no desempenho, já que se interfaceamento é direto com o *hardware*. Entretanto, o Tipo II apresenta algumas vantagens sobre o Tipo I também. Como o VMM é executado como um processo comum do sistema operacional anfitrião, isto possibilita a utilização da computação do anfitrião para realização de tarefas de depuração e monitoramento do VMM e das VMs executando sobre ele. E é possível transformar o sistema convidado em uma *sandbox* do sistema anfitrião, fornecendo um ambiente seguro e restringindo a execução de certos códigos (KEAHEY; DOERING; FOSTER, 2004). É possível criar sistemas sobre o convidado de uma máquina virtual de comunicação em que o anfitrião fica invisível e inacessível para agentes externos.

O isolamento é uma importante propriedade ressaltada em (GARFINKEL et al., 2003). Um VMM permite que múltiplas aplicações sejam executada em diferentes máquinas virtuais. Cada máquina virtual tem seu próprio domínio de proteção de *hardware*, provendo um forte isolamento entre as máquinas virtuais. Isolamento seguro é essencial para fornecer confidencialidade e integridade.

Uma técnica interessante provida por várias tecnologias é a virtualização completa (LI; LI; JIANG, 2010). Nesta abordagem, códigos de *kernel* são traduzidos para substituir instruções por novas sequências de instruções que tem o efeito requerido no *hardware* virtual. O sistema convidado não tem conhecimento de estar virtualizado e não precisa ser modificado. O *hypervisor* simula várias instâncias completamente independentes de computadores virtuais possuindo seus próprios recursos virtuais. Ele traduz todas as instruções do sistema operacional em tempo real e armazena os resultados para futuras utilizações. Por disponibilizar os recursos virtuais, isso permite que a máquina virtual possa executar qualquer sistema operacional que seja suportado pelo *hardware* subjacente. A virtualização completa pode oferecer o melhor isolamento e segurança para máquinas virtuais (LI; LI; JIANG, 2010).

2.4 CONSIDERAÇÕES FINAIS

Neste capítulo foram abordadas as principais definições e conceitos que servem de base para o entendimento da área de sistemas distribuídos. Com estes conceitos torna-se mais simples o entendimento da proposta deste

trabalho.

Com este conceitos é possível entender o principais problemas da área, entretanto não foi ainda abordado Este capítulo lidou com conceitos básicos, sem considerar o estado da arte da área de tolerância a intrusão.

3 TRABALHOS CORRELATOS

Desde a formulação do Problema dos Generais Bizantinos por Lamport (LAMPORT; SHOSTAK; PEASE, 1982) (vide 2.2.1), muitos estudos surgiram com as mais variadas soluções para resolver o consenso bizantino. Os principais problemas que estes trabalhos tentam resolver é tolerar faltas bizantinas com baixa latência, melhorando vazãoe diminuindo a quantidade de recursos necessários (VERONESE et al., 2011; CASTRO; LISKOV, 1998; KOTLA et al., 2008; CLEMENT et al., 2009b; CORREIA; NEVES; VERISSIMO, 2004; CHUN et al., 2007). Dentre as técnicas utilizadas, destaca-se a replicação de máquinas de estados.

O uso da técnica que replicação por máquina de estados foi introduzida por Lamport (LAMPORT, 1978), antes mesmo da formulação do Problema dos Generais Bizantinos. Entretanto, neste trabalho, considerava-se que os sistemas propostos estavam livres da ocorrência de faltas. Mais tarde, em 1982, esta abordagem foi generalizada por Schneider (SCHNEIDER, 1982) considerando que o sistema modelado era suscetível a faltas de *crash*. O trabalho proposto por Schneider serviu de base para (REITER, 1995) e para uma variedade de outros modelos BFT.

A abordagem de RME tem sido utilizada para tolerar faltas bizantinas (arbitrárias) (REITER, 1995; CASTRO; LISKOV, 2002), mantendo o funcionamento correto do sistema ainda que tenha havido intrusões. A partir desta abordagem é possível projetar serviços confiáveis como sistemas de arquivos em rede, *backup* cooperativo, serviços de coordenação, autoridades certificadoras, banco de dados, sistemas de gerenciamento de chaves (CASTRO; LISKOV, 2002; YIN et al., 2003; AIYER et al., 2005; BESSANI et al., 2008; CLEMENT et al., 2009a; GARCIA; RODRIGUES; PREGUIÇA, 2011; REITER et al., 1996; ZHOU; SCHNEIDER; RENESSE, 2002).

Como foi citado, estes trabalhos focam em várias melhorias, dentre elas a diminuição de recursos para tolerar as intrusões, isto é, melhorar a relação entre o número de faltas toleradas e a resiliência do sistema (SOUSA; NEVES; VERISSIMO, 2005). A resiliência de um sistema tolerante a intrusão é dada pelo número mínimo de réplicas que o compõe. Na maioria dos algoritmos BFT são necessárias no mínimo de $3f+1$ réplicas (CASTRO; LISKOV, 2002; REITER, 1995) (KOTLA et al., 2008) para tolerar f faltosas. Em (YIN et al., 2003) foi mostrado que esse número de réplicas é necessário apenas para se atingir o acordo bizantino, mas, quando usada para tolerar faltas de *crash*, a redundância de máquinas de estado necessita apenas $2f+1$ réplicas (SCHNEIDER, 1990a). Considerando este fato, alguns trabalhos focam na separação do acordo bizantino da execução do serviço (YIN et al.,

2003; LUIZ et al., 2008)

Trabalhos recentes conseguem melhorar a resiliência dos sistemas BFT tolerando faltas com apenas $2f+1$ réplicas. Neste trabalhos é utilizada uma abordagem híbrida que considera diferentes suposições para diferentes partes do sistema. Em geral, nestes trabalhos, os sistemas possuem um componente (que poder ser uma rede controlada) com premissa de inviolabilidade, ficando sujeito apenas as faltas de *crash*, enquanto o resto do sistema está sujeito a faltas bizantinas (vide 2.2.3). Estes sistemas podem ser feitos utilizado várias abordagens, através do uso de um componentes de *hardware*, implementações que rodam internamente ao *kernel* do sistema operacional base, separação de redes utilizando mais de uma placa de comunicação, virtualização, etc (VERONESE et al., 2011; CHUN et al., 2007; LEVIN et al., 2009; CORREIA; NEVES; VERISSIMO, 2004; REISER; KAPITZA, 2007; JÚNIOR et al., ; STUMM et al., 2010).

Como nossa abordagem trata-se de um modelo com hipóteses híbridas e faz uso da técnica de replicação de máquinas de estado, neste capítulo subdividimos os trabalhos correlatos em duas seções: 3.1 abordagens homogêneas e 3.2 abordagens híbridas. Os trabalhos aqui apresentados mostram a evolução e o estado da arte da área de tolerância a faltas.

3.1 ABORDAGENS HOMOGÊNEAS

3.1.1 Practical Byzantine Fault Tolerance

Practical Byzantine Fault Tolerance (PBFT) (CASTRO; LISKOV, 1998) é um dos trabalhos pioneiros em apresentar uma solução prática para BFT baseada em replicação de máquina de estados.

Em PBFT um serviço é colocado à disposição para um conjunto de clientes e fica replicado em um conjunto de servidores. O conjunto de servidores precisa ser composto por, pelo menos, $3f+1$ réplicas sendo que f delas podem ser faltosas. Os servidores movem-se através de uma sucessão de configurações chamadas de visões. Em cada visão uma réplica do serviço é chamada de primária e as demais são seus *backups*. Assume-se que o modelo do sistema é parcialmente síncrono para que se garanta *liveness* em relação ao algoritmo. A autenticidade das mensagens trocadas pelo algoritmo é protegida com assinaturas baseadas em criptografia de chave pública ou através de condensações de mensagens produzidas com funções *hash* resistentes a colisões.

O algoritmo segue as seguintes fases:

1. Um cliente envia uma requisição de operação para todos os servidores;
2. O servidor primário envia para as réplicas a requisição em uma mensagem do tipo PRE-PREPARE;
3. Quando uma réplica r recebe uma mensagem do tipo PRE-PREPARE, r valida a mensagem e, ao aceitá-la, envia uma mensagem do tipo PREPARE;
4. Quanto uma réplica correta recebe $2f+1$ mensagens PREPARE, ela envia uma mensagem COMMIT para as demais réplicas;
5. Cada réplica que recebe $2f+1$ mensagens COMMIT aceita a ordem e executa a requisição;
6. O cliente aguarda por $f+1$ respostas iguais entre si, e de diferentes réplicas, para então aceitar o resultado.

A validação de uma mensagem PRE-PREPARE é feita (i) através da comparação entre a sua assinatura e a assinatura da mensagem do cliente e (ii) se a mensagem foi criada pela réplica primária da visão. As fases PRE-PREPARE e PREPARE servem para assegurar a ordem total das requisições, mesmo que a primária seja faltosa. As fases PREPARE e COMMIT são usadas para garantir que os pedidos são totalmente ordenados através das visões.

A réplica primária define a ordem em que as requisições dos clientes serão executadas por todas as réplicas. Quando existe suspeita de que a primária está faltosa é executado o protocolo de mudança de visão para garantir a propriedade *liveness*, permitindo o progresso do sistema. As mudanças de visão são disparadas por *timeouts* que previnem que as réplicas fiquem esperando indefinidamente para executar as requisições. Periodicamente as réplicas trocam mensagens de CHECKPOINT com uma prova do seu estado atual. Quando uma réplica r recebe $2f+1$ mensagens de CHECKPOINT com o mesmo estado e assinadas por diferentes réplicas, r entende que $2f+1$ réplicas progrediram e descarta todas as mensagens com requisições anteriores.

Ao suspeitar que a réplica primária esteja corrompida, cada réplica *backup* envia uma mensagem VIEW-CHANGE para todas as réplicas, contendo o identificador da próxima visão $(v+1)$, número de sequência n do último *checkpoint* estável, conjunto de $2f+1$ mensagens válidas de *checkpoint* para provar a validade de n, um conjunto com todas as mensagens que, antes do início da mudança de visão, estavam com a fase PREPARE completa e que possuem um número de sequência maior que n. Ao receber $2f$ mensagens VIEW-CHANGE, o novo primário (calculado por $p=(v+1)mod||R||$), emite uma mensagem NEW-VIEW para as réplicas restantes, composta pelo

número da nova visão, o conjunto de mensagens de VIEW-CHANGE recebidas, para provar a veracidade da mudança de visão, e uma mensagem PRE-PREPARE para cada mensagem cujo processamento foi interrompido pela mudança de visão. Ao receber uma mensagem de NEW-VIEW, as réplicas realizam a troca de visão, desde que a mensagem esteja correta.

O PBFT tem grande destaque por ser um dos modelos práticos pioneiros na área de tolerância a faltas bizantinas. Vários outros trabalhos derivam de seu modelo (KOTLA et al., 2008; CLEMENT et al., 2009b). Apesar da quantidade de passos e de réplicas necessárias para sua execução terem sido reduzidas em trabalhos mais recentes, o PBFT é considerado o estado da arte na área de tolerância a faltas bizantinas através da abordagem de replicação de máquina de estados.

3.1.2 Zyzzyva

Zyzzyva, apresentado em (KOTLA et al., 2008), é um algoritmo de replicação de máquinas de estado tolerante a faltas bizantinas que explora especulação para reduzir o *overhead* em protocolos de consenso. No Zyzzyva as réplicas, de maneira otimista, adotam o número de ordenação proposto por uma réplica primária e respondem imediatamente para o cliente, sem entrar em acordo sobre a ordem das requisições. Em função disso, as réplicas podem ficar inconsistentes quando a primária é faltosa. Nestes casos, o cliente detecta estas inconsistências e auxilia as réplicas a atingirem a ordem total das requisições.

Em sua execução o Zyzzyva necessita de $3f+1$ réplicas divididas em visões. Em cada visão existe uma primária e as demais réplicas servem apenas de *backup* para ela. A operação na ausência de faltas ocorre em quatro passos: (1) o cliente envia uma requisição para a primária, (2) a primaria ao receber a requisição designa um número de ordenação para a mesma e envia junto com a requisição para todas as réplicas pertencentes à sua visão; (3) as réplicas recebem a requisição com o número de ordenação e, de maneira especulativa, executam e enviam para o cliente as repostas, (4a)[1] o cliente verifica se existem $3f+1$ respostas iguais entre si e, em caso positivo, aceita a resposta.

As respostas dos servidores incluem o resultado da requisição e um histórico com a sequência de todas as requisições executadas anteriormente incluindo a qual o resultado se refere. O cliente espera por um tempo determinado pelas respostas. Se o cliente receber entre $2f+1$ e $3f$ respostas iguais

[1]Usamos aqui a mesma sequência de passos apresentada no artigo original, onde o passo 4 pode tomar quatro formas diferentes.

entre si então ele executa o passo (4b) ao invés de (4a), onde são reenviadas as respostas para as réplicas. Este conjunto de mensagens representa um certificado de comprometimento que prova que $2f+1$ réplicas concordaram com a ordem em que as requisições foram executadas. O passo (5) é referente ao recebimento do certificado pelas réplicas. No passo (6) o cliente aguarda que $2f+1$ réplicas retornem que receberam o certificado de comprometimento, para então aceitar o resultado.

Caso o cliente receba menos de $2f+1$ respostas iguais entre si, o protocolo avança para o passo (4c) ao invés do (4a). Nesta etapa o cliente reenvia o seu pedido para todas as réplicas, que encaminham a requisição para a primária, a fim de garantir que o será atribuído um número de ordem para que a mesma seja executada. Se após um intervalo de tempo a requisição não for ordenada as réplicas irão suspeitar da primária e vão, em algum momento, efetuar uma mudança de visão trocando de primária.

Caso o cliente receba respostas indicando ordenação inconsistente pela primária, ele envia uma prova de mau comportamento para as réplicas, que iniciam uma mudança de visão para trocar a primária. Este passo é o (4d) e ocorre no lugar do (4a).

Como os cliente auxiliam na convergência para a ordem total, algumas réplicas precisam fazer *rollback* de algumas requisições já executadas. Para tanto, existe um mecanismo de *checkpoint* que guarda informações sobre o estado em cada réplica. Este mecanismo ajuda também a reduzir o custo para troca de visão.

Conclusão

Ao explorar sistematicamente as especulações, Zyzzyva apresenta melhorias significativas de desempenho em relação aos protocolos BFT já existentes. O *overhead* de vazãoe latência do Zyzzyva é aproximado dos limites teóricos mais baixos para qualquer protocolo de replicação de máquina de estados BFT. Porém, o mecanismo de *rollback* não pode ser utilizado em alguns tipos de aplicações, como por exemplo sistemas com transações bancárias. Além do mais, se houver na rede utilizada heterogeneidade, na latência, isto pode afetar o desempenho do algoritmo, já que a variação no tempo de resposta para um cliente pode fazer com que o mesmo reenvie sua requisição para todas as réplicas.

3.1.3 Separando o acordo da execução

3.1.3.1 Separating agreement from execution for byzantine fault tolerant services

Em (YIN et al., 2003) é proposto um algoritmo de replicação tolerante a faltas bizantinas que endereça dois problemas que, segundo os autores, limitam o uso de algoritmos BFT em larga escala. Primeiro deles é que, mesmo existindo algoritmos que melhoram integridade e disponibilidade, o comprometimento de uma simples réplica é o bastante para pôr em risco a confidencialidade. E o segundo é que algoritmos BFT requerem $3f+1$ réplicas, o que é um custo significativo mesmo com diminuição dos custos de *hardware*.

O princípio chave desta arquitetura é separar o acordo da execução. Replicação de máquinas de estado primeiro entram em acordo sobre a ordem de execução das requisições para só depois executá-las. O sistema precisa de $3f+1$ réplicas para o consenso de ordenação, entretanto para a execução das requisições são necessárias apenas $2f+1$ réplicas. Essa distinção é crucial, pois para a execução de tarefas espera-se que seja necessário muito mais recurso computacional do que para o acordo.

Os servidores são divididos em *cluster* de acordo e *cluster* de execução. O cliente envia uma requisição para o *cluster* de acordo que executa o algoritmo PBFT para obter um certificado de acordo que define a ordem de execução da requisição. Na sequência, o *cluster* de acordo encaminha a requisição do cliente juntamente com o certificado de acordo para o *cluster* de execução. O *cluster* de execução implementa uma máquina de estados de aplicação específica para processar as requisições de acordo com a ordem determinada pelo *cluster* de acordo.

A separação do acordo e execução leva a uma arquitetura de *firewall* privada para proteger a confidencialidade através de replicação bizantina. Em arquiteturas de replicação de máquinas de estado existentes, a operação de votação é executada pelos clientes que esperam por $f+1$ respostas iguais entre si para aceitar o resultado. Este tipo de arquitetura permite que um cliente malicioso observe informação confidencial que servidores faltosos deixaram vazar, e isto não pode ocorrer se a confidencialidade for um requerimento. Os autores propõem um conjunto redundante de nodos de *firewall* privado para restringir a comunicação dos nodos, filtrando respostas incorretas antes que elas sejam devolvidas para os nodos de acordo ou até mesmo o cliente. O *firewall* privado atua entre o *cluster* de execução e os clientes, passando apenas informações enviadas por servidores de execução corretos. O sistema restringe comunicação fazendo com que (1) nodos de *firewall* sejam conec-

tados apenas a nodos diretamente acima e abaixo deles e (2) requisições e respostas sejam criptografadas. Com estas restrições, a comunicação sempre passará por pelo menos um *firewall* correto e garante que o corpo das mensagens ficará protegido contra leituras indevidas. O uso do *firewall* aumenta a confidencialidade, mas diminui a performance de maneira considerável.

Conclusão

Este trabalho tem grande destaque na área por mostrar que o custo adicional para se tolerar faltas bizantinas reside na necessidade de se atingir o acordo bizantino. Quando esta distinção é colocada em prática em um modelo de replicação de máquina de estados, o resultado é a queda do custo computacional, já que as máquinas que são utilizadas para o consenso necessitam de menos processamento que aquelas usadas na execução das requisições.

3.1.3.2 Espaço aumentado de tuplas e protegido por políticas

Em (LUIZ et al., 2008) os autores, motivados pelo progresso nos estudos na área de tolerância a faltas com uso de replicação de máquinas de estados, apresentam RePEATS (*Replication over Policy-Enforced Augmented Tuple Space*).

Os autores sugerem uma solução para tolerância a faltas bizantinas através da abordagem de replicação de máquina de estados (RME), que combina uma série de mecanismos que contribuem para a manutenção da disponibilidade e integridade das aplicações, bem como dos ambientes de execução.

REPEATS (*Replication over Policy-Enforced Augmented Tuple Space*) é uma arquitetura de RME tolerante a faltas bizantinas fundamentada no modelo PEATS (*Replication over Policy-Enforced Augmented Tuple Space*) (BESSANI et al., 2009), onde os processos (tanto clientes quanto as réplicas do serviço) coordenam-se através de uma abstração de alto nível: um espaço de tuplas resistente a faltas bizantinas.

O uso desta abstração permite também a separação das entidades responsáveis pelo acordo, implementado pelo espaço de tuplas, daquelas responsáveis pela execução das requisições enviadas pelos clientes, com a vantagem de se ter algoritmos de replicação modulares e muito mais simples.

O modelo de execução requer apenas $2f+1$ réplicas para um serviço, e é genérico o suficiente para comportar diversos conjuntos de serviços (com diferentes aplicações) compartilhando o mesmo suporte de comunicação e coordenação (o espaço de tuplas), de modo que as particularidades de um serviço não interferem nas outras.

O espaço de tuplas é uma abstração de memória compartilhada útil para a coordenação de processos, bem como para o armazenamento de da-

dos. Esta abstração é oriunda do modelo de coordenação generativa e teve sua primeira implementação na linguagem LINDA. No espaço de tuplas é possível realizar o armazenamento e a recuperação de estruturas de dados genéricas sob a forma de tuplas. Uma tupla $t = (f1, f2, ..., fn)$ é composta por uma sequência de campos. Um campo fi de uma tupla pode conter um valor definido, um formal(variável) ”?”ou ainda um símbolo especial ”*”. Um campo formal é usado para extrair conteúdos individuais dos campos de uma tupla, já os símbolos especiais são usados para representar campos sem valor definido. Uma tupla t cujos campos têm valores definidos é denominada de entrada. Uma tupla que possui algum campo formal e/ou um campo especial é denominada molde, e é representada por t. Um molde t combina uma entrada t se ambas as tuplas têm o mesmo número de campos e todos os campos com valores definidos de t contém o mesmo valor do campo correspondente em t. Por exemplo, uma tupla $[”RePEATS”, 2008]$ combina com os moldes $[”RePEATS”, *], [*, 2008]$ e $[*, *]$ mas não com $[*, 2007]$.

O modelo clássico de coordenação por espaço de tuplas não provê mecanismos capazes de lidar com processos maliciosos acessando o espaço de tuplas. Este problema foi resolvido com a introdução do PEATS, que consiste em um espaço de tuplas onde as interações entre os processos são reguladas por políticas de acesso de granularidade fina. Estas políticas, que são usadas como mecanismo de controle de acesso ao espaço de tuplas, são compostas por um conjunto de regras padrão para a invocação de operações no espaço de tuplas e condições que devem ser satisfeitas para que estas invocações possam ser executadas, ou negadas. Para isto, o PEATS considera os dados advindos da invocação (o processo invocador e os parâmetros da invocação) e o estado atual do espaço.

O RePEATS consiste em uma concretização de replicação de máquina de estados tendo como elemento de comunicação entre os processos envolvidos em um PEATS, dando origem então a uma arquitetura de suporte à replicação tolerante a faltas bizantinas.

