UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

Marcelo Ribeiro Xavier da Silva

REGPAXOS: UMA ARQUITETURA TOLERANTE A INTRUSÕES MENOS COMPLEXA

Florianópolis

2012

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

1.1 CONTEXTUALIZAÇÃO E DEFINIÇÃO DO PROBLEMA

É impossível imaginar como seria nosso cotidiano sem a participação 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 (VERÍSSIMO; NEVES; CORREIA, 2003) 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 (VERONESE 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, 1990). O número adicional de réplicas para o primeiro caso se faz necessário para tolerar comportamentos maliciosos e/ou arbitrários (CORREIA; NEVES; VERISSIMO, 2012). 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 ABORDAGEM PROPOSTA

O trabalho aqui presente traz uma solução para tolerância a faltas bizantinas baseada na técnica de replicação de máquina de estados, sobre um modelo híbrido de sistema que utiliza virtualização e uma abstração de memória compartilhada emulada para criar um componente inviolável e capaz de suportar o consenso bizantino.

Assim como foi proposto em (CORREIA; NEVES; VERISSIMO, 2004) nós propomos uma abordagem híbrida em que a ordenação de requisições de cliente seja feita utilizando-se um componente inviolável, mantendo o sistema com a resiliência de $2f + 1$ réplicas para tolerar f faltosas. Porém, diferentemente do trabalho acima citado, na abordagem que propomos não existe acoplamento de nenhum trecho do sistema ao *kernel* do sistema operacional, nem necessidade de criação ou modificação do *hardware* da máquina hospedeira. A inviolabilidade do componente de ordenação é obtida através de isolamento por máquina virtual, permitido pelo *hypervisor* presente em tecnologias de virtualização, e o meio para suportar o consenso é obtido através do Registrador Compartilhado Distribuído, uma abstração de memória compartilhada emulada, que é implantado na máquina hospedeira e também isolado pela virtualização.

O modelo prevê que o serviço disponibilizado aos clientes esteja apenas no *guest* da máquina virtual e o Registrador Compartilhado Distribuído na máquina hospedeira. Desta maneira é possível criar duas redes totalmente separadas, a rede de *payload*, onde ocorrem as comunicações cliente-servidor, e a rede controlada, onde ocorrem as comunicações servidor-servidor.

1.3 OBJETIVOS

O principal objetivo do presente trabalho é investigar um modelo híbrido tolerante a faltas bizantinas baseado em replicação de máquina de estados. Este modelo visa trazer uma solução simples para a criação de um componente inviolável. Além disso, com este modelo, queremos criar o primeiro algoritmo não especulativo a equiparar-se com algoritmos especulativos em relação à latência. Através do modelo, o presente trabalho pretende apresentar a especificação de um protocolo, aqui denominado de RegPaxos, para execução de serviços replicados. Finalmente, a partir do protocolo RegPaxos pretende-se desenvolver um servidor *web* tolerante a faltas bizantinas.

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 tolerantes a intrusão.
2. Especificação de um protocolo de consenso baseado na arquitetura descrita acima, bem como a elaboração de suas provas de correção.
3. Implementação de um protótipo de servidor *web* baseado na arquitetura e no protocolo propostos.
4. Testes de execução e avaliação do desempenho obtido com o protótipo.
5. Especificação e implementação de protocolos que, juntamente ao protocolo de consenso, garantam a execução segura de serviços no modelo proposto.

1.4 ORGANIZAÇÃO DO TEXTO

O restante do documento está organizado conforme segue. O capítulo 2 apresenta os principais conceitos da literatura que servem de contextualização e embasamento para a proposta. O capítulo 3 apresenta os trabalhos que se relacionam diretamente com o assunto da proposta.

2 CONCEITOS BÁSICOS EM COMPUTAÇÃO DISTRIBUÍDA

2.1 AMBIENTE DE COMPUTAÇÃO DISTRIBUÍDA

2.1.1 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 os 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 tornas 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.

