UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE ENGENHARIA DE AUTOMAÇÃO E SISTEMAS

Eduardo Adilio Pelinson Alchieri

PROTOCOLOS TOLERANTES A FALTAS BIZANTINAS PARA SISTEMAS DISTRIBUÍDOS DINÂMICOS

Florianópolis

2011

Eduardo Adilio Pelinson Alchieri

PROTOCOLOS TOLERANTES A FALTAS BIZANTINAS PARA SISTEMAS DISTRIBUÍDOS DINÂMICOS

Tese submetida ao Programa de Pós-Graduação em Engenharia de Automação e Sistemas como parte dos requisitos para a obtenção do Grau de Doutor em Engenharia de Automação e Sistemas.
Orientador: Prof. Dr. Joni da Silva Fraga
Coorientador: Prof. Dr. Alysson Neves Bessani

Florianópolis

2011

Eduardo Adilio Pelinson Alchieri

PROTOCOLOS TOLERANTES A FALTAS BIZANTINAS PARA SISTEMAS DISTRIBUÍDOS DINÂMICOS

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

Florianópolis, 09 de setembro 2011.

Prof. Chefe, Dr. José Eduardo Ribeiro Cury
Coordenador do Curso

Prof. Dr. Joni da Silva Fraga
Orientador

Prof. Dr. Alysson Neves Bessani
Coorientador

Banca Examinadora:

Prof. Dr. Joni da Silva Fraga
Presidente

Prof. Dr. Elias Procópio Duarte Júnior

Prof. Dra. Fabíola Gonçalves Greve

Prof. Dr. Mário Antônio Ribeiro Dantas

Prof. Dr. Fábio Favarim

A toda minha família.

AGRADECIMENTOS

Primeiramente, gostaria de agradecer a Deus por tornar tudo isso possível. Agradeço também a minha família, principalmente aos meus pais, Luiz e Aurora, pelo incentivo e apoio na realização deste trabalho, além do exemplo de vida que representam para mim. Tudo que sou e tenho devo a eles. Também gostaria de agradecer a minha companheira, Patricia, por ter surgido em minha vida e me apoiado na reta final do doutorado.

Agradeço aos meus orientadores, Prof. Dr. Joni da Silva Fraga e Prof. Dr. Alysson Neves Bessani, por todas as conversas e pela atenção a mim despendida durate o doutorado. Foi grande o aprendizado adquirido com eles durante este período de convivênvia.

Também quero agradecer aos colegas do DAS, pela grata experiência de tê-los como colegas de trabalho, onde muitas vezes me ajudaram com sugestões e opiniões. Através deles também tive a oportunidade de aprimorar o meu aprendizado.

Não poderia deixar de agradecer a todos os meus amigos, que de uma ou outra maneira contribuiram para a realização deste trabalho. Sem eles seria muito difícil suportar a distância da família. Por fim, também gostaria de enviar meus agradecimentos ao CNPQ e à CAPES pelo apoio financeiro durante a realização do doutorado.

Uma coisa é uma coisa. E outra coisa é outra coisa.

Desconhecido

RESUMO

As novas tecnologias de comunicação e a maior disponibilidade de recursos em redes de comunicação vêm provocando profundas mudanças na forma de se projetar aplicações distribuídas. Estas mudanças possibilitaram o surgimento dos sistemas distribuídos dinâmicos, que se caracterizam por serem sistemas onde os componentes podem entrar e sair do mesmo em qualquer momento. Sendo assim, os protocolos desenvolvidos para estes sistemas devem detectar e tratar mudanças que ocorrem na composição da aplicação distribuída, permitindo sua reconfiguração em tempo de execução. Desta forma, os participantes destas aplicações são caracterizados principalmente pela heterogeneidade e não confiabilidade.

No desenvolvimento de aplicações distribuídas seguras e confiáveis, vários problemas são identificados como peças fundamentais por formarem a base para a maioria das soluções empregadas nestas aplicações. Neste sentido, surge a necessidade do desenvolvimento de protocolos que resolvem estes problemas em sistemas distribuídos dinâmicos. Como os participantes destas computações não são confiáveis, torna-se essencial que estes protocolos suportem os atributos de segurança de funcionamento.

Esta tese apresenta estudos e protocolos para a solução dos principais destes problemas fundamentais em sistemas distribuídos dinâmicos, os quais são: o problema do consenso, sistemas de quóruns e replicação Máquina de Estados. O problema do consenso é estudado em redes desconhecidas, onde são definidas as condições necessárias e suficientes para resolver o consenso. Estas condições especificam o grau de conhecimento sobre a composição do sistema dinâmico que deve ser obtido pelos participantes e o nível de sincronia que deve ser observado no mesmo. A segunda contribuição desta tese é formada principalmente por um conjunto de protocolos para reconfiguração de sistemas de quóruns, os quais podem ser divididos em: *(1)* algoritmos para inicialização da reconfiguração; *(2)* algoritmos para geração de novas visões do sistema; e *(3)* algoritmos para instalação das visões atualizadas. Várias combinações destes algoritmos são possíveis, resultando em um sistema com diferentes características e garantias. Estes protocolos são completamente desacoplados dos protocolos de leitura e escrita no registrador, facilitanto a integração dos mesmos com os mais variados sistemas de quóruns encontrados na literatura, além de aumentar o desempenho do sistema. A última grande contribuição desta tese refere-se à adição de suporte à reconfiguração em replicação Máquina de Estados. Nestas reconfigurações, tanto o conjunto de participantes do sistema quanto parâmetros da replicação podem ser altera-

dos, resultando em um sistema bastante robusto e flexível. Todas as soluções propostas nesta tese suportam a presença de participantes maliciosos no sistema e fornecem protocolos que incorporam os atributos de confiabilidade, disponibilidade e integridade para suas aplicações.
Palavras-chave: sistemas distribuídos dinâmicos, tolerância a intrusões, o problema do consenso, sistemas de quóruns, replicação Máquina de Estados.

ABSTRACT

Many changes in the form of designing applications have been motivated by the emergence of new technologies for communications and by the increasing resource availability in networks. These changes had introduced the dynamic distributed systems, that are systems where resources are able to join and to leave it at any time. Protocols designed for these systems should be able to detect and to handle these events, allowing its reconfiguration at runtime. Thus, the participants of these applications are heterogeneous and unreliable. In the development of secure and reliable applications, some problems are identified as building blocks since they form the basis of almost all solutions used in these applications. Thus, it is necessary to design protocols to solve these problems in dynamic distributed systems. As participants of these systems are unreliable, it is essential that these protocols provide dependability attributes.

This work studies and proposes protocols to solve some of these fundamental problems in dynamic distributed systems, that are: the consensus problem, quoruns systems and state machine replication. The consensus problem is studied in unknown networks, where we define conditions that are necessary and sufficient to solve consensus. These conditions specify the degree of knowledge about the system composition that must be acquired by participants and the level of synchrony that must be provided by the system. The second contribution of this thesis is formed by a set of protocols used to reconfigure quoruns systems, that are divided into: *(1)* algorithms for reconfiguration initialization; *(2)* algorithms for new views generation; and *(3)* algorithms to install updated views. Several combinations of these algorithms are possible, resulting in a system with different characteristics and guarantees. These protocols are decoupled from protocols to write and read the register, what makes it easy to adapt it to any quorum system and increases the system performance. The last contribution of this work is the development of a state machine replication able to execute reconfigurations. Through reconfigurations, the state machine is able to change the set os system participants, that implements it, and also some reconfiguration parameters, resulting in a robust and flexible system. Each solution proposed in this thesis is able to tolerate the presence of malicious participants in the system, providing protocols that incorporate the attributes of reliability, availability and integrity.