Os clientes inserem suas requisições na forma de tuplas no PEATS e as réplicas do serviço que está sendo acessado leem estas tuplas do PEATS para obter as requisições a serem executadas. Em seguida, os serviços replicados processam as requisições e enviam os resultados também na forma de tuplas dentro do PEATS, para que os clientes possam obter as respostas.

Uma premissa do RePEATS é o determinismo de réplica. Este requisito define que réplicas partindo de um mesmo estado inicial e sujeitas à execução de uma mesma sequência de operações, devem chegar ao mesmo estado final. Assim, em um sistema onde as réplicas implementam um serviço determinista, esta propriedade é implementada por meio do uso de protocolos de difusão com ordem total que garantem que todas as operações enviadas

ao sistema são processadas por todas as réplicas (acordo) em uma mesma ordem (ordem total). A partir daí cada réplica executa a operação, atualiza seu estado e envia ao cliente o resultado obtido. O cliente aceita o resultado da operação caso receba $f+1$ respostas iguais de diferentes réplicas considerando f o número máximo de servidores faltosos. O PEATS implementa o algoritmo de difusão com ordem total.

O controle de acesso é o mecanismo que permite que o RePEATS seja tolerante a faltas. Este controle de acesso se dá através das políticas de granularidade fina suportadas pelo PEATS. Quando uma operação é invocada no PEATS, as regras especificadas nestas políticas são verificadas tomando como base o identificados do processo que invoca a operação, a operação que está sendo invocada e o estado atual do espaço de tuplas para negar ou permitir a execução da operação.

O funcionamento do algoritmo no lado do cliente é iniciado quando um cliente deseja enviar um comando C qualquer. Este cliente tenta inserir uma tupla REQUEST no espaço com um número de sequência igual ao seu último acrescido de uma unidade, sendo que seu valor inicial é zero. A chamada *cas* insere esta tupla caso já não esteja inserida, caso contrário o cliente incrementa seu número de sequência e tenta novamente. Após inserir a tupla, o cliente fica em modo espera aguardando por $f+1$ respostas iguais entre si vindas de réplicas diferentes.

O algoritmo no lado do servidor também é iniciado com o valor zero para o número de sequência. As réplicas processam as tuplas REQUEST respeitando a ordem ascendente de chegada, isto é, são processadas primeiro as requisições com número de sequência mais próximo daquele que foi processado na última execução. Ao processar a requisição do cliente o servidor, por questões de desempenho, envia a resposta diretamente ao cliente que fez a requisição.

A política de acesso do PEATS utilizada pelos autores serve para evitar que processos maliciosos quebrem a ordem total, inserindo tuplas REQUEST fora do intervalo de sequência no espaço. Para isso, a inclusão de requisições só pode ser efetuada através da instrução *cas*, na condição de que a tupla REQUEST com número sequencial anterior ao que está sendo incluído esteja presente no espaço. A operação *rd* é permitida somente para tuplas REQUEST, desde que os campos 3 e 4 do molde sejam formais.

Para correção do protocolo, algumas premissas são admitidas: (i) cada requisição do cliente tem um identificador único e crescente; (ii) o cliente só envia uma requisição após ter recebido a resposta da requisição anterior; (iii) um temporizador é associado a cada requisição enviada, e caso ocorra um *timeout* e a resposta da requisição ainda não tenha sido obtida, o cliente reenvia a requisição.

A abordagem traz também um algoritmo de *checkpointing*. Sua função é guardar o estado das réplicas corretas do serviço no espaço de tuplas a cada N requisições executadas, sendo N um parâmetro configurável e igual em todas as réplicas corretas do serviço. No entanto alguns cuidados são tomados para que réplicas maliciosas não criem *checkpoints* com estados incorretos.

Como as requisições são armazenadas no espaço de tuplas, os autores exploraram esta facilidade para fins de definição de um mecanismo de *logging*. Este mecanismo é necessário para a recuperação das réplicas que venham a falhar. As requisições perduram no espaço ate o momento da gravação de um *checkpoint* posterior, que sinaliza que as requisições anteriores não são mais necessárias durante o processo de recuperação de réplicas. Deste modo, para a recuperação pontual de uma réplica, o processo restaura os *checkpoint* necessários e se houver requisições após o último *checkpoint*, estas são recuperadas diretamente do espaço.

Os autores apresentaram uma arquitetura para replicação tolerante a faltas bizantinas baseada no modelo de coordenação por espaço de tuplas cuja contribuição é importante por conseguir uma configuração que diminui a quantidade de recursos para atender uma requisição para $2f+1$, apesar da necessidade de um espaço de tuplas com $3f+1$ recursos. Como um mesmo espaço de tuplas pode ser utilizado para uma grande variedade de serviços a quantidade de recursos total se aproxima de $2f+1$.

3.2 ABORDAGENS HÍBRIDAS

Nesta seção iremos discutir apenas os trabalhos que usam componentes ou redes invioláveis. Eles possuem mais de um subsistema com premissas de tolerância a faltas e sincronismo diferentes (vide 2.2.3).

3.2.1 Attested append-only memory: Making adversaries stick to their word

A abstração de registro confiável *Attested Append-Only Memory* (A2M) (CHUN et al., 2007) foi idealizada para ser pequena, de fácil implementação e consequentemente muito verificável. Um registro A2M oferece métodos para anexar e buscar valores dentro de um registro. O A2M também provê um método para se obter o fim do registro e fazer avançar o sufixo armazenado na memória, utilizado para saltar por múltiplos números sequenciais. Não existem métodos para substituir os valores que já foram atribuídos.

Algoritmos tolerantes a faltas bizantinas têm que lidar com servidores

faltosos que podem prover informações falsas ou inconsistentes de inúmeras maneiras para diferentes clientes ou servidores. O A2M foi especialmente concebido para restringir esse tipo de comportamento. Ele atenua os efeitos de falhas bizantinas nos componentes não confiáveis, baseando-se no histórico de operações providas pelo A2M, que não pode ser violado.

O A2M foi aplicado no algoritmo PBFT (*Practical Byzantine Fault Tolerance*) (CASTRO; LISKOV, 1998) com o intuito de reduzir o número de réplicas de $3f+1$ para $2f+1$, originando o algoritmo A2M-PBFT-EA. Cada réplica foi equipada com o A2M e todas as mensagens trocadas entre as réplicas foram anexadas a registros A2M antes de serem enviadas às outras réplicas. O A2M fornece atestados para proteger as mensagens contra ataques de integridade e as torna não repudiáveis. Com base nisto, certificados para requisições preparadas (*PREPARE*), efetivadas (*COMMIT*), para mudanças de visão e *CHECKPOINT*, em A2M-PBFT-EA, tem tamanho $f+1$ contra $2f+1$ do PBFT.

Para a implementação do A2M-PBFT-EA foram necessários cinco arquivos de *log*: PREPARE (que também contém PRE-PREPARE) e COMMIT para os três passo do acordo, CHECKPOINT para a coleta de lixo (*garbage collection*), e VIEW-CHANGE e NEW-VIEW para mudanças de visão.

O facilitador para sistemas confiáveis A2M provê uma abstração de programação de registro confiável onde é possível criar protocolos imunes a faltas. Através da utilização do A2M é possível produzir variações do trabalho de Castro e Liskov (CASTRO; LISKOV, 1998) para tolerância a faltas bizantinas via replicação de máquina de estados. Com A2M o sistema mantém suas propriedades de *liveness* e corretude mesmo que metade das réplicas estejam faltosas. Esta abordagem é de fácil entendimento e conseguiu, também, atingir a melhor resiliência prática. Entretanto, sua implementação requer o gerenciamento de cinco *logs*, fazendo com que em sua aplicação prática o A2M precise de mais armazenamento, tornando-o mais complexo do que seus autores assumiram.

3.2.2 Componente inviolável através do uso de *Hardware*

O ponto chave por trás do A2M foi a observação de que uma única propriedade em faltas bizantinas é responsável pela necessidade de $3f+1$ réplicas para tolerar faltas bizantinas. Esta propriedade é a equivocação (CHUN et al., 2007), que significa a capacidade de fazer declarações contraditórias por diferentes participantes. Nos últimos anos algumas soluções alternativas foram introduzidas para prevenir a equivocação, o que possibilita a redução do número de réplicas em sistemas tolerantes a faltas bizantinas, indo

de $3f+1$ para $2f+1$ réplicas (CHUN et al., 2007; CORREIA; NEVES; VERISSIMO, 2004). Em (LEVIN et al., 2009) é mostrado que é suficiente um contador monotônico crescente para se conseguir um subsistema confiável. Nesta abordagem, o subsistema assina, de maneira segura, um valor único de contagem para cada mensagem e garante que este valor nunca será atribuído a outra mensagem diferente. Assim, quando uma réplica recebe uma mensagem, ela sabe com certeza que nenhuma outra mensagem com conteúdo diferente possui este número de contagem. Como cada réplica não faltosa valida que a sequência de valores do contador de mensagens recebidas de outra réplica não contém lacunas, réplicas maliciosas não podem criar equívocos nas mensagens. Este contador confiável foi utilizado para construir A2M, a partir do qual um sistema BFT com $2f+1$ réplicas foi alcançado.

3.2.2.1 CheapBFT: Resource-efficient Byzantine Fault Tolerance

Baseando-se na mesma ideia de contador confiável, em (KAPITZA et al., 2012) os autores apresentam um sistema que possui um contador confiável que assegura que um valor de contagem nunca será atribuído para duas mensagens diferentes. Com base neste contador propõe-se uma replicação de máquina de estados passiva para criar um BFT chamado CheapBFT.

Nesta abordagem, cada máquina é equipada com um subsistema CASH (*Counter Assignment Service in Hardware*) que é inicializado com uma chave secreta e identificado unicamente com um id de subsistema que corresponde à réplica que o hospeda. A chave secreta é compartilhada entre os subsistemas de todas as réplicas. Além da chave secreta, o estado interno de um subsistema assim como o algoritmo utilizado para autenticar mensagens precisa ser conhecido publicamente

CASH possui duas funções, a primeira (createMC) é utilizada para criar certificados e outra para verificar (checkMC) a validade dos certificados. Quando a função createMC é chamada com uma mensagem *m*, ela incrementa o valor do seu contador local e usa a chave secreta *K* para gerar um MAC (*Message Authentication Code*) *b* que cobre o *id* do subsistema local *S*, o valor corrente do contador *c*, e a mensagem. O certificado *mc* é criado com *S*, *c* e *b* anexados a ele. Para atestar o certificado gerado por outro subsistema *s*, a função checkMC verifica o MAC e usa a função isNext() para validar que não existem lacunas na sequência do outro subsistema. A função isNext() guarda os últimos valores de contagem de todos os subsistemas.

O algoritmo CheapBFT usa apenas $f+1$ réplicas em caso normal (CheapTiny), as demais réplicas são passivas. As réplicas ativas participam do estágio de acordo e do estágio de execução, enquanto as réplicas passivas

apenas recebem atualizações de estado das réplicas ativas.

Estágio de acordo. Ao iniciar o protocolo, um conjunto de $f+1$ réplicas ativas é selecionado de forma determinística. A réplica ativa com o menor id se torna o líder. À semelhança de outros protocolos de acordo inspirados no PBFT, o líder em CheapTiny é responsável por propor a ordem de execução das requisições. Quando todas as $f+1$ réplicas ativas aceitam o valor proposto, o pedido torna-se efetivado (*commited*) e pode ser processado de forma segura. Ao receber a mensagem m do cliente, o líder verifica sua autenticidade e a envia numa mensagem PREPARE à todas as réplicas, juntamente com a mensagem de certificação mc emitida pelo subsistema CASH. As réplicas ativas, ao receberem a mensagem de PREPARE, verificam se a mensagem do cliente é autentica. Se a mensagem do líder é autentica e se o valor de ordenação não contém lacunas. Se estiver tudo correto, a réplica emite um certificado para a mensagem do líder e o envia numa mensagem COMMIT juntamente com os parâmetros recebidos na mensagem PREPARE. Quando uma réplica ativa recebe uma mensagem COMMIT ela verifica a autenticidade. Ao receber $f+1$ mensagens COMMIT corretas para a mensagem m a réplica encaminha a requisição para o estágio de execução.

Estágio de execução. O processamento de uma requisição m no CheapBFT requer que a aplicação forneça dois objetos, a resposta r para o cliente e uma atualização de estado u que reflete as mudanças no estado da aplicação em função da execução de m. Ao processar uma requisição, uma réplica ativa solicita ao CASH que crie um certificado de atualização u_c para r, u e o conjunto de COMMITs C confirmando que houve comprometimento com m. Em seguida, a réplica envia uma mensagem de UPDATE para todas as réplicas passivas e entao envia a resposta para o cliente. Uma réplica passiva atualiza seu estado quando recebe $f+1$ mensagens de UPDATE vindas das réplicas ativas e verificadas como corretas para uma mesma resposta.

Conclusão

CheapBFT é o primeiro sistema tolerante a faltas bizantinas que, apesar de necessitar de $2f+1$ réplicas, utiliza apenas $f+1$ tanto para o acordo quanto para a execução. É um importante passo para a área de tolerância a intrusão. O único porém é a necessidade de se criar o componente em *hardware*, que, mesmo com a diminuição de um recurso do ponto de vista computacional, torna mais complexa sua viabilização prática.

3.2.2.2 Efficient Byzantine Fault Tolerance

Em (VERONESE et al., 2011) os autores, através de melhorias em trabalhos anteriores, criam dois algoritmos tolerantes a faltas bizantinas (BFT).

O artigo mostra como melhorar os trabalhos BFT e Zyzzyva considerando-se três métricas para isto: número de réplicas, simplicidade de serviço confiável e número de passos de comunicação. Os autores afirmam que os algoritmos são eficientes por serem tão bons ou melhores que os anteriores, levando-se em conta as mesmas métricas.

Número de réplicas. Geralmente os algoritmos de BFT requerem $3f+1$ réplicas para tolerar f servidores bizantinos. Com o uso de um componente inviolável, os autores foram capazes de diminuir essa resiliência para $2f+1$.

Simplicidade de serviço confiável. Trabalhos anteriores mostram que é possível reduzir o número de réplicas de $3f+1$ para $2f+1$ estendendo os servidores com componentes invioláveis, isto é, com componentes que fornecem um serviço correto mesmo que os servidores a que pertencem sejam faltosos. Portanto, um aspecto importante para chegar a $2f+1$ é a arquitetura destes componentes invioláveis. Um objetivo fundamental é fazer com que o componente seja verificável, o que requer simplicidade. A eficiência do algoritmo proposto é também baseada na simplicidade do componente inviolável se comparada às propostas anteriormente (TTCB (CORREIA; NEVES; VERISSIMO, 2004), A2M *Attested Append-Only Memory*).

Número de passos de comunicação. É uma importante métrica para algoritmos distribuídos, pelo fato do atraso na comunicação tender a ter mais impacto na latência do algoritmo. O primeiro algoritmo proposto - MinBFT - segue um padrão de troca de mensagens similar ao PBFT. As réplicas se movimentam através de uma sucessão de configurações chamadas de visões. Cada visão tem uma réplica primária e as demais são apoio (*backups*). Quando algumas réplicas suspeitam que a primária é faltosa, é feita a troca de primária, permitindo o progresso do sistema. Em cada visão existem passos de comunicação onde a primária envia mensagens para todas as réplicas de apoio, e existem passos em que todas as réplicas enviam mensagens entre si. A ideia fundamental do MinBFT é a utilização de um contador, por parte da primária, para assinar números de sequência para as requisições de clientes. Porém, mais do que assinar números, o componente inviolável gera um certificado que prova de maneira inequívoca que o número assinado pertence apenas aquela mensagem.

O segundo algoritmo proposto - MinZyzzyva - é baseado em especulação, isto é, na tentativa de execução de requisições de clientes sem acordo inicial sobre a ordem de execução. MinZyzzyva é uma versão modificada de Zyzzyva, o primeiro algoritmo BFT especulativo. O padrão de comunicação do Zyzzyva é similar ao PBFT, exceto pela especulação: quando as réplicas de apoio recebem uma requisição da primária, de maneira especulativa executam a requisição e enviam uma resposta ao cliente.

O *Unique Sequential Identifier Generator* (USIG) é um serviço local que existe em todos os servidores. O serviço é responsável por fornecer o valor do contador para mensagens e por assinar esta mensagem ao valor passado. Os identificadores são únicos, monotônicos, e sequenciais para o servidor. Estas três propriedades garantem que o USIG (1) nunca irá assinar o mesmo identificador para duas mensagens distintas, (2) nunca assinará um identificador menor que o anterior, e (3) nunca assinará um identificador que não é o sucessor do anterior. Estas propriedades são garantidas mesmo que o servidor esteja comprometido, fazendo com que o serviço tenha que ser implementado em um componente inviolável. A interface do serviço tem duas funções:

- $createUI(m)$ - retorna um certificado USIG que contém um identificador único e a certificação de que este identificador foi criado pelo componente inviolável para a mensagem m. O identificador é a leitura do contador monotônico, que é incrementado sempre que a função $createUI$ é chamada.

- $VerifyUI(PK, UI, m)$ - verifica se o identificador único (UI) é válido para a mensagem m, isto é, se o certificado USIG foi gerado através da mensagem e demais dados em UI.

Existem duas maneira de se implementar o serviço:

- USIG-Hmac: um certificado contém um código de autenticação de mensagem baseado em *hash* (Hmac - acrônimo do inglês) obtido através da mensagem e da chave secreta do USIG. A chave de cada USIG é conhecida pelos demais USIG, assim todos são capazes de verificar os certificados gerados.

- USIG-sign: o certificado contém uma assinatura obtida usando a mensagem e a chave privada do USIG.

No USIG-Hmac as propriedades do serviço são baseadas na chaves compartilhadas. Em USIG-Sign as propriedades são baseadas nas chaves privadas. Para o USIG-Hmac ambas as funções $createUI$ e $verifyUI$ precisam ser implementadas dentro do componente inviolável. No USIG-Sign a verificação necessita apenas da chave pública do USIG que criou o certificado, em função disso, esta operação pode ser executada fora do componente. Em ambas as implementações, as chaves precisam ser compartilhadas para que as verificações sejam executadas.

Os autores implementam o serviço confiável USIG através do *chip* TPM (acrônimo do inglês para *Trusted Platform Module*). Isto é, a inviolabilidade do USIG se dá pelo fato do mesmo estar implementado no *hardware* de cada máquina servidora.

O MinBFT é um algoritmo não especulativo $2f+1$ que segue um padrão de troca de mensagens similar ao PBFT. Na operação normal a sequência de eventos do MinBFT é a seguinte: (1) o cliente envia uma requisição para todos os servidores; (2) a réplica primária assina o número de sequência para a requisição e envia para todos os servidores numa mensagem *PREPARE*; (3) cada servidor difunde uma mensagem de *COMMIT* para os demais, assim que recebe um *PREPARE* da primária; (4) quando um servidor aceita uma requisição, ele executa a operação correspondente e retorna uma resposta para o cliente; (5) o cliente espera por $f+1$ respostas iguais para a requisição e completa a operação. Quando $f+1$ réplicas de apoio suspeitam que a primária esteja faltosa, uma mudança de visão é executada, e um novo servidor se torna o primário. Este mecanismo garante *liveness* permitindo que o sistema faça progresso quando a primária é faltosa.

O MinZyzzyva tem características similares ao MinBFT, entretanto tem o número de passos de comunicação reduzidos em uma execução de caso normal por ser especulativo. A ideia da especulação é que os servidores respondem para os cliente sem primeiro concordar sobre a ordem em que as requisições devem ser executadas. Os servidores adotam, de maneira otimista, a ordem proposta pelo servidor primário, executam a requisição, e respondem imediatamente ao cliente. Esta execução é especulativa porque pode não ser a ordem real em que a requisição deveria ser executada. Se alguns servidores tornam-se inconsistentes em relação aos outros, os clientes detectam a inconsistência e ajudam os servidores a convergirem numa única ordem total de requisições, possivelmente tendo que fazer o *rollback* de algumas execuções. Os clientes só confiam nas respostas que estiverem consistentes com esta ordem total. O MinZyzzyva usa o serviço USIG para restringir o comportamento da réplica primária, permitindo a redução do número de réplicas de $3f+1$ para $2f+1$, preservando as propriedades *safety* e *liveness*. As réplicas *backup* só aceitam uma requisição repassada pela primária se o identificador único gerado pelo USIG for válido e se a mensagem estiver respeitando a ordem FIFO (o primeiro que entra é o primeiro que sai).

Os algoritmos MinBFT e MinZyzzyva melhoram algoritmos anteriores requerendo apenas $2f+1$ ao invés $3f+1$ réplicas. O algoritmo é muito simples e usa o *chip* TPM como componente inviolável para implementar o serviço USIG que designa números de ordenação para as mensagens vindas dos clientes. Assim, toda réplica correta executa as requisições na ordem designada pelo USIG. A contribuição deste trabalho vêm em termos de custo, resiliência e complexidade de gerenciamento, ficando mais próximo dos algoritmos de replicação tolerantes a faltas catastróficas. Entretanto, para a criação do serviço inviolável são necessárias modificações na máquina servidora que precisa possuir o *chip* TPM, em nível de *hardware*, tornando sua

implementação mais complexa do que utilizar-se de tecnologias que estão disponíveis como virtualização e memória compartilhada emulada.