2.1.1.1 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, em outras palavras, 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 2.1.1.5 é apresentado o problema de sincronismo.

2.1.1.2 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 bizantina) - 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 2.1.1.6 é discutido mais profundamente os tipos de falhas.

2.1.1.3 Modelo de segurança

O modelo de segurança, segundo (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 2.1.1.7), 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. Já que estas decisões podem ser importantes para a segurança do sistema, mas podem ser custosas para o resto do modelo, tornando-se impraticáveis.

2.1.1.4 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).

2.1.1.5 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 (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, determinando quando podemos ou não 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 2.1.2.2) 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 eventualmente, 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.

2.1.1.6 Tipos de falhas

Uma falha de sistema ocorre quando o serviço prestado se desvia de cumprir a função 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 (GUERRAOUI; RODRIGUES, 2006a) define 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 ainda afirma 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 sin-

cronismo. Este tipo de falha é irrelevante para sistemas assíncronos.

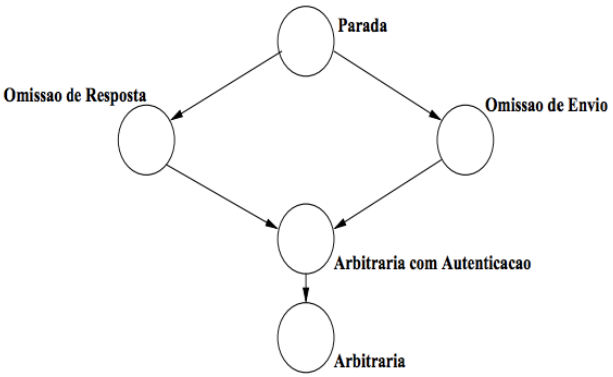

Figura 1 – Grafo de tipos de falhas de acordo com a severidade.

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.

2.1.1.7 Canais de comunicação

A comunicação em sistemas distribuídos ocorre, de maneira geral, através da troca de mensagens entre processos. Para tanto, é necessário que aja 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).

2.1.2 Problemas de acordo

2.1.2.1 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:

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 ou nenhum processo do grupo esta correto;
- 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.

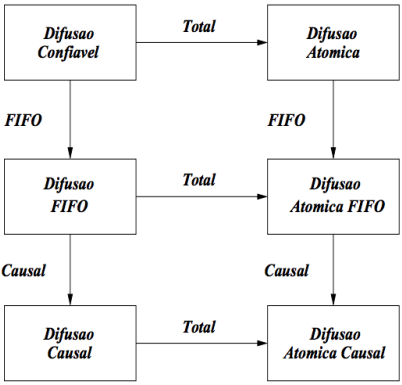

Figura 2 – Relacionamento entre problemas de difusão.

2.1.2.2 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 decido.

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.

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).

2.1.2.3 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 3 e 4 mostram a impossibilidade de acordo bizantino descrita no artigo.

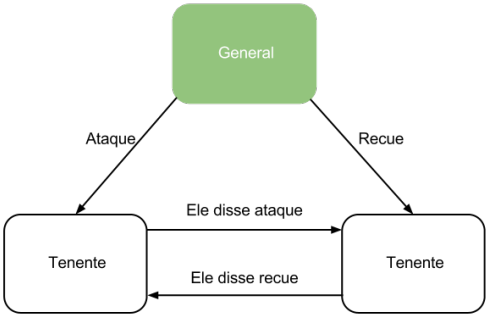

Figura 3 – General traidor.

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 3) o general

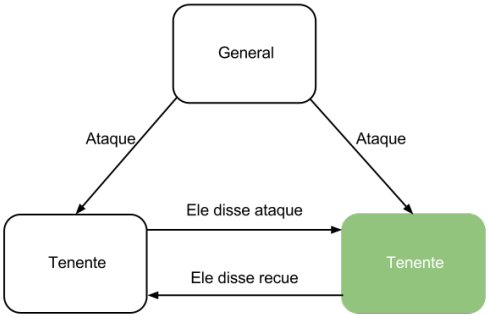

Figura 4 – Tenente traidor.