Keywords: distributed dynamic systems, intrusion tolerance, the consensus problem, quoruns systems, state machine replication.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ALGORITMOS

SUMÁRIO

1 INTRODUÇÃO

As novas tecnologias de comunicação e a maior disponibilidade de recursos em redes de comunicação vêm provocando profundas mudanças na forma de se projetar aplicações nos dias de hoje. Essa evolução não só permitiu que as comunicações se tornassem mais rápidas e baratas, mas também possibilitou o desenvolvimento de novas aplicações e formas de interação que não eram anteriormente possíveis. Dentre essas novas formas de aplicações e sistemas estão as classificadas como Sistemas Distribuídos Dinâmicos (MOSTéFAOUI et al., 2005; BALDONI et al., 2007), que destacaram-se por estarem relacionadas com tecnologias amplamente adotadas e utilizadas pela sociedade, como por exemplo as redes móveis *ad hoc* (MANETs) (ILYAS; DORF, 2003; DJENOURI et al., 2005), as redes de sensores sem fio (RSSF) (ZHAO; GUIBAS, 2004), as redes entre pares (*Peer-to-Peer* - P2P) (STEINMETZ; WEHRLE, 2005), e as grades computacionais (FOSTER, 2002).

A característica principal de um sistema distribuído dinâmico, a qual o difere significativamente dos sistemas convencionais (sistemas distribuídos estáticos), está relacionada com a possibilidade de recursos componentes entrarem e saírem do sistema em qualquer momento. Assim, pode-se definir informalmente um sistema distribuído dinâmico como um sistema que executa continuamente, sendo que seus modelos computacionais devem considerar um número variável de processos (ou recursos) fazendo parte do sistema a cada instante ou intervalo de tempo. Desta forma, as topologias e composições de processos podem mudar arbitrariamente durante uma execução da aplicação.

Outra característica destes sistemas, a qual deve ser considerada no desenvolvimento de protocolos e aplicações, é a possibilidade de que cada processo consiga interagir apenas com parte dos processos que compõem o sistema distribuído. Nestes novos ambientes computacionais, por se caracterizarem em sistemas abertos e auto-organizáveis, os recursos podem ser voluntários e muitas vezes são desconhecidos.

Recentemente, alguns trabalhos apresentaram esforços no sentido de introduzir uma definição formal para sistemas distribuídos dinâmicos (MOSTéFAOUI et al., 2005; BALDONI et al., 2007), apresentando modelos genéricos a partir dos quais as aplicações devem ser projetadas e desenvolvidas. Estas definições retratam principalmente a dinâmica destes sistemas assumindo modelos que consideram a ocorrência de infinitos processos durante o ciclo de vida do sistema. Estes modelos assumem diferentes graus de concorrência de processos no sistema, i.e., número máximo de processos que podem estar ativos simultaneamente participando da execução de um algoritmo em um

mesmo período de tempo (MERRITT; TAUBENFELD, 2000). Nestes modelos, o ciclo de vida do sistema é visto em termos de "épocas" (ou períodos), durante as quais são consideradas determinadas hipóteses sobre o número de processos que podem concorrer e participar da execução no sistema.

1.1 MOTIVAÇÃO

Os sistemas distribuídos dinâmicos (MOSTéFAOUI et al., 2005; BALDONI et al., 2007) são caracterizados como sistemas distribuídos onde as topologias, bem como o conjunto de participantes do sistema, estão sujeitos a modificações arbitrárias, causadas principalmente pela entrada e saída voluntária de participantes, pela mobilidade dos mesmos, ou por eventos como falhas. A dinâmica no conjunto de participantes envolvidos nas computações nestes ambientes (entradas e saídas em tempos arbitrários) é designada pelo termo *churn* (GODFREY et al., 2006) e sua ocorrência gera perturbações no sistema, as quais podem resultar em: perdas de mensagens; inconsistências em dados; aumento da latência observada pelo usuário; aumento no uso de largura de banda; interrupções parciais ou totais de serviços; indisponibilidade de recursos e variações bruscas no desempenho do sistema.

As aplicações projetadas para estes ambientes devem evoluir mesmo diante da mobilidade de recursos, do *churn* e do uso (muitas vezes massivo) de recursos distribuídos e desconhecidos. Para tanto, é necessário propriedades de auto-organização, de auto-adaptação, de execução descentralizada de tarefas, dentre outras. Estas propriedades não são triviais de serem alcançadas nesse tipo de ambiente. Assim, sistemas distribuídos dinâmicos apresentam vários desafios inerentes ao projeto e operação de suas aplicações.

Um destes desafios consiste em projetar e desenvolver arquiteturas e algoritmos adaptativos, capazes de detectar e tratar mudanças que ocorrem na composição de uma aplicação distribuída, permitindo a mesma se reconfigurar em tempo de execução. Esta propriedade de auto-adaptação tem por objetivo assegurar que o sistema continuará atendendo, durante todo o seu ciclo de vida, aos requisitos de funcionamento da aplicação, evitando a deterioração da qualidade de serviço e, principalmente, da segurança.

Devido a estas características dos sistemas distribuídos dinâmicos, os participantes das computações distribuídas nestes ambientes são caracterizados pela heterogeneidade, capacidade de computação variável e pela não confiabilidade. Diante disso, a segurança e a confiabilidade das aplicações são outros desafios que devem ser abordados nestes modelos, pois a maioria destas aplicações possui exigências relacionadas com a sua segurança de funcionamento (*dependability*) (AVIZIENIS et al., 2004).

Segurança de funcionamento refere-se à capacidade do sistema sobreviver a falhas em parte de seus componentes. Estas falhas podem ser de natureza acidental (ex. erros de programação) ou maliciosa (ex. ataques bem sucedidos que levam à ocorrência de intrusões no sistema, comprometendo as propriedades de segurança e o comportamento dos componentes invadidos). Assim, tolerância a faltas e a intrusões é um requisito de qualidade de serviço fundamental para as aplicações executadas em sistemas distribuídos dinâmicos. Um sistema com segurança de funcionamento possui um comportamento que evolui segundo suas especificações, mesmo diante da ocorrência destes eventos de falhas, ataques e intrusões no sistema. A importância de um sistema ser seguro e confiável reside no fato de que, por exemplo, um sistema desprotegido pode ser completamente comprometido por um ataque bem sucedido resultando então numa intrusão neste sistema. Neste caso, o invasor poderia obter o completo controle do sistema.

Neste sentido, as características de mobilidade e do uso de recursos desconhecidos definem novas roupagens a problemas clássicos de sistemas distribuídos. Algoritmos e protocolos desenvolvidos para solucionar problemas em sistemas estáticos não podem ser simplesmente empregados na solução dos mesmos problemas em um sistema distribuído dinâmico, pois tais protocolos não são flexíveis o suficiente para tolerar as mudanças que ocorrem nestes ambientes. Existe então a necessidade do desenvolvimento de protocolos para estes ambientes dinâmicos, os quais apresentam novos desafios no projeto de algoritmos para aplicações distribuídas.

1.1.1 Foco da Tese

Os objetos de pesquisa que são descritos nesta tese envolvem a definição de novos algoritmos que permitam aplicações evoluírem em sistemas distribuídos dinâmicos. Estes algoritmos devem atender aos requisitos de segurança e de confiabilidade das aplicações, provendo uma série de atributos determinantes para o comportamento correto das aplicações nestes ambientes, mesmo diante da ocorrência de falhas e intrusões.