3.2.3 Abordagens com virtualização

Um parâmetro importante endereçado pela maioria das abordagens BFT da literatura é a quantidade de réplicas necessárias para tolerar f faltosas, também conhecido como resiliência. Muitas soluções BFT estão limitadas ao problema do consenso bizantino, necessitando de $3f+1$ réplicas para executar o consenso (CASTRO; LISKOV, 2002; REITER, 1995; KOTLA et al., 2008). Trabalhos recentes conseguiram atingir um mínimo de $2f+1$ réplicas. A diminuição da quantidade de réplicas é obtida de várias maneiras, abstração de *hardware*, tecnologia de virtualização e/ou canais confiáveis. O uso da tecnologia de virtualização, além de facilitar na melhoria da resiliência, abre muitas possibilidades de melhorias, como o uso de imagens de sistema para regeneração do mesmo em caso de detecção de faltas, redundância de *hardware*, migração de *hardware* em caso de falta do equipamento, maior controle de acesso a rede da réplica, menor quantidade de recursos físicos, dentre outras (CULLY et al., 2008; REISER; KAPITZA, 2007, 2008; CHUN; MANIATIS; SHENKER, 2008; WOOD et al., 2008) Em nosso trabalho optamos por utilizar esta tecnologia para aproveitar da sua propriedade de isolamento entre as réplicas e as máquinas físicas. Desta maneira, podemos obter uma parte do sistema com segurança de acesso. Nesta seção trazemos algumas das abordagens de virtualização importantes para a área.

As abordagens que se baseiam em virtualização em sua maioria utilizam para diminuir a quantidade de máquinas físicas e/ou para fazerem uso do isolamento que o *hypervisor* provê.

3.2.3.1 VM-FIT: Supporting intrusion tolerance with virtualisation technology

Virtual Machine - Fault and Intrusion Tolerance (VM-FIT) apresentado em (REISER; KAPITZA, 2007) foi um dos primeiros trabalhos da literatura a apresentar uma solução baseada em máquinas virtuais.

Em sua primeira versão, Redundant Execution on Single Host (RESH), os autores se aproveitaram da arquitetura das máquinas virtuais para fazer o consenso bizantino. Para atingir isto, foi modificado o domínio que provê *drivers* de *hardware*, separando-o em duas máquinas virtuais, criando assim um domínio chamado de NV. A proposta é uma arquitetura com uma única

máquina suportando várias máquinas virtuais para oferecer o serviço replicado. As réplicas virtuais servem apenas para executar as requisições, sem participar da ordenação. A responsabilidade de efetuar a ordenação, ou o consenso, fica a cargo do domínio NV, que tem total acesso à abstração de rede e, por sua vez, a todas as requisições que vêm do cliente. Como o domínio NV é o único responsável pela votação, a execução não mais tem necessidade de utilizar $3f+1$ máquinas, fazendo o número decrescer para $2f+1$. Os autores consideram que, por se tratar também do *hypervisor* da máquina virtual, o domínio NV é inviolável. O problema com esta versão do modelo é a não tolerância a *crash* por parte do sistema anfitrião, já que era apenas uma máquina.

Os autores sugeriram então a abordagem Redundant Execution on Multiple Hosts (REMH) (REISER; KAPITZA, 2008), realizando a replicação do sistema anfitrião. Assim, REMH é capaz de tolerar até f faltas de *crash* em um total de $2f+1$ máquinas anfitriãs.

Fazendo um modelo muito semelhante ao que foi proposto por Castro e Liskov em (CASTRO; LISKOV, 2002), surgiu *Virtual Machine - Fault and Intrusion Tolerance* (VM-FIT), estudo composto pelas abordagens RESH e REMH. Neste novo modelo os autores sugerem uma recuperação proativa. Estas recuperações são disparadas periodicamente para que sistemas corrompidos sejam recuperados e iniciados a partir de uma base de código segura. Em VM-FIT a recuperação das réplicas fica a cargo do *hypervisor*. Como o VMM tem total controle sobre as máquinas virtuais, isso permite que um novo sistema seja iniciado antes que outro seja desligado. Contudo, o estado da nova réplica é diferente do estado das réplicas já iniciadas. Para solucionar este problema,é utilizado um esquema de *checkpoint* das réplicas. Desta maneira, para que uma réplica recém iniciada emparelhe com as demais, são usadas $f+1$ mensagens de CHECKPOINT iguais que possuem o estado a ser transferido. A transferência de estado pode ser feita através de um simples remapeamento de blocos de memória, caso a réplica recuperada apresente comportamento correto.

Por se tratar de um dos primeiros trabalhos a utilizar virtualização, o VM-FIT tem grande importância na área de tolerância a intrusão. Através de sua arquitetura, foi possível a proposta de um protocolo de consenso simples, baseado em um domínio confiável, além disso, o esquema para recuperação ficou simples e direto. Entretanto a criação do domínio NV traz alguma complexidade e dependência para seu código, já que lida com a modificação do *hypervisor*.

3.2.3.2 Remus: High Availability via Asynchronous Virtual Machine Replication

Em (CULLY et al., 2008) é apresentada uma abordagem que, apesar de não focar em dependabilidade, e sim em disponibilidade, traz uma solução interessante que, com poucas modificações, pode ser adaptada para BFT. Com base nisto, cabe uma breve discussão sobre sua arquitetura chamada Remus.

Remus utiliza uma arquitetura de servidores em par, um ativo e outro *backup*, ambos com o mesmo sistema. Ele alcança alta disponibilidade por meio da propagação frequente de *checkpoints* de uma máquina virtual ativa a um *host* físico de *backup*. No *backup*, a imagem da máquina virtual é armazenada em disco e pode iniciar sua execução imediata, se for detectada alguma falha no sistema ativo. Pelo fato do *backup* ser consistente apenas periodicamente com a réplica primária (ativa), toda saída de rede precisa ser guardada em *buffer* até que o estado da réplica de *backup* seja sincronizado. Quando uma imagem completa e consistente do *host* é recebida, o *buffer* é liberado para clientes, deixando o estado do sistema externamente visível. A réplica de *backup* fica inativa em seu *host* até que seja detectada uma falha na máquina virtual ativa. Enquanto inativa, a réplica de *backup* apenas serve de receptora para os *checkpoints*

O objetivo do Remus é assegurar a recuperação, completamente transparente, de falhas de paradas de um único *host* físico. Para tanto, o Remus provê as seguintes propriedades:

1. Tolerância a falhas de paradas em qualquer *host*;

2. Se ambos os *hosts*, primário e *backup*, falharem simultaneamente, os dados protegidos do sistema ficarão em estado consistente contra paradas;

3. Nenhuma saída ficará externamente visível até que o estado do sistema não tenha sido recebido pela réplica de backup.

Por se tratar de um trabalho com foco em disponibilidade e que preocupa-se apenas com falhas de paradas, este trabalho não está fortemente relacionado com nossa proposta. Todavia, a sua solução traz uma arquitetura que pode ser adaptada para serviços BFT, já que apresenta alta disponibilidade, que é muito importante para dependabilidade. Através do uso da solução por eles adotada e com a replicação das máquinas físicas, é possível criar um sistema BFT em que as réplicas, sempre que detectadas faltosas, pudessem se recuperar proativamente, diminuindo os danos do sistema em casos de faltas arbitrárias.

3.2.3.3 ZZ: Cheap practical BFT using virtualization

ZZ é proposto em (WOOD et al., 2008) e apresenta uma arquitetura com virtualização e replicação de máquina de estados que utiliza $3f+1$ réplicas no consenso e $f+1$ réplicas para execução das requisições em casos livres de faltas.

A configuração do ZZ permite que sejam colocadas até $N(f+1)$ máquinas virtuais em cada máquina física, considerando-se N como o número de aplicações em execução.

ZZ é baseado em duas compreensões, 1) se um sistema é construído para ser correto em ambientes assíncronos, ele precisa funcionar corretamente mesmo que algumas réplicas sejam arbitrariamente lentas. 2) Durante um período livre de faltas, um sistema construído para ser correto na presença de f faltas não deve ser afetado se f replicas suas estiverem desligadas. ZZ aproveita a segunda compreensão para desligar f réplicas em períodos livres de faltas, necessitando apenas $f+1$ réplicas para executar as requisições. Quando faltas ocorrem, ZZ aproveita-se da primeira compreensão e se comporta exatamente como se as f réplicas ociosas estivessem muito lentas, mas ainda assim, corretas.

A execução em caso normal do protocolo inicia com o envio de uma requisição de cliente com uma estampilha de tempo anexada e a identificação deste cliente. Esta requisição chega ao *cluster* de acordo que 1) define um número de ordenação para a requisição, 2) envia requisições efetivadas para o *cluster* de execução, 3) recebe relatórios de execução do *cluster* de execução, e 4) retransmite certificados para o cliente quando necessário. Ao receber a requisição de um cliente, cada réplica de acordo envia uma mensagem ORDER para todas as réplica de execução com a visão e o número de ordem. Uma réplica de execução, ao receber uma mensagem ORDER assinada por $2g+1$ réplicas de acordo, verifica se já executou todas as requisições com valor de ordem anterior ao desta mensagem, para então executar a nova requisição do cliente. A réplica de execução então envia uma mensagem contendo um relatório da execução da mensagem para as réplicas de acordo e uma mensagem contendo a resposta para o cliente. O cliente aceita, ao receber $f+1$ respostas iguais de diferentes réplicas.

Se o *cluster* de acordo detectar faltas no *cluster* de execução, isto é, se alguma réplica de acordo não receber os relatórios de execução ou se os relatórios não estiverem em acordo uns com os outros, ela envia uma mensagem de RECOVER para um subconjunto de *hypervisors* que controlam as f réplicas de execução ociosas. Quando um *hypervisor* recebe $g+1$ mensagens de RECOVER válidas e de réplicas diferentes, ele inicia a réplica ociosa. O *hypervisor* é considerado confiável neste modelo. Para recupe-

rar as réplicas ociosas, o ZZ confia numa abordagem de *checkpoint*, assim as réplicas que estavam antes ociosas podem agora ter seus estados igualados aos daquelas que estavam ativas. Entretanto o *checkpoint* pode estar desatualizado em relação às requisições pendentes, portanto, para reduzir o custo da recuperação da réplica ociosa, ZZ usa um esquema diferente. Uma réplica k sendo recuperada primeiramente obtém do *cluster* de acordo um *log* ordenado de requisições já comprometidas (*commited*) desde o *checkpoint* mais recente. Seja m o número de sequência mais recente no *checkpoint* e $n \geq m+1$ o número de ordenação mais recentemente comprometido. E sabendo que requisições no intervalo $[m+1, n]$ envolvem escritas no estado da aplicação, enquanto outras não. Então a réplica k começa a reexecutar as requisições neste intervalo, mas, somente, as que envolvem escritas no estado da aplicação. Desta maneira, ganha-se tempo na recuperação.

A abordagem apresentada em ZZ é vantajosa por manter parte do recurso, pelo menos do ponto de vista de processamento, ocioso. Com isto é possível se chegar a uma quantidade de réplicas de execução de $f+1$ para f faltosas. O esquema de recuperação de réplicas traz uma maneira econômica por fazer com que as réplicas só executem requisições que mudam o estado do sistema, diminuindo o tempo para trazer de volta o sistema a um estado correto. Entretanto, por utilizar mais de uma réplica por máquina física, esta abordagem transforma a máquina real em um gargalo e a quantidade de máquinas necessárias para criar o consenso ainda necessita de $3f+1$ réplicas, o que aumenta consideravelmente a quantidade de recursos computacionais necessários.

3.2.3.4 Diverse replication for single-machine byzantine-fault tolerance

Em (CHUN; MANIATIS; SHENKER, 2008) foi proposta uma abordagem que foca na utilização de replicação em um sistema com servidor único. Os autores sugerem a colocação de várias instâncias do serviço replicado na mesma máquina física para criar um sistema BFT. Este trabalho não apresenta protótipos, seu foco é teórico e traz algumas discussões interessantes sobre os desafios do BFT e como seria possível resolvê-los através do uso de virtualização. Os autores focam em dois desafios. A diversidade de réplicas, através do uso de diferentes tecnologias, sistemas operacionais, linguagens de programação, codificação, etc. E a independência de faltas.

Para resolver o problema da independência de faltas, é preciso garantir isolamento entre as réplicas, assegurando assim que uma não vai inferir em outras. Os autores sugerem que o uso de virtualização pode auxiliar no isolamento através do VMM (*Virtual Machine Monitor*), já que ele provê

isolamento de computação, memória, e disco. De maneira adicional, o *hypervisor* deveria fornecer um mecanismo de comunicação protegido entre as máquinas virtuais.

Os autores trazem uma discussão rica sobre o papel do VMM mostrando que ele escalona o processador de maneira justa entre as múltiplas máquinas virtuais. O VMM deve assegurar as propriedades de *liveness* e disponibilidade não deixando as máquinas virtuais sem ter acesso ao processador. Além disso, o VMM deve proteger a páginas da memória física, não permitindo que uma máquina virtual acesse o espaço de outra. É sua responsabilidade também isolar discos virtuais das convidadas garantindo que não vai haver sobreposição nem acessos indevidos. E por final, o *hypervisor* deve assegurar que a comunicação entre duas máquinas virtuais não seja violada por uma terceira máquina virtual. O VMM deve mediar as comunicações protegendo qualquer comunicação.

Assegurado o papel do *hypervisor*, o isolamento é possível.

Neste artigo os autores fazem um panorama sobre o estado em que a área de BFT se encontrava. Os autores trazem algumas discussões ricas sobre os principais desafios de da área de tolerância a faltas bizantinas focando em diversidade e independência de faltas. Os autores propõe o uso de tecnologias de virtualização para conseguir colocar em prática novas abordagens que consigam sanar estes problemas de maneira satisfatória.

3.2.4 Usando canal confiável

O problema do consenso (PEASE; SHOSTAK; LAMPORT, 1980), consiste em garantir que os processos corretos em um sistema distribuído entrarão em concordância em relação a um valor proposto por algum destes processos. O problema se resume em proposições de valor $v \in V$ e na decisão unânime dos processos corretos em função do v proposto. Fischer, Lynch e Paterson (FISCHER; LYNCH; PATERSON, 1985) provaram que em sistemas assíncronos é impossível se atingir consenso, pois qualquer processo pode sofrer faltas de *crash* inviabilizando o acordo. Algumas abordagens fazem uso de um canal confiável com premissas de sincronia para atingir o consenso de maneira protegida. É uma maneira de tornar o consenso seguro, permitindo ainda que a resiliência do sistema seja melhorada. Nossa abordagem também faz uso de um canal confiável, portanto, se encaixa nesta subárea.

3.2.4.1 How to tolerate half less one byzantine nodes in practical distributed systems

Em (CORREIA; NEVES; VERISSIMO, 2004) é apresentada uma solução através do uso do *wormhole* TTCB (*Trusted Timely Computing Base*). O TTCB é um componente inviolável distribuído utilizado na implementação de um serviço BFT de replicação de máquina de estados que reduz o número de réplicas necessárias para $2f+1$ com até f faltosas.

O modelo do sistema é composto por um conjunto de servidores conectados via rede. O TTCB tem seu próprio canal de comunicação e é distribuído tendo partes locais em alguns servidores. O conjunto de serviços que é provido pelo TTCB é limitado e muito simples. O primeiro serviço provido é o serviço de autenticação local que garante a integridade da comunicação entre o servidor e seu TTCB local. O segundo serviço é TMO *The Multicast Ordering* que é implementado dentro do TTCB e, por ser um serviço de ordenação, é utilizado para implementar um protocolo de difusão atômica (vide 2.2.1) que é a base para o esquema de replicação.

A fim de contornar a impossibilidade de resultado Fischer, Lynch e Paterson (FLP)[1] a versão original da TTCB foi síncrona, mas se outra solução for utilizada com esta finalidade, por exemplo, detectores de falhas, o TTCB pode ser implementado como um componente confiável sem premissa de tempo real. Para aumentar a proteção do TTCB, ele precisa ser implementado em um módulo de *hardware* e seus canais de comunicação precisam ser completamente separados um do outro. Isso obriga que cada servidor possua duas placas de rede. Na primeira é feita a comunicação entre os clientes e os servidores (rede de *payload*). E na segunda a comunicação onde será efetuado o processo de consenso (rede controlada segura).

O algoritmo de replicação de máquina de estados que utiliza o TTCB opera basicamente da seguinte maneira: 1) um cliente envia um comando para um dos servidores; 2) o servidor envia o comando para todos os demais servidores usando o protocolo de difusão total; 3) cada servidor executa o comando e envia uma resposta para o cliente; 4) o cliente aguarda por $f+1$ respostas iguais entre si de diferentes servidores.

O cerne do algoritmo executado por cada servidor é o protocolo de difusão atômica que garante que se uma réplica que enviou as ordenações de mensagem para os demais servidores for correta, então todas as réplicas corretas irão executar as mensagens na mesma ordem. O serviço TMO é quem dá embasamento para o protocolo de difusão atômica, designando um

[1] FLP diz que nenhum protocolo determinístico é capaz de resolver o problema de consenso em um sistema síncrono se um simples processo pode parar de funcionar.

número de ordenação para as mensagens. Quando um processo *p* quer enviar uma mensagem para os outros processos, ele entrega ao TTCB um *hash* da mensagem e difunde a mensagem através da rede. Quando outro processo *q* recebe a mensagem, ele também entrega ao TTCB um *hash* da mensagem; Quando um certo número de processos tiver efetuado esta operação, o TTCB designará um número de ordenação para a mensagem e enviará este número aos processos. Todo processo correto vai processar as mensagens de acordo com a ordem definida pelo TTCB.

Em (CORREIA; VERISSIMO; NEVES, 2002) é explicado mais a fundo como foi implementado o TTCB. Na conclusão abaixo é feita uma discussão mais profunda sobre o assunto.

O trabalho apresentado é de grande importância, pois foi um dos primeiros a apresentar uma solução prática para a melhoria da resiliência em sistemas BFT. Além disso, é um dos pioneiros a colocar em prática a utilização de *wormhole* para criar uma arquitetura híbrida. Porém a implementação do TTCB necessita da utilização de um *hardware* à parte, como pode ser visto em (CORREIA; VERISSIMO; NEVES, 2002). Este componente dificulta sua aplicação prática, além disso, é preciso blindar este *hardware* com um componente de *software* responsável por separar as operações do TTCB das operações do sistema operacional. Os autores fazem essa blindagem através da criação de módulos para o *kernel* do RT-Linux. Essa dependência com o *kernel* do RT-Linux implica diretamente em:

1. Dependência de versão do *kernel*, o que pode fazer com que esteja defasado em relação às versões mais recentes, principalmente sob o aspecto das medidas de segurança. No uso do RegPaxos isto não é um problema, pois o sistema operacional da máquina hospedeira pode ser qualquer um, respeitando a premissa de heterogeneidade (GARCIA et al., 2011; OBELHEIRO et al., 2006).

2. As bibliotecas de acesso ao TTCB precisam ser implementadas para cada linguagem de programação. Isso limita o uso da técnica de diversidade, e com o surgimento de linguagens e paradigmas o TTCB tem que ser adaptado. O Registrador Distribuído Compartilhado do RegPaxos tem seu acesso independente da linguagem de programação, permitindo que o serviço seja implementado através de diversos paradigmas e linguagens.

3. As limitações do TTCB impostas a um atacante são em termos de remoção de privilégios do super usuário do sistema operacional. Portanto, o atacante tem acesso ao sistema hospedeiro e, como o TTCB protege apenas o ID/GID 0 (*root*), o acesso local é suficiente para

técnicas de escalamento de privilégio ou *exploits* locais, que são aqueles que são executados diretamente na máquina e que, normalmente, exploram falhas do tipo *buffer/stack/heap overflow* ou erros de configuração.

Além do mais, a separação entre as redes segura e de *payload* é feita com a separação de placas de rede, o que implica na necessidade de pelo menos duas placas de rede por servidor TTCB. No RegPaxos a separação das redes é feita através do isolamento da máquina virtual diminuindo a quantidade de recursos.

3.3 CONSIDERAÇÕES FINAIS

Neste capítulo foram abordadas os principais trabalhos que serviram de base e inspiração para o trabalho aqui proposto. Foi apresentado de maneira mais profunda o conceito de modelo híbrido. Além disso os trabalhos mostram o estado da arte da área e os principais desafios e próximos passos de evolução.

A utilização de uma abordagem híbrida se mostra bastante promissora, entretanto ainda requer que sejam feitos alguns ajustes no tocante a sua simplicidade. Atualmente os trabalhos que seguem esta linha necessitam de grandes mudanças nos recursos computacionais para que se possa garantir seu funcionamento correto. Este é talvez um dos desafios mais interessantes, pois superá-lo permite que os sistemas híbridos sejam melhor utilizados devido a facilidade de reprodução a partir de recursos computacionais já disponíveis.

4 REPLICAÇÃO DE MÁQUINAS DE ESTADO UTILIZANDO REGISTRADORES COMPARTILHADOS DISTRIBUÍDOS

Serviços e sistemas tolerantes a faltas têm sido muito pesquisados nas últimas décadas. Em sua maioria (REITER, 1995; CASTRO; LISKOV, 2002; YIN et al., 2003; CORREIA; NEVES; VERISSIMO, 2004; AMIR et al., 2006; VERONESE et al., 2011), os trabalhos focam em maneiras de mascarar a ocorrência de falhas e intrusões, através do uso de técnicas de replicação de máquinas de estado. Entretanto, muitas dentre estas abordagens são inviáveis na prática ou são direcionadas para situações muito específicas.

Em (CORREIA et al., 2002) foi apresentada uma abordagem de modelo híbrido de falhas que descreve um sistema distribuído que possui um componente inviolável local aos servidores. Este componente está sujeito apenas a falhas de *crash* e está interconectado aos componentes locais de outros membros através de uma rede controlada. Através deste componente são realizadas operações para o estabelecimento de acordo e consenso. O sistema é dividido entre aqueles componentes que estão sujeitos as faltas bizantinas e aqueles que não estão. Muitos outros trabalhos se seguiram com base neste (VERONESE et al., 2011; CHUN et al., 2007; LEVIN et al., 2009; CORREIA; NEVES; VERISSIMO, 2004). Entretanto o uso de um componente ou subsistema até então requeria que houvesse mudanças no sistema operacional dos servidores e/ou em seus *hardwares*.