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 4) 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 5 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 pelos 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.

2.1.3 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 pro-

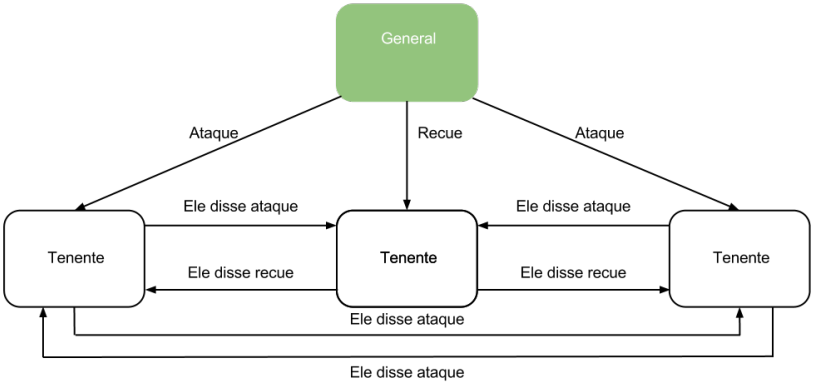

Figura 5 – Possibilidade de acordo bizantino com $3f + 1$ participantes.

cessam as requisições como suas entradas e tem o resultado deste processamento como as suas respostas ou saídas (SCHNEIDER, 1990).

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, 1990), 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, 1990) 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 2.1.2) 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).

2.1.4 Memória compartilhada emulada

Guerraoui (GUERRAOUI; RODRIGUES, 2006b) define 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 o autor, 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 2.1.5), 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 comparti-

lhado 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. Vivacidade (*liveness*) - toda operação eventualmente termina.

2. Segurança (*safety*) - toda leitura retorna o último valor escrito.

Mesmo que não seja 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, isto garante que o registro a ser lido será o último registro já escrito.

2.1.5 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-funcionais, 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) os autores afirmam que estes modelos permitem-nos tirar o melhor de ambas às dimensões. Eles mostram 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. Modelo 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 àqueles 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 difícil 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.

2.2 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 (POPEK; GOLDBERG, 1974) define máquina virtual como uma cópia isolada e eficiente de uma máquina real e, Parmelee (PARMELEE et al., 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 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 com que o mercado de virtualização de sistemas voltasse ser bastante atrativo (ROSENBLUM, 2004). As aplicações de virtualização tornaram-se mais abrangentes do que apenas otimizar os recursos oferecidos, vindo a ser aplicados 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; 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 o 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; 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 é ressaltada em *garfinkel2003terra*. 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).

3 TRABALHOS CORRELATOS

Desde a formulação do Problema dos Generais Bizantinos por Lamport (LAMPORT; SHOSTAK; PEASE, 1982) (vide 2.1.2), 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 *throughput* e 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, 1990). 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 2.1.5). 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.

Existe ainda uma terceira seção que relaciona este trabalho com a literatura de criação de servidor *web* tolerante a intrusões.

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 a vivacidade (*liveness*) do 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 à mensagem e, ao aceitar, 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 sendo 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 vivacidade, 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.

Conclusão

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 superadas 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 os 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)footnotemark[1] o cliente verifica se existem $3f + 1$ respostas iguais entre si e, em caso positivo, aceita a resposta.

A resposta 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 determi-

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

nado pelas respostas. Se o cliente receber entre $2f+1$ e $3f$ respostas iguais entre si então ele executa o passo (4b) ao invés de (4a), onde são reenviados 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, eventualmente, 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 através de protocolos BFT já existentes. O *overhead* de *throughput* e 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 por 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 propõe 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 à 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 checkpoints necessários e se houver requisições após o último *checkpoint*, estas são recuperadas diretamente do espaço.

Conclusão

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 2.1.5).

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*), comprometidas (*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.