Vários problemas comuns são identificados como peças fundamentais no desenvolvimento de aplicações distribuídas seguras e confiáveis. Por isso, tais problemas foram definidos como blocos básicos de construção. Por exemplo, o problema do consenso (LAMPORT et al., 1982; FISCHER et al., 1985; TOUEG, 1984; CHANDRA; TOUEG, 1996; CASTRO; LISKOV, 2002; CORREIA et al., 2006), e de um modo mais geral os algoritmos de acordo, formam a base para a solução da maioria dos problemas encontrados no desenvolvimento de sistemas distribuídos confiáveis. Estes problemas de-

finem situações que permitem que os participantes da computação distribuída coordenem suas ações de forma a manter a consistência em seus estados e garantir o progresso do sistema.

Como a construção de aplicações distribuídas envolve a utilização de protocolos desenvolvidos para solucionar estes problemas fundamentais, torna-se essencial o desenvolvimento de tais protocolos, sendo que os mesmos devem incorporar propriedades de segurança de funcionamento para garantir as propriedades destas aplicações. Vale salientar que a elaboração destes protocolos não é uma tarefa trivial, visto que pequenas mudanças nos protocolos existentes podem exigir modificações em toda a estrutura dos mesmos. Além disso, limitações apresentadas em protocolos existentes fazem com que seja necessário o desenvolvimento de soluções completas.

A complexidade da solução destes problemas é variada e depende de uma série de fatores, geralmente definidos no modelo de sistema ao qual cada protocolo é endereçado. Esta complexidade aumenta na medida em que consideramos um sistema distribuído dinâmico, que adiciona uma série de incertezas que devem ser suportadas pelos protocolos. Também em vista disso, a quantidade de soluções para estes problemas em ambientes estáticos é muito superior às encontradas para ambientes dinâmicos. No entanto, é imprescindível a utilização de protocolos que tratem destes problemas em ambientes dinâmicos, para que aplicações atendam aos requisitos de segurança de funcionamento também nestes ambientes.

A hipótese da ocorrência de faltas maliciosas e intrusões é uma questão que muitas vezes é negligenciada em trabalhos sobre sistemas distribuídos dinâmicos. O comportamento malicioso permanece como um dos principais problemas em aberto nestes sistemas. Técnicas tradicionais de tolerância a faltas maliciosas usadas em sistemas estáticos não podem ser usadas em sistemas dinâmicos, pois dependem fundamentalmente da pré-existência de parâmetros relacionados ao número total de entidades presentes no sistema e ao número de faltas toleradas. Portanto, existe a necessidade da concepção de novos modelos e mecanismos de segurança e tolerância a faltas, projetados especificamente para sistemas distribuídos dinâmicos.

Neste sentido, o desenvolvimento de protocolos endereçados para ambientes dinâmicos representa novos desafios no projeto de algoritmos para aplicações distribuídas. Nestes ambientes, a tarefa de garantir as propriedades de correção (*safety*) e de terminação das computações (*liveness*) é ainda mais complicada quando se compara com um ambiente estático.

Além do problema do consenso, é objeto de estudo desta tese soluções para memória compartilhada em ambientes dinâmicos. Neste sentido, este trabalho investiga e propõe soluções para Sistemas de Quóruns Bizantinos (MALKHI; REITER, 1998a), voltadas para ambientes dinâmicos. Além

disso, esta tese também aborda soluções para a reconfiguração dos parâmetros (conjunto de processos e quantidade de faltas suportadas) da replicação baseada em Máquina de Estados (SCHNEIDER, 1990), possibilitando o emprego da mesma em sistemas distribuídos dinâmicos.

A maneira como estes problemas são resolvidos pode ser, em parte, particular a cada uma das tecnologias envolvidas: redes MANETs, redes *Peer-to-Peer*, ou outra. Neste trabalho, independente do tipo de ambiente computacional, os processos que participam destes sistemas formam uma rede colaborativa, onde as computações e as comunicações são realizadas através da cooperação entre os mesmos. Além disso, devido às entradas e saídas aleatórias de processos nestes sistemas, não é possível prever a existência de entidades centrais responsáveis por prover funções críticas (controle de filiação e de acesso, por exemplo).

1.2 OBJETIVOS DA TESE

1.2.1 Objetivo Geral

O objetivo geral desta tese é propor protocolos e algoritmos para a solução de diversos problemas fundamentais encontrados no desenvolvimento de aplicações distribuídas. Estas soluções sempre estão voltadas para ambientes dinâmicos. Assim, os protocolos propostos devem possuir capacidade de adaptação a estes ambientes, prevendo mudanças na composição do grupo de participantes da computação distribuída. Além disso, os protocolos desenvolvidos devem incorporar propriedades de segurança de funcionamento, provendo os atributos de confiabilidade, disponibilidade e integridade para suas aplicações.

Através da definição destes protocolos e algoritmos, espera-se chegar a conclusões mais gerais sobre o processo envolvido na adequação de algoritmos e protocolos para sistemas distribuídos dinâmicos. Para isso, também é objetivo desta tese a realização de estudos a respeito de conceitos, modelos e problemas relacionados com a elaboração e projeto de protocolos e aplicações para sistemas distribuídos dinâmicos.

1.2.2 Objetivos Específicos

Dentro do objetivo geral da elaboração de protocolos e algoritmos para sistemas distribuídos dinâmicos, os seguintes objetivos específicos são abordados:

1. Analisar as implicações de segurança de funcionamento provenientes da execução de aplicações distribuídas em ambientes dinâmicos, i.e., analisar quais são os riscos e vulnerabilidades a que estão sujeitas tais aplicações.

2. Propor algoritmos e protocolos que suportem e facilitem o desenvolvimento de aplicações distribuídas para ambientes distribuídos dinâmicos.

3. Propor algoritmos e protocolos para coordenação de participantes em sistemas distribuídos dinâmicos.

4. Propor algoritmos e protocolos para armazenamento de dados (memória compartilhada) em sistemas distribuídos dinâmicos.

5. Avaliar os algoritmos e protocolos propostos através de simulações e/ou implementações de sistemas reais.

Os algoritmos e protocolos, relacionados com os objetivos 2, 3 e 4, devem levar em consideração as conclusões obtidas através dos estudos que visam cumprir o objetivo 1.

1.3 ORGANIZAÇÃO DO TEXTO

A organização deste texto reflete as diversas etapas cumpridas para alcançar os objetivos específicos listados na seção anterior.

O Capítulo 2 apresenta conceitos fundamentais da teoria de sistemas distribuídos dinâmicos, discutindo as principais diferenças entre este modelo e o modelo estático. Além disso, este capítulo apresenta alguns exemplos de sistemas distribuídos dinâmicos, juntamente com os principais desafios encontrados no desenvolvimento de aplicações distribuídas sobre estes ambientes.

O Capítulo 3 apresenta os principais conceitos sobre segurança de funcionamento em sistemas distribuídos dinâmicos, abordando algumas vulnerabilidades apresentadas por estes ambientes. Este capítulo também discute alguns problemas fundamentais encontrados na construção de aplicações distribuídas, abordando as principais propostas encontradas na literatura para a solução destes problemas em ambientes dinâmicos.

O Capítulo 4 aborda o problema do consenso em um novo ambiente, no qual os participantes de determinada computação distribuída são inicialmente desconhecidos. Neste modelo, o conjunto de participantes pode sofrer alterações entre as execuções do protocolo. Neste sentido, são determinadas

as condições necessárias e suficientes para que o consenso admita solução nestes ambientes, onde os participantes podem se comportar de forma maliciosa.