Com o intuito de obter uma abordagem para serviços que sobrevivam a intrusões a um custo reduzido, tanto na quantidade de recursos computacionais necessários para tolerar faltas (resiliência), quanto na quantidade de passos de comunicação necessários para oferecer os serviços, é proposta uma abordagem de modelo híbrido que retira a complexidade de modificações em sistemas operacionais e *hardware* e substitui pelo uso de tecnologias acessíveis como virtualização e abstrações de compartilhamento de memória.

Os pontos chave da abordagem proposta são o uso de um componente confiável (DSR- *Distributed Shared Register*) e o uso de virtualização para limitar o acesso a este componente. O DSR é uma abstração de memória compartilhada que provê um mecanismo simples e menos custoso de comunicação entre as réplicas de execução da abordagem.

O foco do trabalho é utilizar virtualização para restringir o acesso à abstração de memória compartilhada, permitindo que as partes cruciais dos algoritmos, isto é, o consenso em relação às transições de estado, sejam executadas dentro do componente confiável. Com isto, a arquiteturas podem ser implementadas com um conjunto de $2f+1$ máquinas físicas para se tolerar f faltas. A execução deste protocolo em um ambiente virtualizado é de suma

importância, já que a virtualização é capaz de oferecer o isolamento total entre máquinas hospedeiras e convidadas.

Será detalhado na Seção 4.1 o modelo do sistema com as suposições necessárias ao funcionamento dos protocolos. A partir daí, será apresentada uma definição detalhada da tecnologia de virtualização (4.1.1) e da abstração memória compartilhada necessárias para a criação dos registradores compartilhados distribuídos (4.1.2). Na Seção 4.2 é apresentado o RegPaxos, a primeiro algoritmo completo de serviço tolerante a intrusão baseado no DSR, bem como as provas de corretude do mesmo. Na seção seguinte é apresentado o algoritmos de difusão atômica DIFATO (seção 4.3), além de suas provas de corretude.

4.1 MODELO DO SISTEMA E ARQUITETURA

O modelo de sistema adotado é híbrido (VERÍSSIMO, 2006), o que significa que existe variação, de componente para componente, em relação às suposições de sincronismo e presença/severidade de faltas e falhas (CORREIA; VERISSIMO; NEVES, 2002; VERÍSSIMO, 2006). Em nosso modelo, consideramos diferentes suposições para os subsistemas que executam no *host* e no *guest* das máquinas virtuais que compõem o sistema. Neste modelo, o conjunto $C = \{c_1, c_2, c_3, ...\}$ representa o número finito de processos clientes e $S = \{s_1, s_2, s_3, ..., s_n\}$ representa o conjunto de servidores contendo n elementos que implementam o serviço oferecido pelo sistema de replicação de máquinas de estado (SMR - *State Machine Replication*). Cada servidor possui uma máquina virtual que contém apenas um sistema como *guest*. O modelo de falhas admite que até $f \leq [\frac{n-1}{2}]$ servidores incorram em faltas de suas especificações, apresentando comportamento arbitrário ou bizantino (LAMPORT; SHOSTAK; PEASE, 1982): um processo faltoso pode desviar de suas especificações omitindo ou parando de enviar mensagens, ou ainda apresentar qualquer tipo de comportamento (malicioso ou não) não especificado. Entretanto, assumimos independência entre as faltas, em outras palavras, a probabilidade de um servidor incorrer em faltas é independente da ocorrência de faltas em outro servidor. Na prática, é possível atingir-se isto através do uso extensivo de diversidade (diferentes *hardwares/softwares*, sistemas operacionais, máquinas virtuais, bases de dados, linguagens de programação, etc) (RODRIGUES; CASTRO; LISKOV, 2001).

Nosso modelo de sistema prevê o uso de duas redes de comunicação. A primeira, a rede de *payload* é assíncrona e é utilizada para transferência de dados da aplicação. Não existem suposições baseadas em tempo para a rede de *payload* e sua utilização se dá entre clientes e servidores para envio de

requisições e respostas. A segunda rede é controlada (registradores compartilhados distribuídos) e é utilizada para que os servidores troquem mensagens do protocolo de acordo. Assumem-se as seguintes propriedades para esta rede:

- é segura, isto é, resistente a qualquer possível ataque, falhando apenas por *crash*;
- é capaz de executar operações com delimitação temporal;
- provê apenas duas operações, leitura e escrita em registradores. Estas operações não podem ser afetadas por faltas maliciosas.

Assumimos que cada par cliente-servidor c_i, s_j e cada par de servidores s_i, s_j está conectado por um canal confiável com duas propriedades: se o remetente e o destinatário de uma mensagem são ambos corretos, então (1) a mensagem é , em algum momento, recebida e (2) a mensagem não é modificada no canal (CORREIA; NEVES; VERISSIMO, 2004). Na prática, estas propriedades podem ser obtidas com o uso de criptografia e retransmissão (WANGHAM et al., 2001). *Message authentication codes* (MACs) são *checksums* criptográficos que servem ao nosso propósito, e usam apenas criptografia simétrica (MENEZES; OORSCHOT; VANSTONE, 1996; CASTRO; LISKOV, 2002). Os processos precisam compartilhar chaves simétricas para fazerem uso de MACs. Assume-se que estas chaves são distribuídas antes do protocolo ser executado. Na prática, isto pode ser resolvido usando protocolos de distribuição de chaves disponíveis na literatura (MENEZES; OORSCHOT; VANSTONE, 1996).

A arquitetura do sistema está representada na figura 5. As máquinas virtuais são os servidores que possuem a réplica do serviço. As máquinas físicas compõe uma abstração de memória compartilhada emulada (GUERRAOUI; RODRIGUES, 2006a) (veja 4.1.2) aqui chamada de registradores compartilhados distribuídos (DSR - acrônimo para *Distributed Shared Registers*). Cada máquina física possui seu próprio espaço dentro dos DSR, onde sua respectiva máquina virtual registra mensagens do tipo PROPOSE, ACCEPT ou CHANGE. Todos os servidores podem ler todos os registros, não importando quem o registrou.

Assume-se que apenas as máquina físicas podem, de fato, conectar-se à rede utilizada pelos DSR. Isto implica que os registradores compartilhados distribuídos só estão acessíveis pelas máquinas físicas que hospedam (*host*) as máquinas virtuais. Transitivamente, os DSR não estão acessíveis através do *guest* das máquinas virtuais. Cada processo é encapsulado dentro de sua própria máquina virtual, assegurando isolamento. Todas as comunicações entre clientes e servidores acontecem em uma rede separada e, do ponto de vista

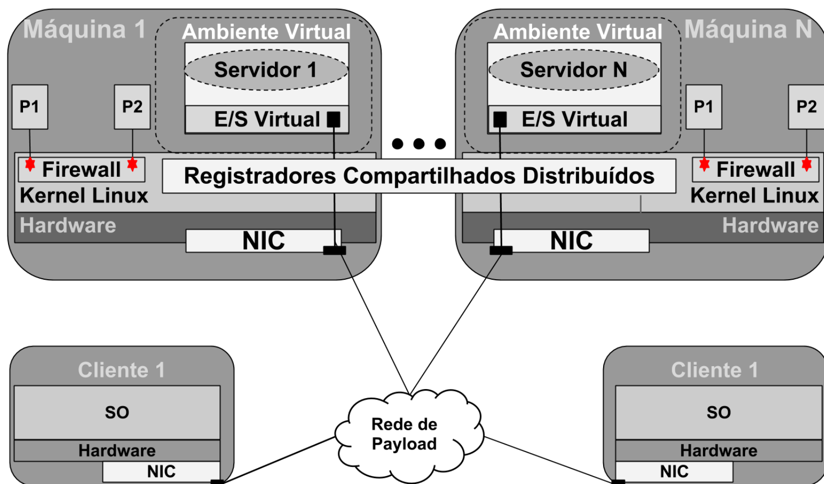


Figura 5 – Visão geral da arquitetura

do clientes, a máquina virtual é transparente. Isto significa que os clientes não reconhecem a arquitetura física-virtual. Cada máquina possui apenas uma interface de rede, regras de *firewall* e/ou o modo *bridge* são utilizados no *host* para assegurar a divisão entre as redes. Assumimos que as vulnerabilidades do *host* não podem ser exploradas através da máquina virtual. O gerenciador de máquinas virtuais (*hypervisor*) assegura este isolamento, garantindo que um atacante não tem meios para acessar o *host* através da máquina virtual. Isto é uma premissa em tecnologias de virtualização, como *VirtualBox, LVM, XEN, VMWare, VirtualPC*, etc. Nosso modelo assume que o sistema *host* é inacessível externamente, o que também pode ser garantido através do uso do modo *bridge* e/ou *firewalls* no sistema hospedeiro.

4.1.1 Tecnologia de virtualização

Dois dos maiores desafios deste trabalho, que são a disponibilização de um serviço confiável de registradores e a garantia do seu isolamento, são resolvidos através do uso de virtualização. Este serviço é implementado nas máquinas hospedeiras que permitem acesso controlado das réplicas. O uso de virtualização traz ainda outros benefícios, como permitir controle e monitoramento da comunicação entre as réplicas e a regeneração de réplicas faltosas, através do uso de imagens do sistema, o que pode ser implementado em tra-

balhos futuros.

A partir da tecnologia de virtualização é possível criar o serviço confiável de registradores mantendo-o isolado. Diversos gerenciadores de máquinas virtuais (VMM - *Virtual Machine Monitor ou hypervisor*) possuem mecanismos que podem ser utilizados para garantir o isolamento. Qualquer VMM tipo 2 pode ser utilizado e, inclusive, ressaltamos que o uso de diversidade na utilização de *hypervisors* é muito interessante, pois dificulta a exploração de vulnerabilidades por parte de um atacante. A segurança do *hypervisor* é essencial para obter isolamento, por isso alguns trabalhos estudaram como melhorá-la (MURRAY; MILOS; HAND, 2008; WANG; JIANG, 2010; SZEFER et al., 2011). Assume-se que um atacante não tem informações suficientes para determinar a arquitetura física-virtual, portanto, outras técnicas podem ser empregadas para garantir o isolamento e a segurança da máquina virtual. Técnicas como Blue Pill (RUTKOWSKA, 2006) impedem/dificultam a detecção de máquina virtual de maneira mais contundente.

4.1.2 Registradores compartilhados distribuídos

Memória compartilhada emulada é uma abstração de registradores de um conjunto de processos que se comunicam através de troca de mensagens (GUERRAOUI; RODRIGUES, 2006a). Esta definição é realmente atraente, já que permite que a memória compartilhada seja construída utilizando-se qualquer tecnologia para compartilhamento de memória. Um memória compartilhada, emulada ou não, pode ser vista como um *array* de registradores compartilhados. Consideramos aqui a definição sob a ótica do programador. O tipo de registrador compartilhado especifica que operações podem ser efetuadas e os valores retornados por elas (GUERRAOUI; RODRIGUES, 2006a). Os tipos mais comuns são registradores de leitura/escrita. As operações de um registrador são invocadas pelos processos do sistema para troca de informações.

Na arquitetura proposta, empregou-se os registradores compartilhados distribuídos (DSR - acrônimo para *Distributed Shared Register*), uma memória compartilhada emulada baseada em troca de mensagens, desenvolvida sobre uma rede controlada e através do uso de arquivos locais. Assume-se que a rede controlada só é acessada por componentes dos DSR. Os DSR são implementados nos *hosts* das máquinas virtuais dos sistemas e assume-se que o VMM assegura o isolamento entre *guests* e *hosts*. Os DSR são utilizados para implementar um serviço de consenso que garante que todos os servidores corretos vão devolver as mesmas mensagens na mesma ordem e, se o remetente é correto, todos os servidores vão devolver a mensagem envi-

ada. Os DSR são capazes de efetuar apenas duas operações (1) $read()$, que lê a última mensagem escrita nos DSR e (2) $write(m)$, que escreve a mensagem m nos DSR. Assume-se duas propriedades com relação às operações acima:

(i) *liveness* - a operação , em algum momento, termina;

(ii) *safety* - a operação de leitura sempre retorna o último valor escrito.

Para garantir isto, cada servidor possui um arquivo em que seu *guest* tem acesso para escrita e um segundo arquivo onde o mesmo possui acesso apenas para leitura. Todos os acessos são feitos por um único processo (GUERRAOUI; RODRIGUES, 2006a). O primeiro arquivo é espaço da réplica e nenhuma outra réplica pode escrever nele. O segundo arquivo representa o espaço de registradores das demais réplicas e é atualizado pelos DSR. O *hypervisor* fornece suporte para fazer com que um arquivo criado no *host* esteja acessível no *guest*, forçando as permissões de escrita e leitura, além disso, pode-se utilizar NFS (*Network FileSystem*) para trabalhar este aspecto.

Os DSR aceitam apenas mensagens tipadas, e existem apenas três tipos: (i) PROPOSE, (ii) ACCEPT e (iii) CHANGE. As mensagens sem tipo ou com tipos não permitidos são ignoradas. Todas as mensagens trocadas nos DSR são automaticamente identificadas, isto significa que, quando um servidor escreve uma mensagem, os DSR colocam o ID da réplica na mensagem, de modo que uma réplica não possa se passar por outra. Assume-se que a comunicação é feita através de *fair links* com as seguintes propriedades: se o remetente e o destinatário de uma mensagem são ambos corretos, então (1) se a mensagem é enviada infinitamente para um destinatário correto, então ela é recebida infinitas vezes; (2) existe um tempo T, tal que, se a mensagem é retransmitida infinitas vezes para um destinatário correto de acordo com um tempo t_0, então o destinatário recebe a mensagem pelo menos uma vez antes de $t_0 + T$; e (3) as mensagens não são modificadas no canal (YIN et al., 2003). Esta suposição parece razoável na prática, uma vez que os DSR são síncronos e podem ser afetados apenas por faltas de *crash*, baseado no isolamento proporcionado pelo VMM.

4.2 O PROTOCOLO REGPAXOS

Nesta seção é apresentado um algoritmo que utiliza-se do modelo de sistema anteriormente discutido para implementar uma replicação de máquinas de estado baseada no Paxos Bizantino (CASTRO; LISKOV, 2002). Neste algoritmo umas das réplicas atua como líder e tem responsabilidade de propor a ordem de execução para as requisições de clientes dentro do ambiente

de Registradores Compartilhados Distribuídos. Os demais servidores servem apenas de réplicas do serviço. A escolha do primeiro líder é baseada em configuração e a mudança de liderança ocorre sempre que a maioria das réplicas ($f+1$) concordarem ser necessário. O protocolo se inicia quando um cliente requisita a execução de alguma tarefa nos servidores. O líder é responsável por ordenar e difundir mensagens dos clientes. A ordenação acontece quando o líder escreve nos DSR uma mensagem do tipo PROPOSE. Mensagens deste tipo incluem a mensagem original do cliente seguida de um número de ordenação para a mesma. Cada servidor delibera individualmente se deve ou não aceitar a proposta. Aceitar a proposta significa escrever nos DSR uma mensagem do tipo ACCEPT, esperar por $f-1$ mensagens de aceitação (pois já possui a sua própria e a mensagem de PROPOSE, isto é $f-1+2 \implies f+1$), executar a tarefa na ordem estipulada e responder para o cliente com o resultado. O fluxo da operação em caso normal pode ser visto na figura 6.

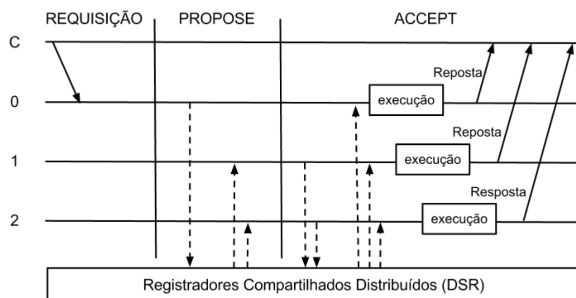


Figura 6 – Fluxo da requisição à resposta

Como em outros sistemas tolerantes a faltas bizantinas (KOTLA et al., 2008; CORREIA; NEVES; VERISSIMO, 2004; YIN et al., 2003; VERONESE et al., 2011; CASTRO; LISKOV, 2002), para lidar com o problema de um servidor malicioso s_j, que poderia deixar de difundir mensagens, o cliente espera por pelo menos $f+1$ respostas de diferentes servidores por um tempo $T_{reenviar}$. Após $T_{reenviar}$ o cliente reenvia a requisição para todos os servidores. Uma réplica honesta (não líder) ao receber uma requisição correta, solicita uma mudança de liderança. Se $f+1$ servidores corretos solicitarem a mudança de líder, então a mudança ocorre e o protocolo progride. Entretanto, o sistema de *payload* é assumidamente assíncrono, portanto, não existem limitantes temporais para sua comunicação, não sendo possível definir um valor $T_{reenviar}$ ”ideal”. Em (CORREIA; NEVES; VERISSIMO,

2004) é mostrado que o valor $T_{reenviar}$ envolve uma troca: se for alto demais, o cliente pode esperar tempo demais para ter sua operação executada; se for baixo demais, o cliente pode fazer o reenvio do comando sem necessidade. O valor deve ser selecionado considerando-se essa troca. Se o comando é reenviado sem necessidade, as duplicatas são descartadas pelo sistema.

4.2.1 Propriedades

O algoritmo de Replicação de Máquinas de Estado consegue assegurar as seguintes propriedades:

- **Ordem Total (safety):** Toda requisição é executada sequencialmente e de acordo com a mesma ordem em todas as réplicas;

- **Terminação (liveness):** Mesmo na ocorrência de falhas, todas as requisições de clientes são terminadas em algum momento.

Para assegurar isto, o número de réplicas faltosas deve ser minoria em relação às corretas, isto é $f \leq \frac{n-1}{2}$. Para que a ordem total seja atingida, todas as réplicas executam as requisições na ordem definida pelo líder e que tenham sido aceitas por pelo menos $f+1$ réplicas componentes do sistema, ou seja, a ordem de execução é dada pela maioria em consenso.

O protocolo garante a propriedade *safety*, através do consenso, porém, como foi discutido, uma certa sincronia é necessária para garantir *liveness*, já que o cliente não pode ficar esperando indefinidamente para ter sua requisição atendida.

4.2.2 Execução Normal do Protocolo

1. O protocolo inicia-se quando um cliente c envia ao servidor líder a mensagem $\langle REQUEST, o, t, c_a, v\rangle_{\sigma ci}$ com a operação o que deseja que seja executada e o seu endereço c_a. O campo t é o *timestamp* da requisição para assegurar a semântica de apenas uma execução, em outras palavras, os servidores não executam uma operação do cliente c cujo *timestamp* não seja maior que último *timestamp* para o mesmo cliente. Esta política impede que uma mesma tarefa seja executada inúmeras vezes. O campo v é o vetor que armazena os MACs gerados pelo cliente para cada servidor. Os MACs são gerados utilizando-se a mensagem do cliente e as chaves que foram compartilhadas previamente entre o cliente e os servidores. Em função disto, cada servi-

dor pode averiguar a integridade da mensagem, descartando-a se necessário.

2. Após receber a requisição do cliente e verificar sua integridade usando o MAC em v, o líder gera uma proposta e a escreve nos DSR. A mensagem $\langle PROPOSE, n_m, m\rangle_{\sigma s}$ gerada pelo líder possui em seu corpo m, que é a mensagem original do cliente, e n_m, que representa o número de ordenação para a mensagem. Vale ressaltar que, como foi discutido, os DSR automaticamente adicionam nas mensagens o identificador do servidor que as registrou. Após escrever a mensagem, o líder aguarda pelas mensagens de aceite de acordo com a proposta. Em paralelo, outras propostas podem ser ordenadas, a medida que chegam requisições. Após receber f mensagens concordando com a proposta (pois a proposta mais f aceites constituem as $f+1$ mensagens mínimas para o acordo) o líder guarda a mensagem e a executa na ordem estipulada. O resultado da execução é enviado para o cliente em uma mensagem $\langle REPLY, t, c_a, r_a, r\rangle_{\sigma si}$ que contém o endereço do cliente c_a, o *timestamp* t, o endereço da réplica r_a, e o resultado r. Este comportamento pode ser visto no algoritmo 1. Como foi discutido em 4.1, todas as mensagens trocadas nos DSR são , em algum momento, entregues se nem o destinatário, nem o remetente, tiverem sofrido falta de *crash*.

Algoritmo 1: Nodo líder

Constants:
f : int // Maximum tolerated faults
T : int // Maximum waiting time for a proposal to be decided
Variables:
accepted : int // counter of acceptance for some ordering
1 **upon** receive $\langle REQUEST, o, t_j, c_a, v\rangle_{\sigma ci}$ from client
2 **if** $t_j \leq t_{j-1}$ *for* c_i *OR isWrong(v)* **then**
3 | return;
4 **end**
5 write($\langle PROPOSE, n, \langle REQUEST, o, t_j, c_a, v\rangle_{\sigma ci}\rangle_{\sigma si}$) into DSR;
6 accepted = waitForAcceptance(T);
7 **if** $accepted \geq f+1$ **then**
8 | store $\langle PROPOSE, n, \langle REQUEST, o, t_j, c_a, v\rangle_{\sigma ci}\rangle_{\sigma si}$ in the execution buffer;
9 **end**
10 return;

3. Sempre que a réplica r_j recebe uma proposta do líder, r_j a valida, o que significa que (i) é feita a verificação da integridade da mensagem do cliente usando o MAC em v, além da verificação de seu *timestamp* e (ii) é feita a verificação de que nenhuma outra mensagem tenha sido ordenada com o mesmo n_m. Após serem feitas as verificações, se r_j aceitar a proposta, então é criada uma mensagem $\langle ACCEPT, n_m, h_m\rangle_{\sigma sj}$ que é

> escrita nos registradores. Os campos da mensagem representam o *hash* da mensagem do cliente h_m e o número de ordenação proposto pelo líder n_m. Após a escrita nos registradores, a réplica aguarda por $f-1$ mensagens de aceitação (já que já possui dois aceites, a sua mensagem e a proposta do líder). Em paralelo, outras mensagens podem ser aceitas, à medida que propostas cheguem aos DSR. A réplica r_j executa as requisições na ordem em que foram aceitas e responde aos clientes com mensagens de REPLY. Este comportamento pode ser observado em 2.