Conclusão

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 sua vivacidade 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. Além disso, são necessário mais passos para a comunicação do que no RegPaxos, já que, por implementar o PBFT, precisa de cinco passos.

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 comprometido (*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 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 reposta *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 de 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 bom 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 à $2f+1$ 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 à convergirem numa única ordem total de requisições, possivelmente tendo que fazer o *rollback* de algumas execuções. Os cliente só confiam nas respostas que estiverem consistentes com esta ordem total. O MinZyzzyva usa o serviço USIG para constranger o comportamento da réplica primária, permitindo a redução do número de réplicas de $3f + 1$ para $2f + 1$, preservando as propriedades de segurança (*safety*) e vivacidade (*liveness*). As réplicas *backup* só aceitam da 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).

Conclusão

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 à cargo do domínio NV, que tem total acesso a abstração de rede e, por sua vez, à todas as requisições que vem 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 à 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. 3A transferência de estado pode ser feita através de um simples remapeamento de blocos de memória, caso a réplica recuperada apresente

comportamento correto.

Conclusão

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.

Conclusão

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 casa 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 comprometidas 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 recuperar 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.

Conclusão

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 vivacidade 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.

Conclusão

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 2.1.2) 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 implemen-

[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.

tado 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 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.

Conclusão

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. No RegPaxos o isolamento é feito através da máquina virtual, portanto o atacante terá que invadir primeiramente a máquina virtual e, só então, achar vulnerabilidades no *hypervisor* para acessar o sistema hospedeiro.

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.

Tabela 1 – Comparação entre as propriedades dos protocolos avaliados

	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

3.3 SERVIDOR *WEB* TOLERANTE A INTRUSÕES

O estudo de técnicas de tolerância a intrusão é sempre embasado por necessidades reais de melhorias nas técnicas atuais. Não basta apenas comprovar em *benchmarks* a sua funcionalidade, pois em casos reais o resultado pode ser diferente. Por isso, ao se estudar técnicas BFT, é comum a utilização de protótipos para validá-las. Dentre estes protótipos, encontram-se: Sistemas de arquivos em rede, *backup* cooperativo, serviços de coordenação, autoridades certificadoras, banco de dados, sistemas de gerenciamento de chaves, etc (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). Um protótipo interessante e que pode ser colocado em prática em problemas reais é a criação de um servidor *web* BFT. Neste trabalho seguiremos esta linha e nesta seção apresentamos a literatura relacionada.

Em (FERRAZ et al., 2004) é apresentada DISTRACT (*Deterministic IntruSion ToleRance ArChiTecture*), uma arquitetura de suporte a serviços tolerantes a intrusão baseada em replicação de máquina de estados. Este artigo reporta a primeira implementação completa de um serviço tolerante a intrusão baseado em *wormhole*, mais especificamente, baseada no TTCB (*Trusted Timely Computing Base*) (CORREIA; VERISSIMO; NEVES, 2002).

A arquitetura do sistema é composta pelos clientes, pelos *proxies* e pelos servidores. No modelo do servidor *web*, os clientes são processos que se comunicam através do protocolo HTTP (FIELDING et al., 1999), por exemplo, *web browser*. Os *proxies* são servidores intermediários que servem para esconder a replicação dos clientes. Os servidores são de fato aqueles que possuem o serviço replicado. Tanto os *proxies* quanto os servidores são munidos do TTCB. Todo o sistema está sujeito a faltas arbitrárias, com exceção dos *proxies*, que estão sujeito apenas a paradas (*crash*). Cada servidor é dividido em duas partes, *skeleton*, que é o serviço tolerante a intrusão e o servidor original do serviço, que é quem processa a requisição do cliente gerando uma resposta. No artigo os autores utilizaram um conjunto de protocolos em que se fazem necessárias $3f+1$ réplicas com f faltosas.