Uma série de protocolos para reconfiguração de sistemas de quóruns bizantinos são apresentados no Capítulo 5, no qual são discutidos os vários problemas relacionados com a utilização destes protocolos em diferentes modelos de sistemas. Deste modo, são propostas soluções para reconfiguração de sistemas de quóruns bizantinos para cada modelo. Além disso, são apresentados protocolos para tolerar também clientes maliciosos nestes ambientes dinâmicos. Por fim, vários aspectos das soluções propostas são discutidos e algumas soluções alternativas são apontadas.

O Capítulo 6 apresenta alguns conceitos envolvidos na adição do suporte à reconfigurações em uma replicação Máquina de Estados, seguindo uma abordagem mais prática. Neste sentido, é discutido como prover tal funcionalidade no sistema BFT-SMART, o qual é uma concretização de uma replicação Máquina de Estados.

Finalmente, no Capítulo 7 são apresentadas as conclusões do trabalho, bem como as perspectivas futuras.

2 SISTEMAS DISTRIBUÍDOS DINÂMICOS

Este capítulo descreve as principais características dos sistemas distribuídos dinâmicos, as quais são fundamentais para entender como estes sistemas funcionam e também para observar as diferenças existentes entre estes ambientes e os clássicos sistemas distribuídos estáticos. Para facilitar a compreensão, são apresentados alguns exemplos de sistemas distribuídos dinâmicos, os quais também são úteis para entender quais são as tecnologias que permitem a implementação destes sistemas.

2.1 INTRODUÇÃO

Os sistemas distribuídos dinâmicos (MOSTéFAOUI et al., 2005; BALDONI et al., 2007) são caracterizados como sistemas distribuídos onde as topologias bem como o conjunto de participantes[1] do sistema estão sujeitos a modificações arbitrárias, causadas principalmente pela entrada e saída voluntária de participantes no sistema, ou por eventos como falhas. Algoritmos empregados nesta classe de sistemas devem tratar a ocorrência desses eventos em tempo de execução, possibilitando que o mesmo se adapte automaticamente às mudanças em sua composição, permitindo que aplicações (ou computações) evoluam normalmente nestes ambientes dinâmicos.

Os algoritmos e protocolos desenvolvidos para solucionar problemas em sistemas distribuídos estáticos não podem ser simplesmente empregados na solução dos mesmos problemas em um sistema distribuído dinâmico. O principal motivo que impede esta utilização é que alguns parâmetros de configuração, nestes sistemas distribuídos estáticos, são assumidos como fixos e nunca sofrem alterações. Este é o caso, por exemplo, do número total de participantes presentes no sistema e do número de falhas toleradas pelo mesmo. Assim, os protocolos desenvolvidos para estes ambientes consideram que estes parâmetros são invariantes e que sistemas distribuídos estáticos nunca serão reconfigurados em termos de suas composições. Em sistemas distribuídos dinâmicos, essas hipóteses e parâmetros não podem ser assumidos previamente em razão do dinamismo do sistema.

Portanto, o desenvolvimento de protocolos endereçados para ambientes dinâmicos representa novos desafios no projeto de algoritmos para aplicações distribuídas. Nestes ambientes dinâmicos, a tarefa de garantir propriedades das aplicações, como a correção (*safety*) e a terminação (*liveness*) das

[1] No decorrer deste texto serão usadas alternadamente, mas com o mesmo sentido, as designações participantes, entidades, nós e processos, de acordo com o contexto em foco.

computações, é ainda mais complicada quando comparada a mesma tarefa em ambientes estáticos.

Este capítulo primeiramente apresenta uma visão geral sobre alguns conceitos relacionados com sistemas distribuídos estáticos. Esta explicação inicial sobre sistemas distribuídos estáticos é útil no entendimento dos conceitos envolvidos na definição de sistemas distribuídos dinâmicos, onde é possível comparar as características destes dois ambientes. Além disso, são discutidas algumas tecnologias relacionadas com ambientes dinâmicos. Por fim, este capítulo descreve alguns desafios encontrados no projeto e desenvolvimento de protocolos para aplicações que executam suas computações em ambientes dinâmicos.

2.2 SISTEMAS DISTRIBUÍDOS ESTÁTICOS

Um sistema distribuído pode ser caracterizado por meio de propriedades comportamentais e propriedades estruturais. Estas propriedades definem modelos computacionais, que variam desde o tipo de falta suportada até o nível de sincronia adotado.

Os modelos computacionais estáticos são os mais estudados. Em um sistema distribuído estático o número total de processos que participam de uma computação distribuída, comumente denotado como n, é um parâmetro estático, previamente conhecido por todos os processos do sistema. As identidades e os endereços destes processos também são conhecidos por todos. Assim, desconsiderando-se possíveis falhas, o conjunto de processos que compõem o processamento distribuído não sofre alterações durante a existência do sistema considerado. Além disso, a camada de rede subjacente, utilizada pelas aplicações que executam neste modelo, comumente é assumida como completamente conectada, i.e., um processo pode enviar/receber mensagens diretamente para qualquer outro processo. Deste modo, não é necessário se preocupar com o roteamento das mensagens que fica abstraído na camada de rede subjacente.

O nível de sincronia adotado pelos protocolos desenvolvidos para sistemas distribuídos estáticos pode variar desde um modelo síncrono até um modelo completamente assíncrono. No modelo síncrono os tempos necessários para o envio de mensagens e para os processamentos locais nos processos são limitados e conhecidos. Já no modelo assíncrono estes tempos não são delimitados e, portanto, são desconhecidos. Outro modelo comumente empregado no projeto de protocolos para estes sistemas é o parcialmente síncrono (DWORK et al., 1988) que pode ser entendido como uma variação entre os dois modelos anteriormente citados. A ideia por trás deste último modelo de

sincronismo é de que o sistema trabalha de forma assíncrona (não respeitando nenhum limite de tempo) a maior parte do tempo. Porém, durante períodos de estabilidade os tempos de transmissão de mensagens e de processamentos locais são delimitados, apesar de não conhecidos.

Outra variação nos modelos definidos para sistemas distribuídos estáticos refere-se ao tipo de falha suportada pelo sistema. Os protocolos geralmente são projetados para tolerar faltas por parada (*crash*) ou faltas Bizantinas (também conhecidas como maliciosas ou arbitrárias (LAMPORT et al., 1982)). Um processo que apresenta falha por parada apenas deixa de processar e enviar mensagens. Já um processo que apresenta falha bizantina pode exibir qualquer comportamento, podendo parar, omitir envios ou entregas de mensagens, ou desviar arbitrariamente de sua especificação. Em um sistema distribuído estático, o número máximo de falhas suportadas pelo sistema (comumente denotado por f) é estático. É importante citar que algumas propostas de sistemas distribuídos estáticos, com o objetivo de aumentar a capacidade de tolerância a faltas, apresentam mecanismos pró-ativos que recuperam periodicamente servidores (réplicas ou processos) falhos (ex.: (CASTRO; LISKOV, 2002; ZHOU et al., 2002; ALCHIERI et al., 2009a, 2009b, 2009c)). Nestas propostas, o limite de falhas de processos suportado pelo sistema é delimitado durante a duração de um período de recuperação, mas é ilimitado durante o ciclo de vida do sistema.