Após receber $f+1$ respostas iguas entre si e de diferentes servidores, o cliente finalmente as aceita como corretas.

Algoritmo 2: Nodo não líder

Constants:
f : int // Maximum tolerated faults
T : int // Maximum waiting time for a proposal to be decided.
Variables:
accepted : int // counter of acceptance for some ordering

1 **upon** read $\langle PROPOSE, n, \langle REQUEST, o, t, c_a, v\rangle_{\sigma ci}\rangle_{\sigma ss}$ from DSR
2 **if** *isValid(v) and isValid(n) and* $t_j > t_{j-1}$ *for* c_i **then**
3 | write($\langle ACCEPT, n, h_m\rangle_{\sigma si}$) into DSR;
4 | accepted = waitForAcceptance(T);
5 | **if** *accepted* $\geq f+1$ **then**
6 | | store $\langle PROPOSE, n, \langle REQUEST, o, t_j, c_a, v\rangle_{\sigma ci}\rangle_{\sigma si}$ in the execution buffer;
7 | **end**
8 **else**
9 | write($\langle CHANGE, h_m, s_s\rangle_{\sigma si}$) into DSR and bufferize;
10 **end**
11 return;

4.2.3 Execução do Protocolo na Ocorrência de Faltas

A operação sob a presença de faltas implica na mudança de líder, portanto, fazemos uma breve explicação de como essa mudança se desenvolve.

Mudança de líder: durante a configuração do sistema, todas as réplicas recebem um número de identificação. Estes números são sequenciais e iniciados em zero. Todas as réplicas conhecem o identificador do líder L_{id} e o número total de réplicas no sistema N. Quando $f+1$ réplicas corretas suspeitam do líder atual, elas simplesmente definem $L_{id} = L_{id} + 1$ como o próximo líder se $L_{id} < N$, senão, $L_{id} = 0$

Como foi discutido, a réplica r_j valida qualquer proposta que receba. Se por alguma razão a réplica decidir não aceitar a proposta ou se r_j receber uma mensagem correta de um cliente e verificar que a mesma não foi proposta, r_j solicitará uma mudança de líder. O diagrama de fluxos da figura 9

exemplifica um dos casos de execução sob a presença de faltas.

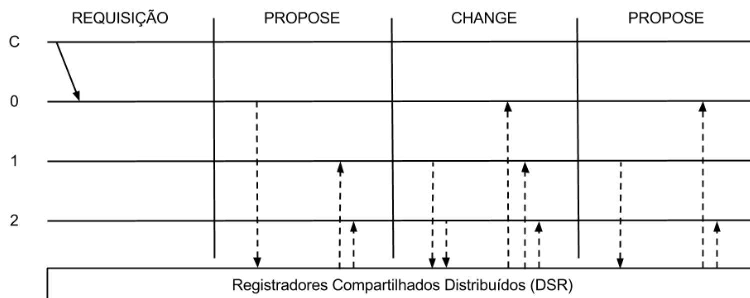


Figura 7 – Fluxo de uma mudança de líder

1. O protocolo pode iniciar o modo faltoso de acordo com as duas maneiras abaixo:

   (a) Quando a réplica r_j recebe uma mensagem do tipo CHANGE, mas ainda não suspeita do líder, ela armazena a mensagem em seu *buffer* para utilizá-la futuramente, se necessário.

   (b) Quando a réplica r_j suspeita do líder por alguma mensagem m, então r_j escreve nos DSR e em seu *buffer* uma mensagem do tipo $\langle CHANGE, h_m, s_s \rangle_{\sigma s j}$ que contém o *hash* da mensagem h_m e o id s_s do líder do qual r_j suspeita.

2. A réplica r_j inicia uma busca em seu *buffer* de suspeita para encontrar $f+1$ mensagens relacionadas à mensagem m. Caso r_j encontre $f+1$ (incluindo ao seu próprio) identificadores para a mesma mensagem, então a réplica efetua a mudança de líder, de acordo com o algoritmo 3.

3. Se o identificador do líder for o da réplica r_j, então a réplica recomeça o processo de ordenação utilizando-se das mensagens de aceite escritas nos DSR para definir o estado atual do sistema. Recomeçando o processo de acordo com a última mensagem que foi aceita pela maioria. Se o identificador não for o da réplica r_j, então a réplica aguardará pela mensagem do novo líder.

4. Quando r_j receber uma proposta do novo líder, então ela limpará seu *buffer* e voltará o protocolo para a operação sem a presença de faltas.

Algoritmo 3: Mudança de líder

```
    Constants:
    f : int // Maximum tolerated faults
    n : int // The number of replicas in the system
    Variables:
    leader : int // The actual leader ID
    changeCounter : int // has the count of the different servers that suspect from the leader
 1  upon suspect s_s or read ⟨CHANGE, h_m, s_s⟩_σs from DSR
 2  bufferize(⟨CHANGE, h_m, s_s⟩_σs);
 3  changeCounter = searchFor(buffer, h_m, s_s);
 4  if changeCounter ≥ f + 1 then
 5  |   leader++;
 6  |   if leader == n then
 7  |   |   leader = 0;
 8  |   end
 9  end
10  return;
```

4.2.4 Provas de corretude

Na seção 4.2.1 foi definido que o algoritmo deve garantir que todas as réplicas executem as mesmas requisições na mesma ordem - Ordem Total (*safety*) - e que todas as requisições iniciadas pelo cliente devem ser executadas em algum momento pelo serviço - Terminação (*liveness*).

Os seguintes axiomas discutidos anteriormente são necessários para auxiliar nas provas:

Axioma 1. *Apenas uma minoria de réplicas pode falhar em suas especificações. Para cada f réplicas faltosas, existem $f+1$ corretas.*

Axioma 2. *Toda réplica que se desvia do comportamento determinado é considerada faltosa.*

Axioma 3. *A máquina hospedeira é inacessível através máquina hospedada.*

Axioma 4. *Todos os processos tem acesso irrestrito ao espaço de registradores, e, portanto, leem os mesmos registros.*

Axioma 5. *Nenhum registro pode ser alterado dentro do DSR.*

Axioma 6. *Não existe concorrência - dentro de cada hospedeiro - no acesso ao espaço de registradores. Apenas um processo efetua as operações.*

Axioma 7. *Existe um tempo máximo Δ_t para entrega das mensagens no DSR.*

Axioma 8. *Sendo um modelo parcialmente síncrono, existe um tempo máximo - que é variável e desconhecido - para entrega das mensagens na rede.*

Axioma 9. *O cliente não espera indefinidamente pela resposta de sua requisição. O cliente reenvia sua requisição periodicamente até que receba uma resposta válida.*

Axioma 10. *Não é possível a falsificação de mensagens por uma réplica maliciosa.*

Axioma 11. *O serviço executado pelas réplicas é determinístico e todas as réplicas iniciam o protocolo no mesmo estado.*

Após declaradas estas premissas, podemos dar continuidade às provas de corretude.

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 7, 8 e 9, e se a a réplica líder for correta, então todas as réplicas receberão a requisição do cliente em algum momento. Caso a réplica líder não seja correta, então são efetuadas trocas de líder até que uma correta receba a liderança.

Segundo o algoritmo 1 (linha 5), a réplica líder enviará uma requisição a todas as réplicas do sistema, contendo, não somente a mensagem do cliente, mas também a ordem proposta por este líder. Se a ordem for considerada em consenso correta, todas as réplicas vão guardá-la em seus *buffers* de execução para executá-la e entregá-la na ordem estipulada, como pode ser visto nos algoritmos 1 (linhas 7 - 9) e 2 (linhas 5-7). Se a proposta não for considerada correta, então haverá a troca de líder até que se atinja o consenso em relação a alguma ordenação, como pode ser visto nos algoritmos 2 (linhas 8 - 10) e 3. □

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

Demonstração. Prova (esboço). Devido aos axiomas 4 e 5, todas as réplicas obtém as mesmas informações do DSR. Portanto, todas conhecem a ordem definida sob consenso da maioria para a qualquer requisição. Se duas réplicas corretas obtiverem dados diferentes do espaço de registradores, então esta condição entra em contradição com os axiomas supracitados. Além disso, nas réplicas corretas, as requisições são colocadas na fila de execução de acordo com a ordem definida pela maioria em consenso, como pode ser visto nos algoritmos 1 (linhas 7 - 9) e 2 (linhas 5-7). □

Lema 3. *Se um processo correto p_i executa uma requisição r e obtém o valor x como resultado desta execução, então pelo menos $f+1$ processos corretos obterão x da execução de r.*

Demonstração. Prova (esboço). Devido ao axioma 11 e ao lema 2, todas as réplicas iniciam no mesmo estado, fazem as mesmas transições de estado e executam requisições na mesma ordem. Portanto, se não houver $f+1$ réplicas corretas, existirá uma contradição com o axioma 1. □

Teorema 1. *Ordem total. Todas as requisições são executadas na mesma ordem por todas as réplicas corretas.*

Demonstração. Prova (esboço). Conforme os lemas 1, 2 e 3 todas as requisições são, em algum momento, entregues, executadas na mesma ordem e respondidas ao cliente por pelo menos $f+1$ réplicas corretas. Portanto, todas as requisições são efetuadas em ordem total por uma maioria de réplicas corretas no sistema, provando que o RegPaxos apresenta a propriedade de *safety*. □

Teorema 2. *Terminação. O RegPaxos sempre faz progresso.*

Demonstração. Prova (esboço). Devido aos axiomas 7, 8 e 9 o atraso na entrega das requisições não pode ser indefinido, além disso, o cliente não espera infinitamente para que sua requisição seja respondida. O lema 3 demonstra que sempre existirão $f+1$ réplicas corretas para manter o funcionamento e progresso do sistema. O lema 1 mostra que mesmo que o líder seja malicioso, a sucessão de trocas de líder ocorrerá até que um novo líder seja correto, e por isso haverá um consenso, caso contrário ocorrerá uma contradição com o axioma 1. Desta forma, demonstra-se a presença no RegPaxos da propriedade de *liveness*. □

4.3 DIFATO - DIFUSÃO ATÔMICA TOLERANTE A FALTAS

A dificuldade em construir sistemas distribuídos pode ser drasticamente reduzida através do uso de primitivas de comunicação em grupo, tal como a difusão atômica (ou difusão com ordem total) (DÉFAGO; SCHIPER; URBÁN, 2004). A difusão atômica assegura que mensagens enviadas para um conjunto de processos serão entregues por estes na mesma ordem, sendo empregada nos mais diversos domínios de aplicação como: sincronização de relógios, CSCW (*Computer Supported Cooperative Work*), memórias distribuídas, replicação de base de dados (RODRIGUES; VERÍSSIMO; CASIMIRO, 1993; KEMME et al., 2003; BESSANI; FRAGA; LUNG, 2006), e é a base para abordagens de replicação de máquina de estados (SCHNEIDER, 1990b), e também o componente principal de muitos sistemas tolerantes a faltas (CASTRO; LISKOV, 2002; YIN et al., 2003; CORREIA; NEVES; VERÍSSIMO, 2006; FAVARIM et al., 2007).

A literatura nos mostra uma quantidade considerável de trabalhos sobre difusão com ordem total, com as mais variadas abordagens e algoritmos. Entretanto, em sua maioria, estes algoritmos consideram modelos de sistema sujeitos apenas a faltas de parada (i.e. *crash*) (DÉFAGO; SCHIPER; URBÁN, 2004; EKWALL; SCHIPER; URBÁN, 2004), de modo que poucos são os trabalhos que endereçam faltas arbitrárias/Bizantinas (CORREIA; NEVES; VERÍSSIMO, 2006; REITER, 1994). Em geral, as abordagens usam algoritmos de consenso para estabelecer um acordo acerca da ordenação das mensagens, e necessitam de, pelo menos, $3f+1$ processos envolvidos no procedimento. Por outro lado, alguns trabalhos propõem a separação do consenso do acordo, o que dá origem a um serviço de consenso (GUERRAOUI; SCHIPER, 2001; PIERI; FRAGA; LUNG, 2010).

Esta seção apresenta DIFATO (**Dif**usão **A**tômica **To**lerante a Faltas Bizantinas), um serviço de consenso com o propósito de difundir as mensagens atomicamente, a despeito de faltas Bizantinas. Com o uso dos registradores compartilhados distribuídos é possível melhorar a resiliência do sistema, requerendo apenas $2f+1$ servidores para compor o serviço de consenso.

É importante ressaltar que o RegPaxos e o DIFATO são protocolos completamente independentes. Não existe nenhuma ligação entre ambos, são dois problemas distintos que são resolvidos através da utilização do serviço de consenso aqui proposto.

4.3.1 Propriedades da Difusão Atômica

O problema de difusão atômica, ou difusão confiável com ordem total, consiste em garantir a entrega de um conjunto de mensagens, na mesma ordem, para todos os processos que fazem parte de um sistema. A definição em um contexto Bizantino pode ser feita, considerando as seguintes propriedades:

DA1 Validade - Se um processo correto difunde uma mensagem m, então algum processo correto, em algum momento, entrega m.

DA2 Acordo - Se um processo correto entrega uma mensagem m, então todos os processos corretos , em algum momento, entregam m.

DA3 Integridade - Para qualquer mensagem m, todo processo correto entrega m no máximo uma vez, e se o remetente de m for correto, então m foi anteriormente difundida por este remetente.

DA4 Ordem total - Se dois processos corretos entregam duas mensagens com

os prefixos m_{i-1} e m_i, então ambos os processos entregam as duas mensagens de maneira que m_{i-1} antecede m_i.

4.3.1.1 Execução Normal do Protocolo

Passo 1) O procedimento inicia-se quando algum cliente c_i envia a mensagem $\langle ORDER, m, t, v\rangle_{\sigma ci}$ para o sequenciador com sua mensagem m incluída. O campo t é a marca de tempo da mensagem para assegurar a semântica de apenas uma ordenação por mensagem. Desta maneira os servidores só executam a ordenação de mensagens cuja marca de tempo seja maior que a anterior, para um mesmo cliente. O campo v é o vetor que gera um MAC por servidor, cada um obtido através da chave compartilhada entre clientes e servidores. Portanto, cada servidor pode testar a integridade da mensagem utilizando este vetor. Caso uma mensagem já tenha sido ordenada, o servidor apenas a reenvia para o cliente.

Algoritmo 4: Algoritmo executado pelo nó sequenciador.

```
Constants:
f : int // Maximum tolerated faults
T : int // Maximum waiting time for a proposal to be decided
Variables:
accepted : int // counter of acceptance for some ordering
1  upon receive ⟨ORDER,m,t_j,v⟩_σci from client
2  if t_j ≤ t_{j−1} for c_i then
3  |  if has(m) into buffer then
4  |  |  rmulticast( getOrdered( m ) from buffer );
5  |  end
6  |  return;
7  end
8  if isWrong( v ) then
9  |  return;
10 end
11 write( ⟨PROPOSE,n_o,⟨ORDER,m,t_j,v⟩_σci,mac⟩_σsi ) into DSR;
12 accepted = waitForAcceptance( T );
13 if accepted ≥ f + 1 then
14 |  store ⟨PROPOSE,n_o,⟨ORDER,m,t_j,v⟩_σci,mac⟩_σsi in the atomic buffer;
15 end
16 return;
```

Passo 2) Depois de verificar se o MAC em v é correto e se a marca de tempo é válida para a mensagem do cliente, o sequenciador gera uma mensagem $\langle PROPOSE, n_o, o, mac\rangle_{\sigma si}$ onde o representa a mensagem original do cliente, n_o é o número de ordenação para o e mac é o MAC gerado pelo sequenciador. Os DSR automaticamente identificam a mensagem proposta com o id do sequenciador. O sequenciador espera pela aceitação dos demais servidores, isto é, f processos concordando com a pro-

posta. Ao ter a mensagem aceita, o sequenciador salva a mensagem e a ordenação em seu *buffer*. Este comportamento pode ser observado no algoritmo 4. Como foi discutido, todas as mensagens escritas nos DSR serão entregues se o destinatário e o remetente não sofreram *crash*.

Passo 3) Ao receber uma proposta, o servidor s_k a valida: (i) s_k verifica, usando o vetor de MACs, se o conteúdo da mensagem m está correto e (ii) verifica se não existe outra proposta anteriormente aceita para o número de ordenação n. Depois de aceitar a proposta, s_k escreve uma mensagem $\langle ACCEPT, n_o, h_m, mac \rangle_{\sigma sk}$ nos registradores. Esta mensagem possui o *hash* da mensagem do cliente h_m, o número de ordenação aceito e o MAC *mac* gerado pelo servidor. Após escrever a mensagem de aceite, o processo aguarda por $f-1$ mensagens de aceitação para, então, salvar a mensagem no *buffer*. Este comportamento pode ser visto no algoritmo 5.

Passo 4) O sequenciador difunde de maneira confiável a mensagem com o número de ordenação e um vetor de MACs assinado por pelo menos $f+1$ servidores diferentes que aceitaram a ordem proposta. Após receber e validar o vetor, os clientes finalmente aceitam a mensagem e a entregam na ordem estipulada. A figura 8 mostra estes quatro passos.

Algoritmo 5: Algoritmo executado pelos nós não sequenciadores.

```
   Constants:
   f : int // Maximum tolerated faults
   T : int // Maximum waiting time for a proposal to be decided.
   Variables:
   accepted : int // counter of acceptance for some ordering
1  upon read ⟨PROPOSE, n_o, ⟨ORDER, m, t_j, v⟩_σci, mac⟩_σss from DSR
2  if isValid( v ) and isValid( n ) and t_j > t_{j-1} for c_i then
3  |    write( ⟨ACCEPT, n_o, h_m⟩_σsi ) into DSR;
4  |    accepted = waitForAcceptance( T );
5  |    if accepted ≥ f + 1 then
6  |    |    store ⟨PROPOSE, n_o, ⟨ORDER, m, t_j, v⟩_σci, mac⟩_σss in the atomic buffer;
7  |    end
8  else
9  |    write( ⟨CHANGE, h_m, s_s⟩_σsi ) into DSR and bufferize;
10 end
11 return;
```

4.3.2 Execução do Protocolo na Ocorrência de Faltas

A operação na presença de faltas implica que uma mudança de sequenciador ocorrerá, portanto, fazemos aqui uma breve explicação de como

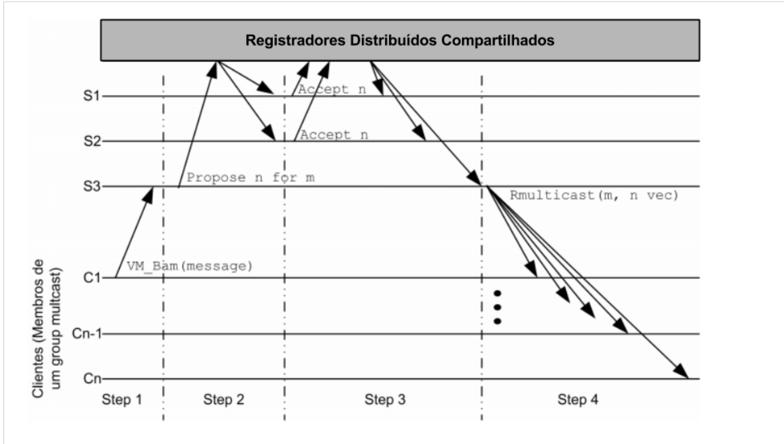


Figura 8 – Fluxo da difusão atômica.

isso se desenvolve.

Mudança de sequenciador: Durante a configuração do sistema, todos os servidores recebem um número de identificação. Estes números são sequenciais e iniciam em zero. Todos os servidores conhecem o identificador S do sequenciador e o número total de servidores no sistema. Quando $f+1$ servidores corretos suspeitam do sequenciador atual, eles simplesmente definem $S = S+1$ como o próximo sequenciador, se $S < n-1$, senão $S = 0$.

Algoritmo 6: Mudança de líder

```
    Constants:
    f : int // Maximum tolerated faults
    n : int // The number of replicas in the system
    Variables:
    leader : int // The actual leader ID
    changeCounter : int // has the count of the different servers that suspect from the leader
 1  upon suspect s_s or read ⟨CHANGE, h_m, s_s⟩_σs from DSR
 2  bufferize(⟨CHANGE, h_m, s_s⟩_σs);
 3  changeCounter = searchFor(buffer, h_m, s_s);
 4  if changeCounter ≥ f + 1 then
 5  |   leader++;
 6  |   if leader == n then
 7  |   |   leader = 0;
 8  |   end
 9  end
10  return;
```

Ao validar uma proposta, o servidor s_k verifica, usando o MAC no vetor v, se o conteúdo da mensagem está correto. Se o conteúdo estiver correto

e se o número de ordenação estiverem corretos, o servidor aceita a proposta. Caso contrário, s_k vai solicitar uma mudança de sequenciador:

1. Um servidor correto pode entrar em modo faltoso de operação de duas maneiras:

   (a) Quando o servidor s_k lê dos registradores uma mensagem de mudança de sequenciador, mas ainda não suspeita do servidor s_s. O servidor apenas armazena a mensagem em seu *buffer* local para utilização futura.