De maneira simples, o processo ocorre com um cliente enviando uma requisição para um *proxy*. Este *proxy* envia a requisição para cada *skeleton* de cada servidor. Cada *skeleton*, por sua vez, chama o servidor original que processa e envia a resposta para o *proxy*. Quando o *proxy* tem um certo número de respostas iguais entre si, ele responde para o cliente.

Os *proxies* e servidores se comunicam através de canais seguros que garantem que as mensagens não serão corrompidas na rede e que serão eventualmente recebidas. Quando um *proxy* recebe uma requisição *R*, ele usa o

TTCB para tirar um uma estampilha de tempo t e, em seguida, envia a mensagem com a estampilha de tempo para todos os servidores. A ordenação das mensagens é feita através da estampilha de tempo, porém, os servidores também precisam concordar com o conjunto de mensagens a ser ordenado e entregue. O servidor possui dois estados: Normal e acordo. Se o servidor está ocioso, então ele está em modo normal. O segundo estado corresponde ao acordo sobre entrega das mensagens.

Um servidor usa dois critérios para decidir se participará do protocolo acordo: quando ele recebe uma ou mais mensagens e já se passou um tempo T desde o último acordo.

Quando o protocolo de acordo termina, cada servidor executa as requisições de acordo com a ordem de suas estampilhas de tempo. Assim que todas as requisições são executadas por todos os servidores, cada um deles envia uma resposta para o *proxy*. Ao receber $f+1$ respostas idênticas entre si e de diferentes servidores, o *proxy* encaminha a resposta para o cliente.

DISTRACT usa *Domain Name System* (DNS) (MOCKAPETRIS, 1987) para devolver um *proxy* para os clientes. Isto é, o serviço de DNS permite que mais de um endereço IP esteja associado a um domínio, estes IPs são devolvidos de acordo com algumas políticas definidas no serviço. Se um cliente recebe um IP de um servidor inacessível por qualquer razão, basta que seja reenviada a requisição (*refresh*) para que o DNS devolva um outro IP.

Um problema que os autores relatam é que pacotes HTTP gerados para uma mesma requisição podem ter valores diferentes em seu cabeçalho, isto afetaria o determinismo da máquina de estados, portanto, os autores descartam as três primeiras linhas do cabeçalho HTTP (pois são as que variam) no momento de comparar as $f+1$ respostas.

Conclusão

A solução apresentada mostra vários benefícios interessantes, pois cria um sistema de servidores *web* tolerantes a faltas bizantinas sem que haja necessidade de se modificar os clientes ou os servidores originais. A diversidade de servidores é suportada de maneira transparente. Por ser o primeiro, e pela literatura o único, trabalho a criar um servidor *web* tolerante a faltas, seu destaque é inegável. Entretanto, é possível melhorar sua resiliência utilizando modelos híbridos e *wormholes* mais recentes. Além disso seria interessante eliminar a necessidade do uso de intermediários (*proxies*) na comunicação. Ou seja, é possível estudar outras abordagens, como, por exemplo, criação de *add-ons* para *browsers* como o Mozilla Firefox[1] fazendo com que o mesmo espere $f+1$ respostas para um requisição ao invés de uma.

[1] http://www.mozilla.org/en-US/firefox/new/

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.

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.; 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.

FERRAZ, R. et al. An intrusiontolerant web server based on the distract architecture. In: CITESEER. *In Proceedings of the Workshop on Dependable Distributed Data Management*. [S.l.], 2004.

FIELDING, R. et al. *Hypertext transfer protocol–HTTP/1.1*. [S.l.]: RFC 2616, June, 1999.

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.; 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.

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.

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.

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

MOCKAPETRIS, P. Domain names-concepts and facilities. 1987.

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.

PARMELEE, R. et al. Virtual storage and virtual machine concepts. *IBM Systems Journal*, IBM, v. 11, n. 2, p. 99–130, 1972.

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.

POPEK, G.; GOLDBERG, R. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, ACM, v. 17, n. 7, p. 412–421, 1974.

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.

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.

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.

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.

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

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.

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

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.