2.3 SISTEMAS DISTRIBUÍDOS DINÂMICOS

A característica principal de um sistema distribuído dinâmico (SDD) que o diferencia significativamente de sistemas distribuídos estáticos, está relacionada com a possibilidade de processos entrarem e saírem do sistema em qualquer momento. Estas entradas e saídas em um SDD determinam a possibilidade de mudanças arbitrárias nas topologias e no conjunto de processos que compõem estes sistemas. Uma maneira usual de tratar estas mudanças é definir intervalos de execuções nos quais a composição e a topologia do sistema são assumidos como constantes. A passagem entre estas execuções caracteriza as mudanças de composição e topologia. Um sistema distribuído dinâmico, portanto, executa continuamente, através destes intervalos de execuções com parâmetros estáticos. Esta discretização dos parâmetros permite ainda que em cada uma destas execuções seja assumido um número arbitrário, porém constante, de processos e que cada um destes processos possua uma visão parcial desta composição do sistema.

Neste sentido, os sistemas distribuídos dinâmicos apresentam as seguintes características básicas (PIERGIOVANNI, 2005): *(i)* execução contí-

nua, durante a qual os nós podem entrar e sair, ou mesmo falhar, de forma arbitrária; *(ii)* número de nós no sistema não é conhecido a priori e pode mudar sem quaisquer limites; *(iiii)* instante de tempo de ocorrência dos eventos de entrada e saída de participantes do sistema é desconhecido, i.e., estes eventos podem ocorrer a qualquer tempo.

Recentemente, alguns trabalhos apresentaram esforços no sentido de apresentar uma definição formal para sistemas distribuídos dinâmicos (MOSTéFAOUI et al., 2005; BALDONI et al., 2007). Estes trabalhos apresentam modelos genéricos para SDD, a partir dos quais as aplicações devem ser projetadas e desenvolvidas para operarem sobre estes ambientes dinâmicos. Além disso, em (MOSTéFAOUI et al., 2005) um conjunto de primitivas (abstrações) são definidas para assegurar a evolução e término das computações executadas no sistema. Este trabalho também apresenta uma metodologia para tradução dos protocolos projetados no modelo estático para protocolos equivalentes no modelo dinâmico.

Tabela 1 – Modelos de sistemas com infinitos processos.

Nome	**Descrição**
Modelo de Chegada Finita (M_1)	O sistema possui ∞ processos, porém, apenas uma parcela finita destes participa de cada execução de um algoritmo. Esse limite, apesar de existente, não é conhecido pelos processos e varia de execução para execução.
Modelo de Chegada Infinita (M_2)	O sistema possui ∞ processos, porém, em cada intervalo finito de tempo somente um número finito e desconhecido de processos executam alguma ação.
Modelo de Chegada Infinita com Concorrência b-Limitada (M_2^b)	Semelhante ao M_2, mas com a restrição de que a concorrência em cada execução é limitada por uma constante b conhecida.
Modelo de Chegada Infinita com Concorrência Limitada (M_2^{finito})	Semelhante ao M_2^b, porém, o limite de concorrência, embora finito, não é conhecido.
Modelo de Concorrência Infinita (M_3)	Sistema possui ∞ processos em cada execução e em um intervalo de tempo finito podem haver ∞ processos executando ações. Modelo de interesse teórico apenas.

Estas definições para SDD (MOSTéFAOUI et al., 2005; BALDONI et al., 2007) retratam a dinâmica destes sistemas assumindo modelos que consideram a ocorrência de *infinitos processos* durante o ciclo de vida do sistema. Estes modelos assumem diferentes graus de concorrência de processos no sistema, i.e., número máximo de processos que podem estar ativos simulta-

neamente participando da execução de um algoritmo em um mesmo período de tempo (MERRITT; TAUBENFELD, 2000). Nestes modelos, o ciclo de vida do sistema é visto como uma sequência de *execuções*, durante as quais são consideradas determinadas hipóteses sobre o número de processos que podem concorrentemente participar da execução do sistema. Alguns destes modelos, apresentados em (AGUILERA, 2004), são brevemente descritos na Tabela 1.

As definições para sistemas distribuídos dinâmicos apresentadas em (MOSTéFAOUI et al., 2005) e (BALDONI et al., 2007) são brevemente descritas nas seções seguintes.

2.3.1 SDD com Chegada Finita de Processos

A proposta de modelo para SDD apresentada em (MOSTéFAOUI et al., 2005) baseia-se no *modelo de chegada finita* de processos. Assim, o sistema é formado por um conjunto infinito de processos $\prod = \{..., p_i, p_j, p_k, ...\}$ unicamente identificados, porém, apresenta um número finito de processos em cada execução. O limite de processos presentes em cada execução é desconhecido e pode variar de uma execução para outra. Os processos comunicam-se por meio de canais ponto a ponto e utilizam primitivas de requisição-resposta (*query-response*) e de difusão confiável persistente (*persistent reliable broadcast*) (FRIEDMAN et al., 2005b, 2005c). Este modelo assume também um sistema assíncrono onde os processos estão sujeitos a falhas por parada (*crash*).

2.3.1.1 Progresso do sistema

Em um SDD é possível que nenhum processo permaneça tempo suficiente para completar alguma computação útil. Assim, estes sistemas devem apresentar alguma forma de estabilidade para conseguir chegar ao fim de uma computação. Neste sentido, (MOSTéFAOUI et al., 2005) introduz a propriedade de **estabilidade terminal** (*eventual stability*), a qual afirma que sistemas distribuídos dinâmicos são caracterizados por uma sucessão de períodos instáveis seguidos por períodos estáveis. No entanto, é possível que determinadas computações também terminem durante períodos de instabilidade, mas o progresso do sistema não pode ser garantido durante estes períodos. Desta forma, o sistema deve seguir (prover) determinadas **condições de estabilidade** (*stability conditions*) durante um período suficiente para garantir progresso e término de computações.

Neste sentido, é estabelecido um parâmetro (α) que indica o número mínimo de processos ativos que devem estar presentes no sistema, durante um período longo o suficiente para assegurar a evolução e término da computação distribuída. Este parâmetro α é equivalente ao limiar $n - f$ considerado no modelo estático, i.e., número de processos corretos presentes no sistema. Assim, parâmetros como o número total de nós (n) e o número de falhas toleradas (f), próprios do modelo estático, não são considerados como premissas fundamentais no projeto de algoritmos para sistemas distribuídos dinâmicos. Portanto, a condição fundamental para estes sistemas é a proporção α de processos ativos que devem persistir no sistema.

2.3.1.2 Gerenciamento do Grupo de Processos

Um processo pode entrar no sistema (operação *join()*) e, depois de algum tempo falhar ou sair voluntariamente (operação *leave()*). Um processo também pode voltar a integrar o sistema utilizando uma nova identidade, sendo, desta maneira, considerado um novo processo. Devido à premissa de chegada finita, o número de vezes que um processo pode sair e entrar no sistema é limitado para uma dada execução.

O conjunto de processos presentes no sistema em um tempo t, é denotado por $up(t)$. Os processos presentes no sistema e que não apresentaram falhas e nem deixaram o sistema através da operação *leave()* formam o conjunto *STABLE*. Dado um intervalo I definido pelos tempos de início (t_b) e fim (t_e), o conjunto estável associado a este intervalo é definido como:

$$STABLE(I) = \{i | \exists t \in [t_b, t_e] : \forall t' : t \leq t' \leq t_e : p_i \in up(t')\},\ |STABLE| \geq \alpha$$

Note que, o número de processos pertencentes ao conjunto *STABLE* precisa ser maior ou igual ao parâmetro α para que o fim de uma determinada computação seja garantido.

2.3.1.3 Modelo Comunicação

O modelo de comunicação discutido em (MOSTéFAOUI et al., 2005) possui como base as primitivas *query-response* e *persistent reliable broadcast*, que foi introduzida em (FRIEDMAN et al., 2005b, 2005c) e corresponde à versão adaptada da primitiva *reliable broadcast* definida para modelos estáticos.