   (b) Se o processo s_k suspeita do sequenciador s_s com relação à mensagem m, s_k escreve uma mensagem $\langle CHANGE, sid, h_m, s_s \rangle_{\sigma sk}$ nos DSR contendo o *sid* como seu próprio identificador, o *hash* da mensagem h_m que originou a suspeita e o identificador s_s do servidor em suspeita.

2. O servidor s_k inicia uma busca em seu *buffer* local, no intuito de encontrar $f+1$ mensagens de mudança relacionadas à mensagem m e ao servidor s_s. Caso s_k encontre $f+1$ (incluindo o próprio s_k) diferentes *sid* para a mesma mensagem, então o servidor efetua a mudança de sequenciador. Se o novo sequenciador for o próprio servidor s_k, então o servidor vai reiniciar a ordenação baseando-se nas mensagens já aceitas nos DSR. Caso não seja s_k o novo sequenciador, s_k apenas aguarda pelas novas propostas. O algoritmo 6 mostra como é efetuada a mudança de sequenciador em cada réplica.

Com a escolha de um novo sequenciador, o protocolo faz progresso como ocorre na operação normal, isto é, na ausência de faltas.

4.3.3 Provas de corretude

Em 4.3.1 foram apresentadas as propriedades de atomicidade, o protocolo definido precisa cumprir de DA1 a DA4:

Os seguintes axiomas discutidos anteriormente são necessários para auxiliar nas provas:

Axioma 1. *Apenas uma minoria de réplicas pode falhar em suas especificações. Para cada f réplicas faltosas, existem $f+1$ corretas.*

Axioma 2. *Toda réplica que se desvia do comportamento determinado é considerada faltosa.*

Axioma 3. *A máquina hospedeira é inacessível através máquina hospedada.*

Axioma 4. *Todos os processos tem acesso irrestrito ao espaço de registradores, e, portanto, leem os mesmos registros.*

Axioma 5. *Nenhum registro pode ser alterado dentro do DSR.*

Axioma 6. *Não existe concorrência - dentro de cada hospedeiro - no acesso ao espaço de registradores. Apenas um processo efetua as operações.*

Axioma 7. *Existe um tempo máximo Δ_t para entrega das mensagens no DSR.*

Axioma 8. *Sendo um modelo parcialmente síncrono, existe um tempo máximo - que é variável e desconhecido - para entrega das mensagens na rede.*

Axioma 9. *O cliente não espera indefinidamente pela resposta de sua requisição. O cliente reenvia sua requisição periodicamente até que receba uma resposta válida.*

Axioma 10. *Não é possível a falsificação de mensagens por uma réplica maliciosa.*

Axioma 11. *O serviço executado pelas réplicas é determinístico e todas as réplicas iniciam o protocolo no mesmo estado.*

Após declaradas estas premissas, podemos dar continuidade às provas de corretude.

Lema 1. *Toda requisição r correta enviada por algum cliente será, em algum momento, entregue, ordenada, e difundida para os clientes.*

Demonstração. Prova (esboço). Devido aos axiomas 7, 8 e 9, e se a a réplica líder for correta, então todas as réplicas receberão a requisição do cliente em algum momento. Caso a réplica líder não seja correta, então são efetuadas trocas de líder até que uma correta receba a liderança.

Segundo o algoritmo 4 (linha 11), a réplica líder enviará uma requisição a todas as réplicas do sistema, contendo, não somente a mensagem do cliente, mas também a ordem proposta por este líder. Se a ordem for considerada correta em consenso, todas as réplicas vão assinar o vetor e vão guardar a mensagem em seus *buffers* como pode ser visto nas linhas 3 à 7 do algoritmo 5. O sequenciador vai guardar também a mensagem em seu *buffer* e vai entregá-la na ordem estipulada, como pode ser visto no algoritmo 4 (linha 13). Se a proposta não for considerada correta, então haverá a troca de líder até que se atinja o consenso em relação a alguma ordenação, como pode ser visto nos algoritmos 5 (linha 9) e 6. □

Teorema 1 (DA1). *Validade - Se um processo correto difunde uma mensagem m, então algum processo correto, em algum momento, entrega m.*

Demonstração. Prova (esboço). Segundo o lema 1, um cliente correto que difundir uma mensagem para os servidores, terá sua mensagem ordenada e difundida para todos os clientes. Portanto, se este cliente é correto, ele receberá sua mensagem ordenada e vai entregá-la na ordem definida pelo serviço de consenso atômico. □

Teorema 2 (DA2). *Acordo - Se um processo correto entrega uma mensagem m, então todos os processos corretos , em algum momento, entregam m.*

Demonstração. Prova (esboço). Segundo o lema 1 e o teorema 1, uma mensagem difundida por um cliente c_i correto é ordenada e é enviada para todos os clientes. Se um cliente correto c_j recebe do serviço de consenso atômico a mensagem de c_i ordenada, e se a mensagem estiver corretamente assinada, então c_j vai entregá-la, assim como todos os demais clientes corretos. □

Teorema 3 (DA3). *Integridade - Para qualquer mensagem m, todo processo correto entrega m no máximo uma vez, e se o remetente de m for correto, então m foi anteriormente difundida por este remetente.*

Demonstração. Prova (esboço). Pelo Algoritmo 4, especificamente nas linhas 2 à 7 é possível observar que o sequenciador efetua a ordenação das mensagens apenas uma vez, portanto, uma mensagem tem um, e somente um, valor de ordenação. De acordo com os teoremas 1 e 2, todos os clientes receberão a mensagem difundida. Com isso, os clientes corretos entregam as mensagens apenas uma vez, e na ordem estipulada. E como pode ser visto nas linhas 8 à 10 do Algoritmo 4, e na linha 2 do Algoritmo 5, para garantir que uma mensagem só pode ter sido difundida por seu remetente, o vetor de MACs enviado com a mensagem é validado por cada servidor, a fim de evitar que um cliente ou um servidor possa se passar por outro cliente. Estas asserções provam a propriedade de integridade. □

Teorema 4 (DA4). *Ordem total - Se dois processos corretos entregam duas mensagens com os prefixos m_{i-1} e m_i, então ambos os processos entregam as duas mensagens de maneira que m_{i-1} antecede m_i.*

Demonstração. Prova (esboço). Pelo que se pode verificar nas linhas 11 à 15 do Algoritmo 4, e 1 à 8 do Algoritmo 5, respectivamente, se observa que o sequenciador apenas efetua a difusão confiável da mensagem após ter sido executado o consenso. As mensagens difundidas são aquelas em que a ordenação foi aceita por pelo menos $f+1$ servidores, e a ordem de entrega é determinística. Como o sequenciador difunde a mensagem de maneira confiável, juntamente com sua ordenação e o vetor de MACs para atestar que aquela ordem é aceita pela maioria, consequentemente todos os processo corretos

entregarão a mensagem na ordem estipulada, o que prova a propriedade em DA4. □

4.4 CONSIDERAÇÕES FINAIS

Neste capítulo foi introduzido um modelo híbrido que faz uso de virtualização e compartilhamento de memória para criar sistemas tolerantes a faltas Bizantinas. Com esta abordagem, é possível criar sistemas de maneira simples e com baixa resiliência - $n \geq 2f + 1$ réplicas - capazes de manter seu funcionamento correto a despeito da ocorrência de falhas.

Outras abordagens são capazes de atingir o mesmo resultado do ponto de vista da criação de um modelo híbrido, entretanto, adicionam complexidade e dependência ao exigirem mudanças no *kernel* dos sistemas operacionais e/ou criação de *hardwares* específicos para fazerem o componente inviolável. A utilização de virtualização mostrou-se uma excelente opção para criar o isolamento entre os subsistemas e os Registradores Compartilhados Distribuídos simplificaram ao máximo a comunicação entre as réplicas do consenso.

5 RESULTADOS EXPERIMENTAIS E ANÁLISES

Para validar as abordagens propostas, foram implementados os serviços descritos anteriormente - RegPaxos e DIFATO - e executados sob diferentes condições. Neste capítulo apresentaremos detalhes da implementação e os resultados obtidos em diferentes situações. Os testes foram elaborados para salientar os pontos positivos e também mostrar os problemas do modelo híbrido utilizando os Registradores Compartilhados Distribuídos. Primeiramente serão descritos os detalhes do ambiente em que foram realizados os testes, na seção 5.1. A análise e os resultados dos testes são apresentados na seção 5.2.

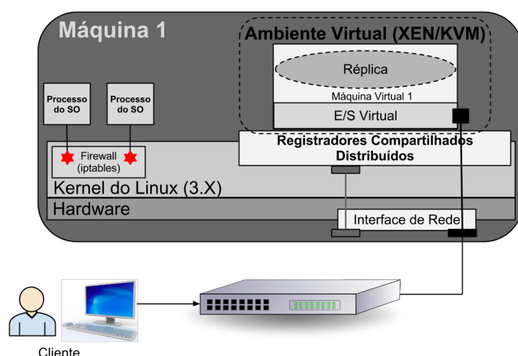


Figura 9 – Camadas dos protótipos

5.1 DETALHES DA IMPLEMENTAÇÃO

A figura 9 ilustra as camadas dos protótipos. Em parte das réplicas foi utilizado o VMM *hypervisor* KVM (KERNEL-BASED..., 2012) e nas demais foi utilizado o *hypervisor* XEN (XEN..., 2012). A escolha destes VMM foi devido a pesquisas sobre testes em *benchmarks*, onde ambos apresentavam os melhores resultados. Além disso, se dá também a facilidade de instalação e integração com ambientes linux, e o fato ambos possuirem uma comunidade ativa buscando sempre atualizações, dando robustez, segurança e a possibilidade de melhorias continuas.

Os sistemas operacionais executados nas máquinas virtuais foram versões diferentes de linux com diferentes versões de *kernel*. O objetivo é dimi-

nuir a chance de falhas e intrusões através da exploração de vulnerabilidades em comum. Em função das escolhas dos ambientes de virtualização, não foram utilizados outros sistemas operacionais, e também por se tratar de um protótipo, entretanto, na utilização do modelo em um ambiente de produção, é importante prezar pela heterogeneidade do sistema. Os clientes foram variados, com sistemas operacionas Windows, Linux, e Mac OS. Os algoritmos foram implementados usando a linguagem Java, JDK 1.6.x. Os canais de comunicação foram implementados usando *sockets* TCP da API NIO.

5.2 TESTES E ANÁLISES SOBRE OS PROTÓTIPOS

Para os testes do protocolo RegPaxos foi desenvolvida uma aplicação que atua como uma calculadora simples, isto é, efetua as operações básicas, divisão, multiplicação, adição e subtração. Os clientes enviam requisições com um valor e a operação desejada. Todos os servidores são iniciados e tem um valor gerado aleatoriamente que é igual para todos. Ao receber a requisição de um cliente, os servidores efetuam a operação que o cliente solicitou, utilizando o valor base e o valor enviado pelo cliente. O valor base é substituído pelo resultado da operação e o resultado é enviado para o cliente. A substituição do valor base pelo resultado faz com que o estado da aplicação seja modificado. Os clientes aguardam por $f+1$ resultados iguais vindos de diferentes servidores.

No caso do DIFATO, o serviço executado é a própria ordenação das mensagens dos clientes.

5.2.1 Ambiente de testes

No ambiente de testes a rede de *payload* - comunicação entre servidores e clientes - se dá através de uma rede local com 100Mbps. Os teste foram feitos em uma rede isolada e os computadores foram interconectados por um *switch*. As máquinas utilizadas eram todas *multicore* com 8GB de memória RAM. Cada máquina virtual recebeu 4GB de RAM e o compartilhamento de arquivos foi feito através de uma biblioteca Python de *Network File System* (NFS).

5.2.2 Considerações Sobre os Testes Realizados

Na realização dos testes, foi considerado que o número de réplicas maliciosas seria igual a um, isto é, $f = 1$. Este valor foi estipulado em função da escassez de recursos computacionais para a realização dos testes. Os testes foram realizados considerando-se um cenário (1) livre da ocorrência de faltas e um outro cenário (2) onde uma réplica atuava de maneira maliciosa. Os testes ainda trazem a variação da quantidade de clientes para avaliar o desempenho sob concorrência. Todos os testes foram executados em rodadas com um total de mil requisições, independentemente da quantidade de clientes foram feitas 5 rodadas considerando cada cenário e a média aritmética destas rodadas foi utilizada nas análises. O tempo de resposta foi calculado localmente em cada cliente, considerando o momento em que o mesmo enviou a requisição até o momento em que recebeu as $f+1$ respostas de diferentes servidores (*Round-Trip Time*).

5.2.3 Execução normal

Este cenário tem o objetivo de mostrar o tempo de resposta dos protótipos sem influência de agentes maliciosos, isto é, sem falhas nem faltas. Este cenário mostra o comportamento do protocolo. O comportamento do RegPaxos e do DIFATO se mostrou muito semelhante neste cenário, quando considerado apenas 1 cliente, portanto, o gráfico de dispersão 10 usou os dados unificados e representa ambos os protocolos. O gráfico mostra que os valores de ambas as execuções giram em torno de 4 milissegundos. E que, apesar de haver alguma dispersão e alguns pontos discrepantes, os valores ficam realmente concentrados em torno deste valor.

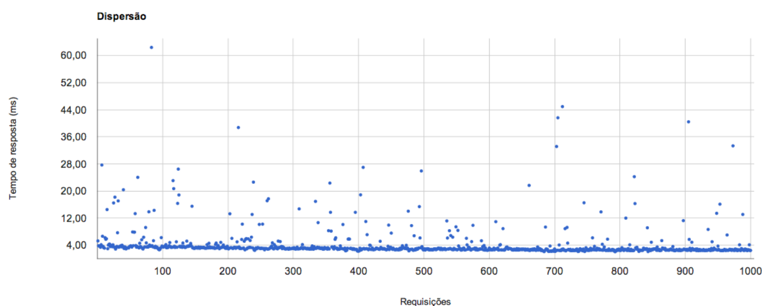


Figura 10 – RegPaxos e DIFATO - Dispersão com 1 cliente e sem faltas

O histograma da distribuição geral dos tempos de resposta das requisições é apresentado na Figura 11. No eixo ”x”são apresentados os blocos, que correspondem aos tempos de resposta das requisições em milissegundos (ms), enquanto no eixo ”y”apresenta a frequência com que estes tempos de respostas aparecem dentro da amostra de 1000 requisições. Podemos observar que os valores variam até 60ms, porém, como o esperado, a grande maioria das requisições fica concentrada próxima da média.

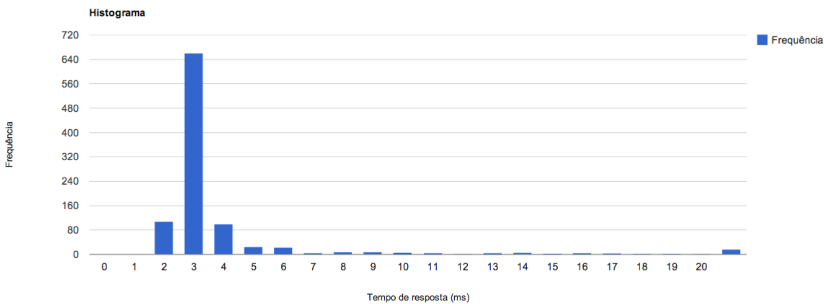


Figura 11 – RegPaxos e DIFATO - Histograma com 1 cliente e sem faltas

Para entender melhor o comportamento dos protótipos em um ambiente com maior concorrência, aumentou-se o número de clientes para 10. Como é possível ver, em ambos os casos, o ambiente de concorrência afeta diretamente na média de tempo de execução. Entretanto, no caso do DIFATO, essa mudança é muito mais visível. Isto é facilmente explicado devido ao fato do DIFATO (Figuras 13) exigir mais comunicações entre os servidores e os clientes, já que a resposta de uma ordenação tem que ser difundida de maneira confiável para todos os clientes, diferentemente do RegPaxos (Figuras 12), onde a resposta da execução de uma tarefa é devolvida apenas para o cliente que a requisitou.

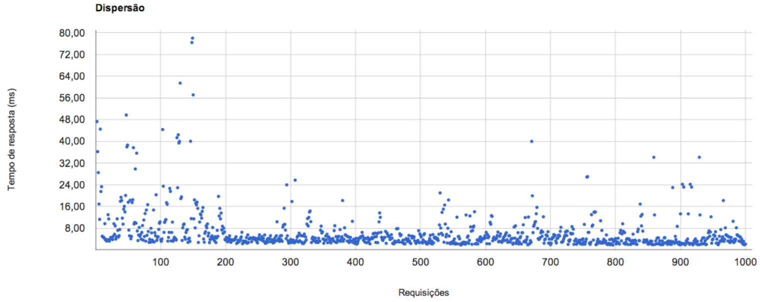


Figura 12 – RegPaxos - Dispersão com 10 clientes e sem faltas

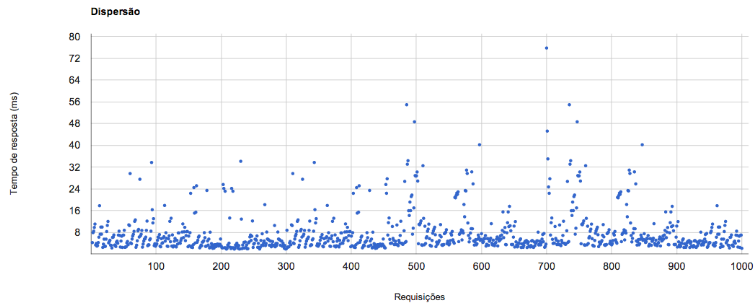


Figura 13 – DIFATO - Dispersão com 10 clientes e sem faltas

Apesar dos indícios destes comportamentos diferenciados nos gráficos de dispersão, os histogramas 14 e 15 ainda apresentam uma concentração muito semelhante. A partir destes dados, a informação ainda é inconclusiva, não sendo possível afirmar que realmente no caso do DIFATO a deterioração foi mais grave.

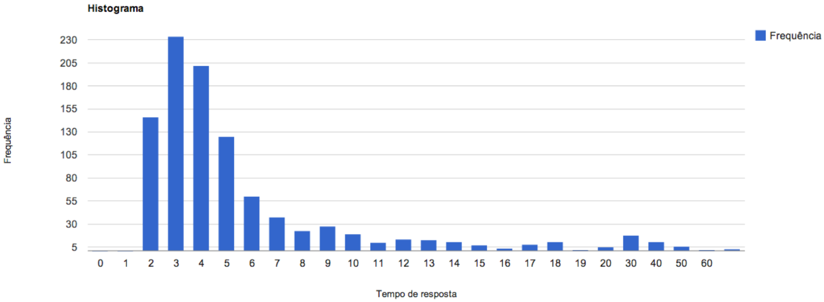


Figura 14 – RegPaxos - Histograma com 10 clientes e sem faltas

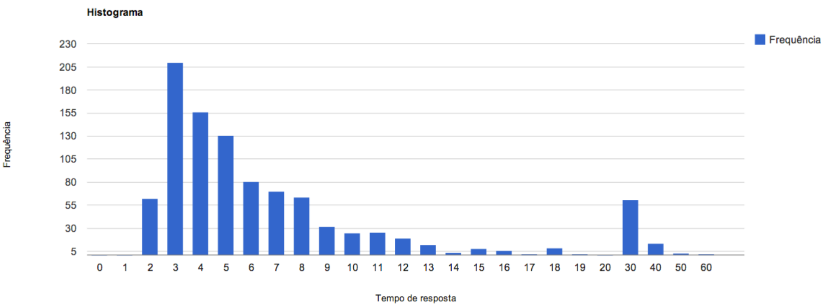


Figura 15 – DIFATO - Histograma com 10 clientes e sem faltas

Para verificar se a quantidade de cliente tem maior influência no protocolo DIFATO, foi feito um terceiro teste com 20 clientes.

O curioso neste terceiro teste foi a evidência de que no caso do RegPaxos os valores ficam mais dispersos afetando de maneira mais isolada cada cliente em espera pela resposta da requisição. Entretanto, no caso do DIFATO, houve o aumento da dispersão, mas também é possível notar que a média subiu, é possível ver que os valores agora giram em torno de 11 milissegundos de forma mais homogênea. As figuras 16 e 17 mostram este comportamento.

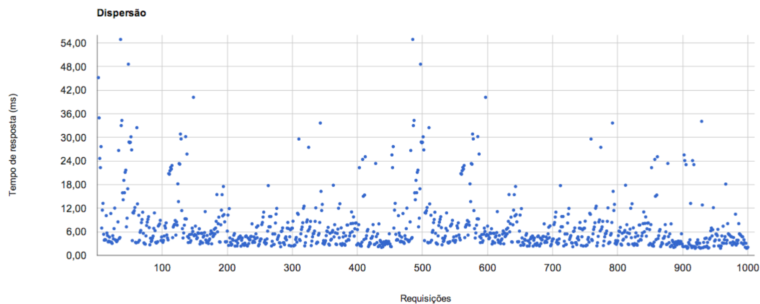


Figura 16 – RegPaxos - Dispersão com 20 clientes e sem faltas

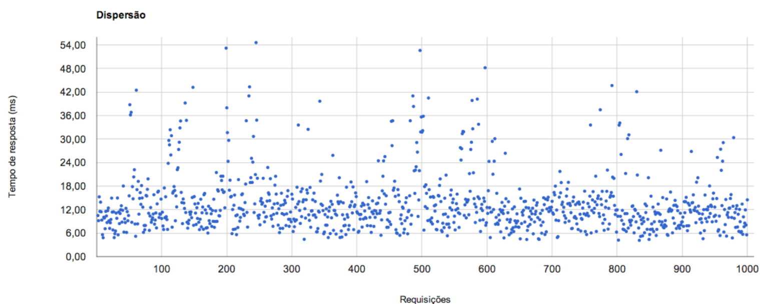


Figura 17 – DIFATO - Dispersão com 20 clientes e sem faltas

Os histogramas de ambos os protótipos agora mostram valores bem diferentes e é possível ver que no caso do RegPaxos a figura 18 mostra o aumento da dispersão. Porém, no caso do DIFATO, a figura 19 mostra um comportamento de dispersão e elevação da média, com os valores distribuídos em torno dos 11 milissegundos.