- *Query-response*: Esta abstração define um padrão de comunicação en-

tre processos onde o iniciador envia a mensagem (requisição) para todos os outros processos. Qualquer processo que receber esta requisição envia uma resposta para o iniciador. O protocolo termina quando o iniciador receber α respostas.

A principal diferença desta primitiva quando comparada com sua definição para o modelo estático refere-se ao limiar de respostas necessárias para considerar como completo o processamento de uma requisição. No modelo estático este limiar é normalmente assumido como $n-f$, enquanto neste modelo o limiar é α.

Formalmente, este padrão satisfaz as seguintes propriedades:

- *QR-Validade:* Se uma requisição r é entregue por um processo, então r foi enviada por um processo;
- *QR-Uniformidade:* Uma requisição r é entregue no máximo uma vez por um processo;
- *QR-Terminação*: Se um processo p_i não falhar enquanto estiver realizando uma requisição r, então r gerará pelo menos α respostas.

Uma implementação simples deste padrão consiste em fazer com que o iniciador de uma requisição continuamente (ou de tempos em tempos) inunda o sistema com a requisição até receber pelo menos α respostas.

- *Persistent reliable broadcast*: A segunda abstração de comunicação fornecida por este modelo consiste em uma primitiva para *broadcast* de mensagens. Este *broadcast* consiste no envio das mensagens para um subgrupo de processos suficientemente grande, de forma que uma mensagem não seja perdida pelo sistema se algum processo falhar. Assumindo que cada mensagem m possui um tipo $type(m)$ e uma data lógica $ld(m)$, esta abstração é definida com base nas seguintes propriedades:

  - *PRB-Validade:* Se uma mensagem m é entregue por um processo, então m foi difundida por um processo;
  - *PRB-Uniformidade:* Uma mensagem m é entregue no máximo uma vez por um processo;
  - *PRB-Terminação*: Se um processo que difunde uma mensagem m com uma data lógica ld não falhar enquanto está executando a difusão, ou m é entregue por um processo, então qualquer processo p_j, tal que $j \in STABLE$, termina por entregar toda mensagem m' (do mesmo tipo de m) tal que $ld(m') = ld(m)$ ou uma mensagem m' (do mesmo tipo de m) tal que $ld(m') > ld(m)$.

A principal propriedade desta primitiva é a de terminação, responsável por assegurar a vivacidade (*liveness*) da entrega de mensagens. Devido ao assincronismo do sistema, à possibilidade de falhas de processos e aos eventos de entrada e saída de processos no sistema, não é possível garantir que todos os processos que estão ativos quando uma mensagem é difundida entreguem tal mensagem. No entanto, após uma mensagem ser entregue por um processo, todos os processos que permanecem ativos no sistema devem acabar por entregar tal mensagem ou uma mensagem do mesmo tipo enviada posteriormente.

Uma implementação desta abstração pode seguir o seguinte procedimento (MOSTéFAOUI et al., 2005; FRIEDMAN et al., 2005b): ao receber uma mensagem m, um processo primeiro encaminha m para todos os outros processos, e somente então entrega m. Além disso, os novos processos que entram no sistema (grupo) devem enviar uma mensagem *inquiry* via *broadcast* para os demais processos. Estes devem responder com o seu estado e, para cada tipo de mensagem, a data lógica da última mensagem entregue.

2.3.1.4 Considerações

Apenas é possível transpor para este modelo os algoritmos de sistemas distribuídos estáticos que usam as mesmas abstrações de comunicação definidas neste modelo e que assumem um limiar mínimo de processos que não podem falhar ($\alpha = n - f$). Algoritmos de modelos estáticos que usam os parâmetros n e f individualmente não são compatíveis com este modelo dinâmico. Outro aspecto relevante é que os algoritmos são aplicados a falhas por parada (*crash*).

A Tabela 2 apresenta um paralelo entre modelos estáticos e o modelo dinâmico descrito nesta seção, de maneira a facilitar a percepção das equivalências entre os elementos de ambos os modelos.

Tabela 2 – Modelo de sistema distribuído estático vs. dinâmico.

	Sistema Distribuído Estático	*Sistema Distribuído Dinâmico*
Classificação de processos	p_i é correto	$i \in STABLE$
Limiar de garantia de progresso	$n - f$	α
Primitiva *query-response*	$n - f$ respostas esperadas	α respostas esperadas
Primitiva de difusão	*Reliable Broadcast*	*Persistent Reliable Broadcast*

2.3.2 SDD com Chegada Infinita de Processos

Em (BALDONI et al., 2007) um sistema distribuído dinâmico é definido considerando-se dois fatores de dinamicidade do sistema: *(1)* tamanho do sistema, i.e., número de nós que o compõe; e *(2)* dimensão geográfica (vizinhança) percebida por cada entidade que compõe o sistema. A dimensão relacionada ao tamanho dinâmico do sistema, em termos de número de entidades, é derivada do *modelo de chegada infinita* e suas variações (Tabela 1). Neste sentido, o sistema está sujeito a eventos de entrada e saída de nós, e o nível de concorrência durante uma execução do mesmo pode ser: *(i)* limitado por uma variável b conhecida; *(ii)* limitado, porém, tal limite é desconhecido; e *(iii)* ilimitado. Em relação à dimensão geográfica do sistema, assume-se que cada processo possui somente uma visão parcial do sistema, *i.e.*, pode interagir diretamente apenas com um subconjunto de processos que estão presentes no sistema. A formalização desta dimensão é realizada através da teoria de grafos:

- Em qualquer momento, o sistema é representado por um grafo $G = (P,E)$, onde P é o conjunto de processos que estão presentes no sistema neste momento ($P = \{..., p_i, p_j, p_k, ...\}$) e E é o conjunto de arestas que estabelece a relação simétrica de vizinhança entre pares de processos. Se $(p_i, p_j) \in E$, então existe um canal confiável e bidirecional conectando p_i e p_j.

- A chegada e partida de processos, bem como a mobilidade dentro do sistema, implica na adição e/ou remoção de vértices e arestas de G, resultando em um novo grafo G'. A adição de um processo p_i em G, por exemplo, gera um novo grafo G' contendo o novo vértice p_i e algumas arestas (p_i, p_j), onde p_j representa um processo que já pertencia a G e que em G' está diretamente conectado a p_i. Já a remoção de um processo p_i gera um novo grafo G' onde o vértice p_i e as suas arestas relacionadas são suprimidos.

- Seja $\{G_n\}_{run}$ a sequência de grafos G_n pelos quais o sistema passa durante uma execução e $\{D_n\}_{run}$ o conjunto de diâmetros[2] dos respectivos grafos em $\{G_n\}_{run}$, então os diâmetros podem ter as seguintes propriedades:

  - *Diâmetro limitado e conhecido*: limite dado por uma constante b;
  - *Diâmetro limitado e desconhecido*: limite desconhecido;

[2] O diâmetro de um grafo é o máximo das distâncias entre dois vértices.

– *Diâmetro ilimitado*: limite possivelmente infinito.

Com base nestas definições, alguns novos modelos são definidos. Estes novos modelos são, em essência, extensões dos *modelos de chegada infinita* (M_2, M_2^b e M_2^{finito}) e são representados como $M^{N,D}$, onde N representa o nível de concorrência assumido e D o diâmetro possível da rede. As variáveis N e D podem assumir, individualmente, cada um dos seguintes valores: b (número de entidades/diâmetro limitado em b); n (número de entidades/diâmetro com limite desconhecido); ∞ (número de entidades/diâmetro possivelmente infinito). Assim, a combinação dos possíveis valores para N e D geram os seguintes modelos: $M^{b,b}$, $M^{n,b}$, $M^{\infty,b}$, $M^{n,n}$, $M^{\infty,n}$ e $M^{\infty,\infty}$. Alguns desses modelos são impraticáveis, como é o caso do $M^{\infty,\infty}$ (BALDONI et al., 2007).

2.4 EXEMPLOS DE SISTEMAS DISTRIBUÍDOS DINÂMICOS

A teoria de sistemas distribuídos dinâmicos tem se formado principalmente a partir de trabalhos desenvolvidos no contexto das tecnologias emergentes de *redes móveis ad hoc* (MANETs), *redes de sensores sem fio*, *grades computacionais oportunistas*, *redes overlay* e *redes entre pares* (*peer-to-peer*). Todas estas tecnologias compartilham as características atribuídas a sistemas distribuídos dinâmicos, além do fato de executarem suas aplicações usando recursos distribuídos e de maneira descentralizada, já que, na maioria dos casos, não podem contar com entidades permanentes e estáveis que permitam estabelecer uma infra-estrutura fixa sob o sistema, à qual poderiam ser delegadas funções críticas das aplicações.

Esta seção descreve brevemente as características principais de algumas destas tecnologias sobre as quais os estudos sobre sistemas distribuídos dinâmicos estão sendo desenvolvidos.

2.4.1 Redes Móveis *Ad Hoc* (MANETs)

Talvez uma rede *ad hoc* móvel seja o melhor exemplo de uma instância de sistema distribuído dinâmico. Isso se deve principalmente à grande variação que pode apresentar na sua composição e às incertezas que as aplicações estão sujeitas nestas redes. Conceitualmente, uma rede móvel *ad hoc* (MANET – *Mobile Ad hoc NETwork*) (ILYAS; DORF, 2003) é uma rede temporária formada por dispositivos móveis que se organizam dinamicamente e sem o auxílio de uma entidade administrativa centralizadora ou mesmo de uma infra-estrutura pré-estabelecida.

Estes dispositivos, comumente chamados *nós*, podem apresentar características heterogêneas e formam uma rede com topologia variável, onde a comunicação entre os mesmos é realizada sem a utilização de fios (*wireless*). Nestes ambientes, várias operações em níveis de rede e de aplicações são executadas por meio de cooperação entre os próprios dispositivos (ILYAS; DORF, 2003; DJENOURI et al., 2005).

A tecnologia de MANETs torna possível que dispositivos computacionais sejam capazes de estabelecer uma rede de comunicação em áreas onde não há uma infra-estrutura pré-existente. Esta característica é explorada em um amplo campo de aplicações, tais como: aplicações militares, robótica distribuída, operações em regiões de desastres, controle de tráfego urbano, entre outras.

A área de pesquisa de segurança em MANETs está em grande parte voltada para a proteção da infra-estrutura de comunicação, em particular no que tange às operações de roteamento e de encaminhamento de dados da camada de rede. Este é o aspecto primário a ser protegido, através do qual se torna possível prover segurança a serviços e aplicações das camadas superiores (camada de aplicação, por exemplo). Entretanto, prover segurança nestes ambientes é uma tarefa que envolve muitos desafios, em razão da arquitetura dinâmica, cooperativa e aberta deste tipo de rede, além das características particulares dos dispositivos que compõem uma MANET.

2.4.2 Redes de Sensores sem Fio (RSSF)

Uma rede de sensores sem fio (RSSF) (ZHAO; GUIBAS, 2004) geralmente é formada por uma grande quantidade de nós sensores, que são capazes de comunicar-se entre si. Estas redes podem ser vistas como um tipo especial de rede móvel *ad hoc*. No aspecto organizacional, RSSFs e MANETs são idênticas, pois possuem elementos computacionais que se comunicam diretamente entre si por meio de enlaces de comunicação sem fio (LOUREIRO et al., 2003). No entanto, em MANETs estes elementos computacionais podem estar executando tarefas distintas enquanto que numa RSSF estes elementos tendem a executar uma função colaborativa.

As posições dos sensores que compõem a rede não são pré-determinadas, pois geralmente estes sensores são implantados em locais de difícil acesso com o objetivo de monitorar algum fenômeno. Esta implantação pode ocorrer através de helicópteros onde os sensores são distribuídos aleatoriamente sobre a área monitorada. A comunicação entre estes nós é realizada através de uma rede sem fios, onde um nó transmite os valores do sensoriamento a outro nó próximo, que repassa estes dados para o próximo nó, e

assim por diante. Uma RSSF tente a ser autônoma e requer um alto grau de cooperação para executar as tarefas definidas para a rede. Isso significa que algoritmos distribuídos tradicionais, como protocolos de comunicação e eleição de líder, devem ser revistos para este tipo de ambiente antes de serem usados diretamente.

A ideia destas redes é tirar proveito de dispositivos pequenos e baratos que possam ser usados em larga escala. Deste modo, normalmente estas redes são formadas por um grande número de nós distribuídos, os quais possuem capacidade limitada de energia (LOUREIRO et al., 2003). Assim, estas redes podem apresentar problemas como falhas nas comunicações e perdas de nós. Como consequência das características destas redes, os protocolos de comunicação e gerenciamento desenvolvidos para as mesmas devem ter capacidades de auto-organização.

2.4.3 Redes entre Pares (*Peer-to-Peer*)

As redes entre pares (*Peer-to-Peer* - P2P) (STEINMETZ; WEHRLE, 2005) foram primeiramente projetadas e utilizadas para compartilhar (trocar) arquivos na Internet. No entanto, os mecanismos desenvolvidos para estas redes podem ser usados por qualquer aplicação que utiliza recursos distribuídos na Internet. Resumidamente, uma rede P2P é um sistema auto-organizável formado por nós interconectados através de uma rede, onde a gerência dos recursos é completamente descentralizada.

Para utilizar recursos compartilhados, as partes (*peers*) que compõe o sistema interagem diretamente entre si. Geralmente, esta interação é realizada sem o auxílio de um coordenador centralizado (STEINMETZ; WEHRLE, 2005). Esta é uma das principais propriedades que diferem um sistema P2P de um modelo cliente-servidor tradicional. Além disso, os participantes de um sistema P2P comumente exercem o papel tanto de cliente quanto de servidor, sendo que cada participante é completamente autônomo.

Idealmente, os recursos de sistemas P2P devem ser acessados sem o conhecimento prévio da localização dos mesmos, sendo que estes sistemas devem ser auto-organizáveis (STEINMETZ; WEHRLE, 2005). Esta característica pode ser violada por questões de desempenho, onde alguns elementos do sistema centralizariam algumas funções (STEINMETZ; WEHRLE, 2005). Mesmo assim, a natureza descentralizada destes sistemas deve ser mantida. Como resultado disto, surge algumas definições como sistemas P2P híbridos, os quais possuem alguns elementos centralizados.

No entanto, baseado em como os nós encontram os recursos disponíveis na rede, a comunidade P2P comumente divide estes sistemas em estrutu-

rados e não estruturados. Primeiramente, os sistemas não estruturados foram empregados no compartilhamento de arquivos. Estes sistemas podem utilizar um servidor para encontrar a localização dos dados desejados (abordagem híbrida) ou utilizar a técnica de inundação (*flooding*) entre os participantes do sistema. Já num sistema estruturado, cada participante mantém informações sobre os recursos disponíveis em seus vizinhos, necessitando um número menor de mensagens para acessar determinado dado (recurso). Esta abordagem utiliza DHTs (*Distributed Hash Tables*) para armazenamento e busca de dados. Estas tabelas, além de possibilitarem a indexação dos recursos, aumentam a escalabilidade do sistema bem como sua capacidade de tolerar faltas (STEINMETZ; WEHRLE, 2005). No entanto, como a topologia destas redes pode sofrer mudanças em qualquer momento (entrada ou saída de participantes), a manutenção destas informações pode se tornar custosa.