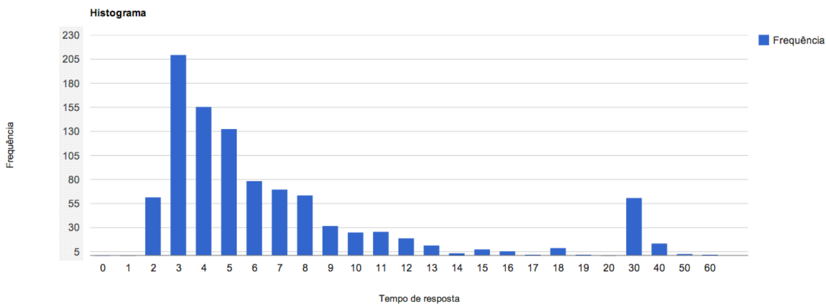


Figura 18 – RegPaxos - Histograma com 20 clientes e sem faltas

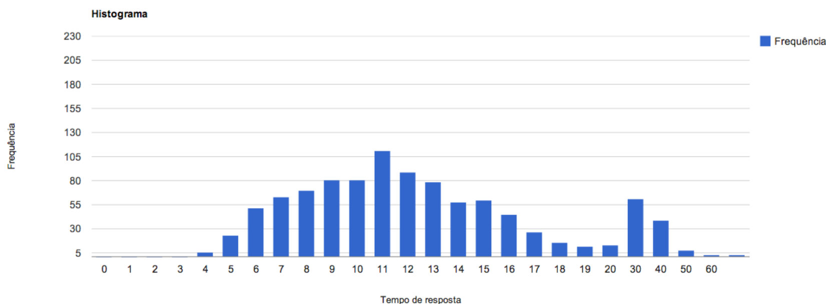
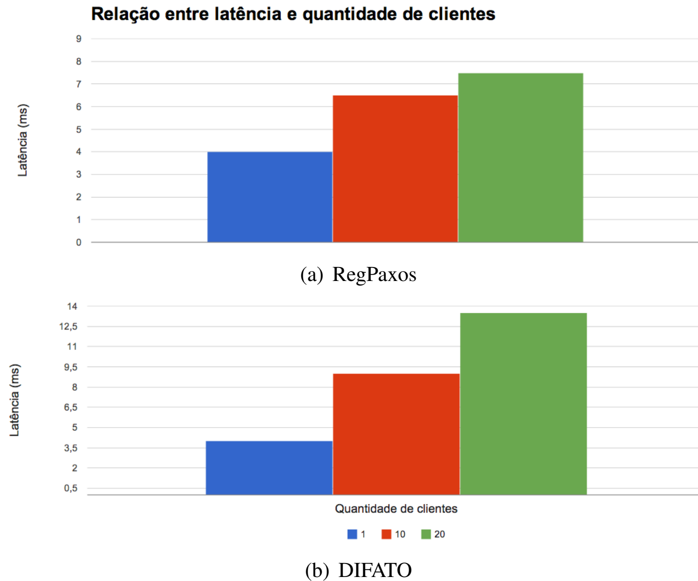


Figura 19 – DIFATO - Histograma com 20 clientes e sem faltas

O gráfico 20 mostra a relação entre a quantidade de clientes e sua influência na latência dos sistemas (*Round-Trip Time*). Como pode ser visto, a quantidade de clientes aumenta a latência de maneira considerável em ambos os protótipos. Mas, como pode ser visto, no caso do RegPaxos (figura 20(a)) a latência com 20 clientes não chega nem ao dobro da latência com apenas 1 cliente, enquanto que no caso do DIFATO figura(20(b)) o valor chega a triplicar, considerando-se a mesma variação no número de clientes.

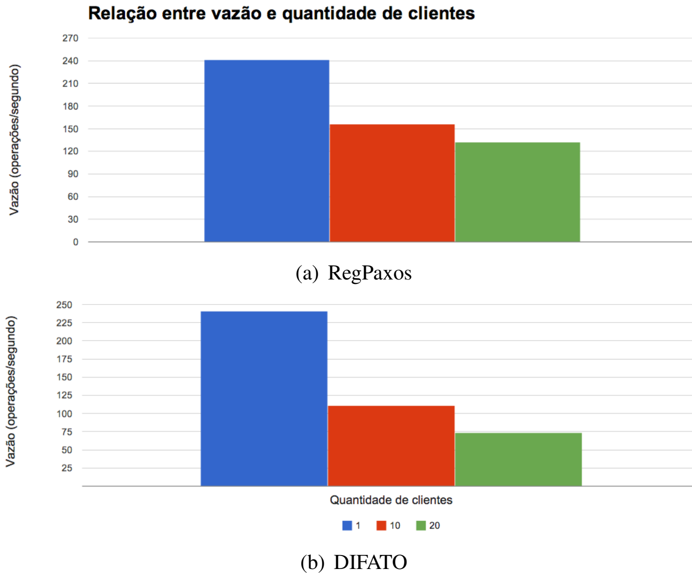


(a) RegPaxos

(b) DIFATO

Figura 20 – Quantidade de clientes e a influência na latência

O gráfico 21 mostra a relação entre a quantidade de clientes e a vazão dos sistemas. Como é possível avaliar, o número de clientes influencia muito fortemente na quantidade de operações realizadas por segundo em ambos os sistemas. A figura 21(a) mostra que o protótipo do RegPaxos com 20 clientes teve uma piora de desempenho de mais de 45% se comparado com o mesmo algoritmo operando com apenas 1 cliente. Já a figura 21(a) mostra que o protocolo proposto em DIFATO teve uma diminuição de mais de 69% de sua capacidade quando operando com 1 e 20 clientes.



(a) RegPaxos

(b) DIFATO

Figura 21 – Quantidade de clientes e a influência na vazão

5.2.4 Execuções com falhas

Diferentemente da seção anterior, onde apresentamos os protótipos operando no caso otimista (ou sem falhas), nesta seção é mostrado o comportamento dos algoritmos sob diferentes cenários de falhas. Nestes cenários, réplicas começam a se comportar de maneira arbitrária, apresentando não somente respostas erradas, mas também se omitindo e atuando de maneira maliciosa, no intuito de atrapalhar o funcionamento dos sistemas.

Assim como em outras abordagens de tolerância a faltas presentes na literatura, em nosso sistema as falhas são limitadas - até f falhas - para que o protocolo possa se manter em progresso. Por esta limitação e também pela limitação da quantidade de recursos disponíveis para os testes, em nossos ambientes tomamos o cuidado para garantir que mais que f réplicas não viessem a falhar, comprometendo os sistemas, e por sua vez, os testes.

$f_{\%}$	N_f	Δ_r
1	10	99
10	100	9
25	250	3
50	500	1

Tabela 1 – Distribuição das faltas

Para verificar adequadamente o comportamento dos sistemas na presença de faltas, foram montados diferentes distribuições percentuais de falhas. A tabela 1 representa esta distribuição. A primeira coluna - $f_{\%}$ - traz o valor percentual de falhas que ocorrerão. A segunda coluna - N_f - representa o número total de faltas que ocorrerão durante as 1000 execuções de cada protótipo. A última coluna - Δ_r - representa o número de execuções corretas até que uma execução faltosa ocorra. Em função da necessidade de controlarmos a quantidade de faltas em cada experimento, optamos por simular as faltas em requisições pré-determinadas, o que também facilita na visualização dos resultados.

Por questão de facilidade, consideramos apenas um cliente na execução dos protótipos. O comportamento de ambos os algoritmos é semelhante quando se trata de apenas um cliente. Desta forma, os resultados apresentados são médias da execução de ambos os algoritmos, foram duas execuções de cada e os valores são extraídos da média aritmética das quatro execuções.

5.2.4.1 Experimento 1% faltas

Os experimentos com faltas utilizaram apenas um cliente, portanto, os resultado aqui apresentados são também uma média da execução tanto do RegPaxos quanto do DIFATO. As figuras 22 e 23 mostram o comportamento do algoritmo quando ocorre 1 falta a cada 100 execuções. Como havia uma pequena dispersão, mesmo na ausência de faltas, este experimento tem uma variação quase que imperceptível e este valor nem chega a se refletir de maneira contundente no histograma.

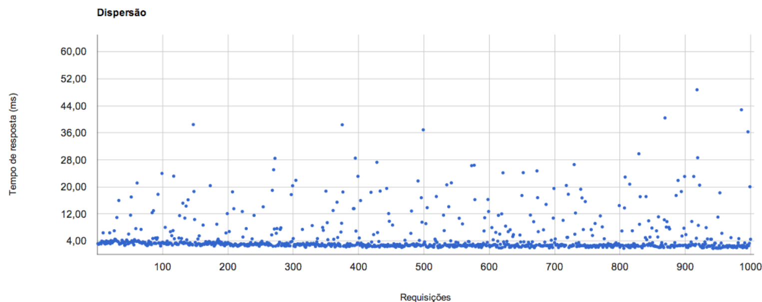


Figura 22 – Dispersão com 1% de faltas

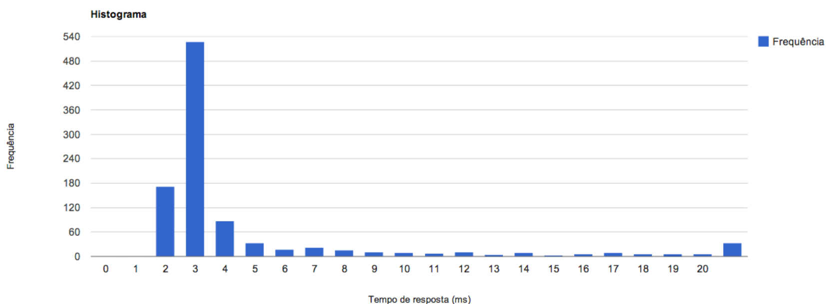


Figura 23 – Histograma com 1% de faltas

5.2.4.2 Experimento 10% faltas

Neste experimento já se torna mais clara a dispersão dos valores. Nele, já é possível visualizar uma leve diminuição dos valores 2, 3 e 4 no histrograma (figura 25) e é possível ver também na figura 24 que, apesar de suavemente, os valores mais altos começam a se concentrar próximos aos momentos de faltas.

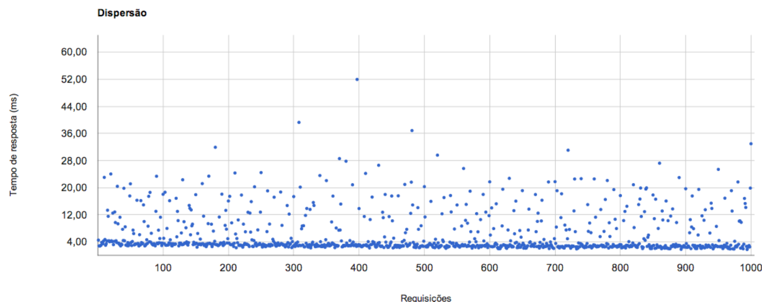


Figura 24 – Dispersão com 10% de faltas

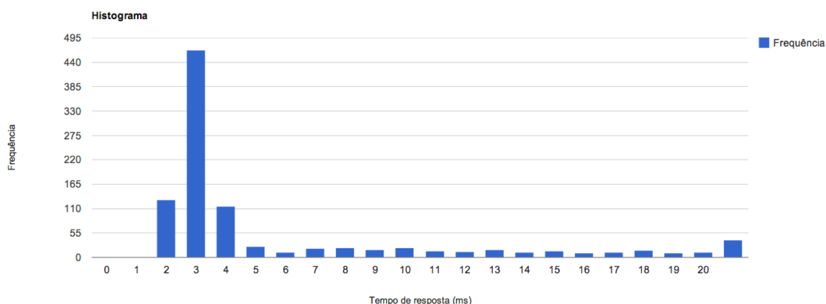


Figura 25 – Histograma com 10% de faltas

5.2.4.3 Experimento 25% faltas

Neste cenário a ocorrência de faltas é bem numerosa, já que 1 em cada 4 execuções é faltosa. Portanto, esperava-se que o resultado fosse ainda mais visível. Este fenômeno é bem menos visual do que se esperava e a ocorrência desta quantidade de faltas afeta de maneira geral o sistema fazendo com que a dispersão seja mais homogênea do que o esperado, como pode ser visto no histograma 27. O comportamento neste experimento é de aumento da média com a criação de duas separações bem claras na figura 26.

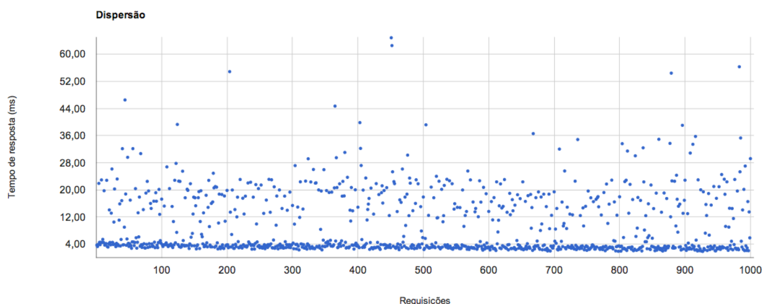


Figura 26 – Dispersão com 25% de faltas

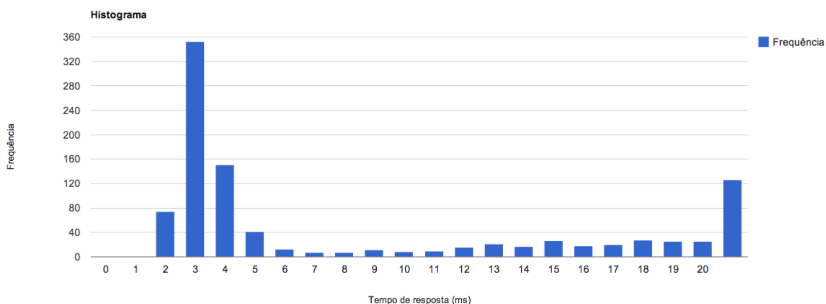


Figura 27 – Histograma com 25% de faltas

5.2.4.4 Experimento 50% faltas

Este experimento traz um valor bem elevado de faltas tendo apenas uma execução correta para cada execução faltosa. O valor da média de resposta de uma requisição triplicou em relação ao cenário sem faltas. É possível ver no histograma da figura 29 que a quantidade de valores acima de 12 milissegundos é equivalente, senão maior, que a quantidade de valores a baixo, mostrando a drástica diminuição da frequência dos valores originais dos protótipos. Além disso, na figura 28 é possível ver que a média está subindo e começa a se concentrar entre 12 e 28 milissegundos.

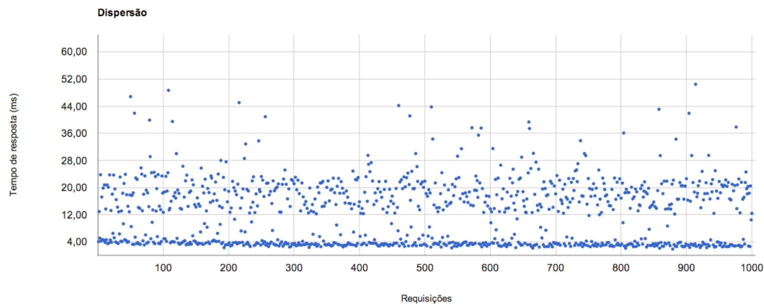


Figura 28 – Dispersão com 50% de faltas

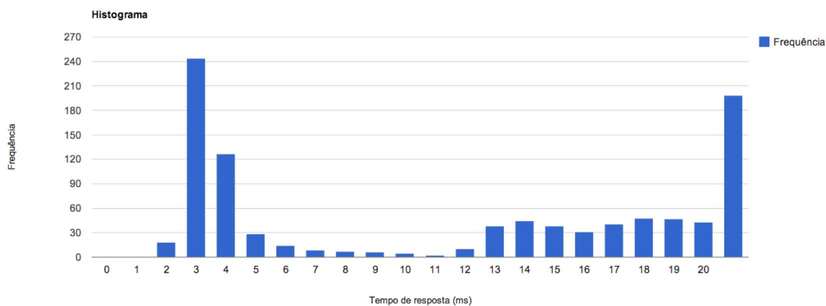


Figura 29 – Histograma com 50% de faltas

A figura 30 mostra a relação entre a quantidade percentual de faltas e a latência dos protótipos. É possível visualizar nesta imagem que com o aumento do número de faltas para 50%, a latência dos sistemas é penalizada, ficando 200% mais lentos do que no caso otimista. Algo interessante deste experimento foi visualizar que com 50% de faltas o algoritmo DIFATO tem um comportamento muito semelhante ao que ocorre quando, na ausência de faltas, se coloca 20 clientes concorrendo pelo serviço. No caso do RegPaxos isto não ocorre, uma vez que a latência na presença de faltas é muito mais afetada.

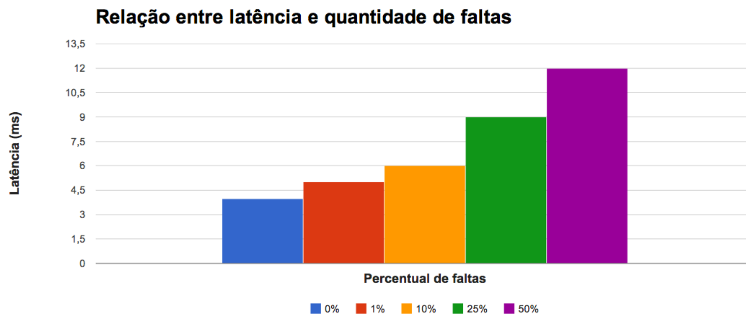


Figura 30 – Relação entre a quantidade percentual de faltas e a latência

Já na figura 31 é possível ver a relação entre a quantidade de faltas e a vazão. Fica claro que a vazão dos sistemas é muito afetada pelas faltas, entretanto, isto já era esperado, dado que esta métrica considera a quantidade de operações efetuadas por segundo, e se demora-se mais para efetuar uma tarefa, menos tarefas serão finalizadas considerando-se o mesmo intervalo. Neste gráfico é possível ver que houve uma queda percentual de mais de 66% na vazão dos sistema.

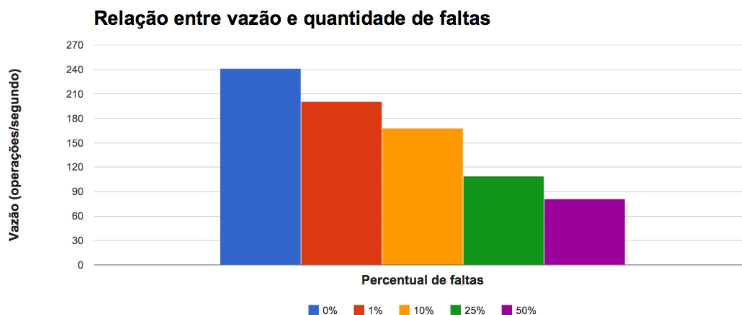


Figura 31 – Relação entre a quantidade percentual de faltas e a vazão

5.3 COMPARAÇÃO QUALITATIVA COM OUTRAS ABORDAGENS

A tabela 2 contém dados comparativos entre o RegPaxos e demais abordagens BFT da literatura. Os valores consideram execuções sem falta. Os benefícios do RegPaxos são em termos de número de réplicas, processos e máquinas físicas, além da quantidade de passos ser menor, apenas encontrada em sistemas otimistas.

	PBFT	Zyzzyva	TTCB	A2M-PBFT-EA	MinBFT	MinZyzzyva	RegPaxos
Número de réplicas	$3f+1$	$3f+1$	$2f+1$	$2f+1$	$2f+1$	$2f+1$	$2f+1$
Réplicas com o estado	$2f+1$ (YIN et al., 2003)	$2f+1$	$2f+1$	$2f+1$	$2f+1$	$2f+1$	$2f+1$
Número de processos	$2f+1$ (YIN et al., 2003)	$2f+1$	$2f+1$	$2f+1$	$2f+1$	$2f+1$	$2f+1$
Número de Máq. Físicas	$3f+1$	$3f+1$	$2f+1$	$2f+1$	$2f+1$	$2f+1$	$2f+1$
Passos de Comunicação	5	3	5	5	4	3	3
Especulativo/Otimista	não	sim	não	não	não	sim	não

Tabela 2 – Comparação entre as propriedades dos protocolos avaliados (cenário otimista)

Já a tabela 3 contém dados comparativos entre o DIFATO e demais abordagens de difusão atômica da literatura. Assim como no caso do RegPaxos, as vantagens do DIFATO vem em termos de resiliência, passos de comunicação e quantidade de mensagens trocadas.

Protocolos Avaliados	*Propriedades e características verificadas*			
	Resiliência	**Passos de comunicação**	**Mensagens trocadas**	**Tipo de faltas**
Rampart (REITER, 1994)	$3f+1$	6	$6n-6$	Bizantina
Guerraoui e Schiper (GUERRAOUI; SCHIPER, 2001)	-	5	$3n_c+2n_s-3$	parada
Correia e Veríssimo (CORREIA; NEVES; VERÍSSIMO, 2006)	$3f+1$	-	$18n^2+13n+1+16n^2f+10nf$	Bizantina
Pieri et al. (PIERI; FRAGA; LUNG, 2010)	$3f_c+1+2f_s+1$	5	$2(ns^2+3nc-ns-1)$	Bizantina
DifATo	$2f+1$	4	$n_s{}^2-n_s+n_c+1$	Bizantina

Tabela 3 – Comparação entre as propriedades de protocolos de difusão atômica (cenário otimista).

Ambas as abordagens tem vantagens e desvantagens em relacão as demais, entretanto, a real contribuição não está nos protótipos e sim no modelo de sistema que os suporta. A ideia de um modelo hibrido desenvolvido através de máquinas virtuais e compartilhamento de dados simplifica a criação de sistemas tolerantes a faltas Bizantinas, os protótipos apenas corroboram este fato.

5.4 CONSIDERAÇÕES FINAIS DO CAPÍTULO

Neste capítulo foi apresentada a implementação de dois protótipos a fim de avaliar o modelo híbrido com registradores distribuídos compartilhados. O primeiro serviço - RegPaxos - efetua as quatro operações básicas de uma calculadora, guardando o valor resultado como estado do sistema e reutilizando nas requisições subsequentes. O segundo serviço - DIFATO - provê um serviço de consenso atômico aos seus clientes. Em ambos os casos, os servidores precisam entrar em consenso em relação a ordem de cada requisição. Efetuamos testes deste protótipos em cenários diferentes que envolviam a variação da quantidade de clientes e ocorrência de faltas. O foco foi testar o comportamento do sistema diante da concorrência em seu acesso e de sua adaptabilidade na presença de faltas.

Os cenários com variação de cliente demonstraram que a quantidade de clientes influencia bastante na criação do sistema tolerante a faltas. No caso do DIFATO, este fato fica mais evidente, uma vez que, neste protocolo, o servidores precisa difundir a resposta para todos os clientes componentes do sistema. Os cenários na presença de faltas mostraram que o algoritmo, apesar de se adaptar bem, é bastante afetado. Entretanto, um cenário com 50% de faltas não é tão comum de ocorrer, uma vez que em sistemas reais costuma-se fazer o acompanhamento e qualquer anomalia pode ser detectada. O uso do modelo híbrido com DSR se comportou bem, e o uso de tecnologia de virtualização parece ser uma boa opção para o isolamento entre os subsistemas.

6 CONCLUSÃO E PERSPECTIVAS FUTURAS

Neste trabalho foi apresentada uma abordagem que faz uso de virtualização e compartilhamento de memória com finalidade de criar um modelo híbrido de tolerância a intrusões (faltas maliciosas ou bizantinas) para provedores de serviço. A abordagem com registradores compartilhados distribuídos, por ser híbrida, permite que o sistema opere com a menor resiliência prática - $2f+1$ - e facilita a criação do modelo, pois não requer mudanças nem no *software* dos sistemas operacionais, nem no *hardware* das réplicas servidoras.

A partir do modelo híbrido com registradores compartilhados distribuídos foi possível criar o protótipo RegPaxos. Este é um modelo de replicação de máquinas de estado tolerante a faltas Bizantinas que atende às propriedades de Ordem Total (*safety*) e Terminação (*liveness*). Suas principais contribuições são a sua resiliência em termos de réplicas, que é de $2f+1$ tolerando até f faltosas, e com o uso do compartilhamento de registradores, foi possível diminuir a quantidade de passos para execução das requisições dos clientes.

Um segundo modelo foi criado a partir do DSR. O modelo DIFATO de difusão atômica tolerante a faltas Bizantinas que atende às propriedades de Validade, Acordo, Integridade e Ordem Total. Este modelo atua como um serviço de consenso atômico, ordenando as mensagens dos clientes e as difundindo para todos os clientes do sistema. Sua contribuição foi a criação de um serviço de consenso atômico que necessita de apenas $N = 2f+1$ réplicas para atingir o consenso em relação a ordenação das requisições.

Com a implementação, teste e análises dos protótipos, foi possível validar não apenas o seu funcionamento, mas também viabilizar a utilização de máquinas virtuais e registradores compartilhados distribuídos para a criação de um modelo híbrido de sistemas. Esta foi também a última etapa para atingir todos os objetivos propostos na seção 1.3.

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

- Fazer um estudo comparativo de execução entre os protótipos criados e os modelos encontrados na literatura.
- Analisar a possibilidade de diminuir a quantidade de recursos físicos para $f+1$ aumentando a quantidade de máquinas virtuais em cada um deles.
- Analisar formas de melhorar o desempenho em relação a quantidade de clientes requisitando serviços

- Fazer análise do desempenho aumentando em conjunto a quantidade de clientes e a quantidade de faltas.

O desenvolvimento deste trabalho resultou nas seguintes publicações em eventos e periódicos da área:

- SILVA, M. R. X., LUNG, Lau Cheuk, MAGNABOSCO, Leandro Quibem, RECH, Luciana de Oliveira BAMcast - Byzantine Fault-Tolerant Consensus Service for Atomic Multicast in Large-scale Networks In: 18th IEEE Symposium on Computers and Communications, 2013, Split, Croatia. Proceedings of the IEEE 18th ISCC. Los Alamitos, California, USA: IEEE Computer Society, 2013. v.1. p.1 - 6 (Qualis A2)
- SILVA, M. R. X. ; LUNG, Lau Cheuk ; LUIZ, Aldelir Fernando ; MAGNABOSCO, Leandro Quibem . RegPaxos: Tolerância a Faltas Bizantinas Usando Registradores Distribuído e Compartilhado. In: XXXI Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos, 2013, Brasilia - DF. Anais do SBRC'13. Porto Alegre, RS: SBC, 2013. v. 1. p. 1-14. (Qualis B2)
- SILVA, M. R. X. ; LUNG, Lau Cheuk ; MAGNABOSCO, Leandro Quibem . Vbam - Byzantine Atomic Multicast in LAN Based on Virtualization Technology. In: 8th International Conference on Dependability and Complex Systems, 2013, Brunow, Polônia. Monographs of System Dependability series - 8th DepCoS-RELCOMEX. Wroclaw, Polônia: Wroclaw University of Technology, 2013. v. 1. p. 1-10. (Qualis B3)
  - SILVA, M. R. X. et al. Vbam - byzantine atomic multicast in lan based on virtualization technology. In: New Results in Dependability and Computer Systems. [S.l.]: Springer, 2013. p. 365-374. (journal)
- SILVA, M. R. X. ; LUNG, Lau Cheuk ; LUIZ, Aldelir Fernando ; MAGNABOSCO, Leandro Quibem . DIfaTo Difusão Atômica Tolerante a Faltas Bizantinas Baseada em Tecnologia de Virtualização. In: XIV SBC Workshop de Teste e Tolerância a Falhas, 2013, Brasília, DF. Anais do WTF'13. Porto Alegre, RS: SBC, 2013. v. 1. p. 1-14. (Qualis B4)

REFERÊNCIAS BIBLIOGRÁFICAS

AIYER, A. et al. Bar fault tolerance for cooperative services. *ACM SIGOPS Operating Systems Review*, v. 39, n. 5, p. 45–58, 2005.

AMIR, Y. et al. Scaling byzantine fault-tolerant replication to wide area networks. In: IEEE. *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*. [S.l.], 2006. p. 105–114.

ANDERSON, J. *Automata theory with modern applications*. [S.l.]: Cambridge University Press, 2006.

ATTIYA, H.; WELCH, J. *Distributed computing: fundamentals, simulations, and advanced topics*. [S.l.]: Wiley-Interscience, 2004.

BACKE, H. a zepto-second atomic clock for nuclear contact time measurements at superheavy collision systems. In: *Exploring Fundamental Issues in Nuclear Physics: Nuclear Clusters–Superheavy, Superneutronic, Superstrange, of Anti-Matter*. [S.l.: s.n.], 2012. v. 1, p. 172–181.

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

BARHAM, P. et al. Xen and the art of virtualization. In: ACM. *ACM SIGOPS Operating Systems Review*. [S.l.], 2003. v. 37, n. 5, p. 164–177.

BASU, A.; CHARRON-BOST, B.; TOUEG, S. Simulating reliable links with unreliable links in the presence of process crashes. *Distributed algorithms*, Springer, p. 105–122, 1996.

BESSANI, A. et al. Depspace: a byzantine fault-tolerant coordination service. In: ACM. *ACM SIGOPS Operating Systems Review*. [S.l.], 2008. v. 42, p. 163 –176.

BESSANI, A. et al. Sharing memory between byzantine processes using policy-enforced tuple spaces. *Parallel and Distributed Systems, IEEE Transactions on*, IEEE, v. 20, n. 3, p. 419–432, 2009.

BESSANI, A. N.; FRAGA, J. da S.; LUNG, L. C. Bts: A byzantine fault-tolerant tuple space. In: ACM. *Proceedings of the 2006 ACM symposium on Applied computing*. [S.l.], 2006. p. 429–433.

CASTRO, M.; LISKOV, B. Practical byzantine fault tolerance. *Operating Systems Review*, ACM ASSOCIATION FOR COMPUTING MACHINERY, v. 33, p. 173–186, 1998.

CASTRO, M.; LISKOV, B. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, ACM, v. 20, n. 4, p. 398–461, 2002.

CHANDRA, T.; TOUEG, S. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, ACM, v. 43, n. 2, p. 225–267, 1996.

CHANDY, K.; LAMPORT, L. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, ACM, v. 3, n. 1, p. 63–75, 1985.

CHARRON-BOST, B.; DÉFAGO, X.; SCHIPER, A. Broadcasting messages in fault-tolerant distributed systems: the benefit of handling input-triggered and output-triggered suspicions differently. In: IEEE. *Reliable Distributed Systems, 2002. Proceedings. 21st IEEE Symposium on*. [S.l.], 2002. p. 244–249.

CHUN, B.; MANIATIS, P.; SHENKER, S. Diverse replication for single-machine byzantine-fault tolerance. In: *USENIX Annual Technical Conference*. [S.l.: s.n.], 2008. p. 287–292.

CHUN, B. et al. Attested append-only memory: Making adversaries stick to their word. In: ACM. *ACM SIGOPS Operating Systems Review*. [S.l.], 2007. v. 41, p. 189–204.

CLEMENT, A. et al. Upright cluster services. In: ACM. *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. [S.l.], 2009. p. 277–290.

CLEMENT, A. et al. Making byzantine fault tolerant systems tolerate byzantine faults. In: USENIX ASSOCIATION. *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*. [S.l.], 2009. p. 153–168.

CORREIA, M. et al. Efficient byzantine-resilient reliable multicast on a hybrid failure model. In: IEEE. *Reliable Distributed Systems, 2002. Proceedings. 21st IEEE Symposium on*. [S.l.], 2002. p. 2–11.

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

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

CORREIA, M.; NEVES, N.; VERISSIMO, P. Bft-to: Intrusion tolerance with less replicas. *The Computer Journal*, Br Computer Soc, 2012.

CORREIA, M.; NEVES, N. F.; VERÍSSIMO, P. From consensus to atomic broadcast: Time-free byzantine-resistant protocols without signatures. *The Computer Journal*, Br Computer Soc, v. 49, n. 1, p. 82–96, 2006.

CORREIA, M.; VERISSIMO, P.; NEVES, N. The design of a cots real-time distributed security kernel. *Dependable Computing EDCC-4*, Springer, p. 634–638, 2002.

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

CRISTIAN, F. Probabilistic clock synchronization. *Distributed computing*, Springer, v. 3, n. 3, p. 146–158, 1989.

CRISTIAN, F. et al. Atomic broadcast: From simple message diffusion to byzantine agreement. *Information and Computation*, Citeseer, v. 118, n. 1, p. 158, 1995.

CRISTIAN, F.; FETZER, C. The timed asynchronous distributed system model. *Parallel and Distributed Systems, IEEE Transactions on*, IEEE, v. 10, n. 6, p. 642–657, 1999.

CULLY, B. et al. Remus: High availability via asynchronous virtual machine replication. In: *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*. [S.l.: s.n.], 2008. p. 161–174.

DÉFAGO, X.; SCHIPER, A.; URBÁN, P. Total order broadcast and multicast algorithms: Taxonomy and survey. ???????????????????, 2003.

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

DOLEV, D.; STRONG, H. Authenticated algorithms for byzantine agreement. *SIAM J. Comput.*, v. 12, n. 4, p. 656–666, 1983.

EKWALL, R.; SCHIPER, A.; URBÁN, P. Token-based atomic broadcast using unreliable failure detectors. In: IEEE. *Reliable Distributed Systems, 2004. Proceedings of the 23rd IEEE International Symposium on*. [S.l.], 2004. p. 52–65.

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

FISCHER, M.; LYNCH, N.; PATERSON, M. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, ACM, v. 32, n. 2, p. 374–382, 1985.

GARCIA, M. et al. Os diversity for intrusion tolerance: Myth or reality? In: IEEE. *Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*. [S.l.], 2011. p. 383–394.

GARCIA, R.; RODRIGUES, R.; PREGUIÇA, N. Efficient middleware for byzantine fault tolerant database replication. In: ACM. *Proceedings of the sixth conference on Computer systems*. [S.l.], 2011. p. 107–122.

GARFINKEL, T. et al. Terra: A virtual machine-based platform for trusted computing. In: ACM. *ACM SIGOPS Operating Systems Review*. [S.l.], 2003. v. 37, n. 5, p. 193–206.

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

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

GUERRAOUI, R.; RODRIGUES, L. *Introduction to reliable distributed programming*. [S.l.]: Springer-Verlag New York Inc, 2006.

GUERRAOUI, R.; RODRIGUES, L. *Reliable Distributed Programming*. [S.l.]: Springer Verlag, Berlin, 2006.

GUERRAOUI, R.; SCHIPER, A. The generic consensus service. *Software Engineering, IEEE Transactions on*, IEEE, v. 27, n. 1, p. 29–41, 2001.

GUSELLA, R.; ZATTI, S. The accuracy of the clock synchronization achieved by tempo in berkeley unix 4.3 bsd. *Software Engineering, IEEE Transactions on*, IEEE, v. 15, n. 7, p. 847–853, 1989.

HADZILACOS, V.; TOUEG, S. A modular approach to the specification and implementation of fault-tolerant broadcasts. *Dep. of Computer Science, Cornell Univ., New York, USA, Tech. Rep*, p. 94–1425, 1994.

JÚNIOR, V. et al. Smit: Uma arquitetura tolerante a intrusoes baseada em virtualizaç ao.

KAPITZA, R. et al. Cheapbft: resource-efficient byzantine fault tolerance. In: ACM. *Proceedings of the 7th ACM european conference on Computer Systems*. [S.l.], 2012. p. 295–308.

KEAHEY, K.; DOERING, K.; FOSTER, I. From sandbox to playground: Dynamic virtual environments in the grid. In: IEEE. *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*. [S.l.], 2004. p. 34–42.

KEMME, B. et al. Using optimistic atomic broadcast in transaction processing systems. *Knowledge and Data Engineering, IEEE Transactions on*, IEEE, v. 15, n. 4, p. 1018–1032, 2003.

KERNEL-BASED Virtual Machine. dez. 2012. <http://www.linux-kvm.org/>.

KING, S.; DUNLAP, G.; CHEN, P. Operating system support for virtual machines. In: *Proceedings of the 2003 Annual USENIX Technical Conference*. [S.l.: s.n.], 2003. p. 71–84.

KOTLA, R. et al. Zyzzyva: speculative byzantine fault tolerance. *Communications of the ACM*, ACM, v. 51, n. 11, p. 86–95, 2008.

LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, ACM, v. 21, n. 7, p. 558–565, 1978.

LAMPORT, L.; MELLIAR-SMITH, P. Synchronizing clocks in the presence of faults. *Journal of the ACM (JACM)*, ACM, v. 32, n. 1, p. 52–78, 1985.

LAMPORT, L.; SHOSTAK, R.; PEASE, M. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, ACM, v. 4, n. 3, p. 382–401, 1982.

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

LEVIN, D. et al. Trinc: Small trusted hardware for large distributed systems. In: *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. [S.l.: s.n.], 2009.

LI, Y.; LI, W.; JIANG, C. A survey of virtual machine system: Current technology and future trends. In: IEEE. *Electronic Commerce and Security (ISECS), 2010 Third International Symposium on*. [S.l.], 2010. p. 332–336.

LINDHOLM, T.; YELLIN, F. *Java virtual machine specification*. [S.l.]: Addison-Wesley Longman Publishing Co., Inc., 1999.

LUIZ, A. et al. Repeats-uma arquitetura para replicaç ao tolerante a faltas bizantinas baseada em espaço de tuplas. *Anais do XXVI Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuıdos. SBC*, 2008.

LYNCH, N. A. *Distributed algorithms*. [S.l.]: Morgan Kaufmann, 1996.

MENEZES, A.; OORSCHOT, P. V.; VANSTONE, S. *Handbook of applied cryptography*. [S.l.]: CRC, 1996.

MILLS, D. Improved algorithms for synchronizing computer network clocks. *Networking, IEEE/ACM Transactions on*, IEEE, v. 3, n. 3, p. 245–254, 1995.

MURRAY, D. G.; MILOS, G.; HAND, S. Improving xen security through disaggregation. In: ACM. *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. [S.l.], 2008. p. 151–160.

OBELHEIRO, R.; BESSANI, A.; LUNG, L. Analisando a viabilidade da implementaçao prática de sistemas tolerantes a intrusoes. *Anais do V Simpósio Brasileiro em Segurança da Informaç ao e de Sistemas Computacionais-SBSeg 2005*, 2005.

OBELHEIRO, R. et al. How practical are intrusion-tolerant distributed systems? Department of Informatics, University of Lisbon, 2006.

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

PEASE, M.; SHOSTAK, R.; LAMPORT, L. Reaching agreement in the presence of faults. *Journal of the ACM*, v. 27, n. 2, p. 228–234, 1980.

PIERI, G.; FRAGA, J. da S.; LUNG, L. C. Consensus service to solve agreement problems. In: IEEE. *Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on*. [S.l.], 2010. p. 267–274.

REISER, H.; KAPITZA, R. Vm-fit: supporting intrusion tolerance with virtualisation technology. In: *Proceedings of the Workshop on Recent Advances on Intrusion-Tolerant Systems*. [S.l.: s.n.], 2007.

REISER, H.; KAPITZA, R. Fault and intrusion tolerance on the basis of virtual machines. *Tagungsband des*, v. 1, p. 11–12, 2008.

REITER, M. The rampart toolkit for building high-integrity services. *Theory and Practice in Distributed Systems*, Springer, p. 99–110, 1995.

REITER, M. et al. The ? key management service. In: ACM. *Proceedings of the 3rd ACM conference on Computer and communications security*. [S.l.], 1996. p. 38–47.

REITER, M. K. Secure agreement protocols: Reliable and atomic group multicast in rampart. In: ACM. *Proceedings of the 2nd ACM Conference on Computer and Communications Security*. [S.l.], 1994. p. 68–80.

RODRIGUES, L.; VERÍSSIMO, P.; CASIMIRO, A. Using atomic broadcast to implement a posteriori agreement for clock synchronization. In: IEEE. *Proceedings of the 12th Symposium on Reliable Distributed Systems - SRDS'93*. [S.l.], 1993. p. 115–124.

RODRIGUES, R.; CASTRO, M.; LISKOV, B. Base: Using abstraction to improve fault tolerance. *ACM SIGOPS Operating Systems Review*, ACM, v. 35, n. 5, p. 15–28, 2001.

ROSENBLUM, M. The reincarnation of virtual machines. *Queue*, ACM, v. 2, n. 5, p. 34, 2004.

ROSENBLUM, M.; GARFINKEL, T. Virtual machine monitors: Current technology and future trends. *Computer*, IEEE, v. 38, n. 5, p. 39–47, 2005.

RUTKOWSKA, J. Introducing blue pill. *The official blog of the invisiblethings. org*, v. 22, 2006.

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

SCHNEIDER, F. Synchronization in distributed programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, ACM, v. 4, n. 2, p. 125–148, 1982.

SCHNEIDER, F. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, ACM, v. 22, n. 4, p. 299–319, 1990.

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.

SMITH, J.; NAIR, R. The architecture of virtual machines. *Computer*, IEEE, v. 38, n. 5, p. 32–38, 2005.

SOUSA, P.; NEVES, N.; VERISSIMO, P. How resilient are distributed f fault/intrusion-tolerant systems? In: IEEE. *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*. [S.l.], 2005. p. 98–107.

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

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

TANENBAUM, A. *Modern operating systems*. [S.l.]: Prentice Hall New Jersey, 1992.

TANENBAUM, A. S.; STEEN, M. V. *Distributed systems*. [S.l.]: Prentice Hall, 2002.

VERÍSSIMO, P. Travelling through wormholes: a new look at distributed systems models. *ACM SIGACT News*, ACM, v. 37, n. 1, p. 66–81, 2006.

VERÍSSIMO, P.; CASIMIRO, A. The timely computing base model and architecture. *Computers, IEEE Transactions on*, IEEE, v. 51, n. 8, p. 916–930, 2002.

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

VERONESE, G. et al. Efficient byzantine fault tolerance. *Computers, IEEE Transactions on*, IEEE, p. 1–1, 2011.

WANG, Z.; JIANG, X. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In: IEEE. *Security and Privacy (SP), 2010 IEEE Symposium on*. [S.l.], 2010. p. 380–395.

WANGHAM, M. S. et al. Integrating ssl to the jacoweb security framework: Project and implementation. In: *Integrated Network Management'01*. [S.l.: s.n.], 2001. p. 779–792.

WOOD, T. et al. *ZZ: Cheap practical BFT using virtualization*. [S.l.], 2008.

XEN Project. dez. 2012. <http://www.xenproject.org/>.

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

ZHOU, L.; SCHNEIDER, F.; RENESSE, R. V. Coca: A secure distributed online certification authority. *ACM Transactions on Computer Systems (TOCS)*, ACM, v. 20, n. 4, p. 329–368, 2002.