2.5 DESAFIOS NO DESENVOLVIMENTO DE APLICAÇÕES

Como observado na seção anterior, cada um dos exemplos de sistemas distribuídos dinâmicos citados possui características particulares e, portanto, diferentes desafios são encontrados no desenvolvimento de aplicações para cada um destes ambientes. Por exemplo, devido ao meio de comunicação empregado, uma aplicação projetada para uma ambiente de comunicação sem fio (MANET) provavelmente apresentará maiores problemas para trocar dados entre os processos do que uma aplicação desenvolvida no âmbito de um sistema P2P (cabeado).

No entanto, existem alguns desafios inerentes aos sistemas distribuídos dinâmicos que são comuns para qualquer um destes ambientes. Da mesma maneira que estes ambientes dinâmicos abrem caminho para novas classes de aplicações, os mesmos também criam vários desafios ligados à necessidade de assegurar que os algoritmos distribuídos mantenham suas propriedades, mesmo diante das variações que estão previstas para estes ambientes.

Os primeiros desafios encontrados no desenvolvimento de aplicações distribuídas, que executarão suas computações nestes ambientes, devem estar relacionados com o projeto e implementação de algoritmos que tratem dos problemas fundamentais em sistemas distribuídos, em particular os problemas de **acordo** e de **coordenação**. Algoritmos empregados na solução destes problemas são denominados *blocos básicos de construção* (*building blocks*), e são consideradas abstrações essenciais no projeto de aplicações distribuídas.

A seção seguinte apresenta um dos principais desafios, que não deve ser ignorado, no projeto de protocolos para sistemas distribuídos dinâmicos. Este desafio consiste em contornar os problemas relacionados com o *churn*,

que engloba precisamente a principal característica de um sistema distribuído dinâmico, i.e., a entrada e saída de nós do sistema em qualquer momento.

2.5.1 *Churn*

O termo *churn* foi introduzido para traduzir a variação no conjunto de participantes dos sistemas distribuídos dinâmicos, i.e., os eventos de entradas e saídas arbitrárias de nós no sistema ou ainda de falhas de processos. A ocorrência destes eventos gera perturbações no sistema, as quais podem resultar em (GODFREY et al., 2006): perda de mensagens, inconsistência de dados, aumento da latência observada pelo usuário, aumento no uso de largura de banda, interrupções parciais ou totais de serviços; indisponibilidade de recursos e variações bruscas no desempenho do sistema.

Deste modo, um desafio inerente aos sistemas distribuídos dinâmicos consiste em projetar e desenvolver arquiteturas auto-organizáveis, capazes de detectar e tratar as mudanças que podem ocorrer no grupo de entidades que compõem a aplicação distribuída, permitindo assim a reconfiguração da arquitetura em tempo de execução. Esta propriedade de auto-adaptação tem por objetivo assegurar que o sistema continuará atendendo, durante todo o seu ciclo de vida, aos requisitos de funcionamento da aplicação, evitando a deterioração da qualidade de serviço e, principalmente, da segurança.

A implementação destas arquiteturas auto-organizáveis, principalmente quando considerado um ambiente assíncrono e sujeito à presença de processos faltosos, deve envolver o uso de mecanismos que constantemente avaliem e forneçam dados sobre a composição atual do grupo de entidades que sustentam o sistema. Estes dados de composição são fundamentais no processamento da reconfiguração do sistema. No entanto, estes mecanismos devem operar com uma visão restrita (parcial) da composição do referido grupo, já que não é realista assumir em sistemas dinâmicos a possibilidade de se obter visões globais e consistentes do sistema.

Assim, o grande desafio que surge nestes ambientes é o de manter o progresso de aplicações distribuídas diante do *churn*. Os estudos relacionados ao *churn* buscam avaliar o seu impacto sobre sistemas distribuídos dinâmicos e apresentar alternativas para minimizar a sua ocorrência. Tentativas de minimizar a influência do *churn* envolvem o uso de heurísticas na seleção dos nós usados para compartilhar determinado recurso ou prover algum serviço. Estas heurísticas consistem em escolher nós que possuem maior probabilidade de permanecerem por mais tempo no sistema, servindo como fontes confiáveis de recursos ou serviços. A seleção dos nós pode ser realizada de maneira aleatória dentre o conjunto de nós que estão presentes no sistema ou através da

atribuição de pesos aos nós com base em observações realizadas no decorrer do tempo.

Em (GODFREY et al., 2006) as estratégias de seleção são divididas em dois eixos: *(1)* eixo relacionado com o uso ou não de informações sobre os nós, buscando identificar quais nós são mais estáveis, i.e., permanecerão no sistema por mais tempo; e (2) eixo relacionado com a substituição de nós faltosos, i.e., se os nós pertencentes ao conjunto escolhido serão ou não substituídos caso apresentarem falhas. No primeiro eixo as estratégias são classificadas como: *estabilidade prevista* - usam as informações sobre as características de cada nó; ou *estabilidade agnóstica* - não usam informações sobre os nós. Com relação ao segundo eixo, as estratégias são classificadas como: *fixas* - nós faltosos não são substituídos; ou *de substituição* - nós faltosos são substituídos por outros nós, caso tais nós estejam disponíveis no sistema. Assim, a combinação entre estes dois eixos gera as seguintes estratégias: prevista fixa, prevista com substituição, agnóstica fixa e agnóstica com substituição. Cada uma destas estratégias também pode ser subdividida em várias outras estratégias, dependendo de quais informações são utilizadas na seleção dos nós para compor o grupo inicial ou para substituir algum nó faltoso, se este for o caso.

Outro trabalho que considera o histórico dos nós para definir o conjunto escolhido como responsável por determinado serviço é apresentado em (STUTZBACH; REJAIE, 2006). Este trabalho indica que existe uma forte correlação entre os tempos que determinado nó permanece no sistema. Assim, é possível estimar de maneira aproximada o tempo em que um nó permanecerá no sistema com base em seus comportamentos prévios observados.

2.6 CONSIDERAÇÕES FINAIS

A partir do surgimento e da consolidação de tecnologias que viabilizam o desenvolvimento de sistemas distribuídos dinâmicos, como as descritas na Seção 2.4, aumentaram os estudos nesta área de pesquisa. Como não existe uma definição universalmente aceita para sistemas distribuídos dinâmicos, cada trabalho define e adota um modelo particular que muitas vezes não contempla todas as características de um sistema distribuído dinâmico. Na verdade, não existe um consenso sobre um modelo formal que contemple todos os aspectos de um sistema distribuído dinâmico. Alguns trabalhos visam definir tais modelos, como os apresentados na Seção 2.3, porém, estes modelos ainda não estão sendo completamente adotadas nos trabalhos nesta área.

Uma estratégia muito utilizada nos trabalhos sobre sistemas distribuídos dinâmicos (LYNCH; SHVARTSMAN, 2002; MARTIN; ALVISI, 2004;

RODRIGUES; LISKOV, 2004) é a de dividir o ciclo de vida do sistema em períodos (ou épocas), dentro dos quais o conjunto de participantes não sofre alterações e algumas premissas são assumidas para o correto funcionamento do sistema, como por exemplo, o número máximo de participantes que podem falhar ou sair do sistema. Para trocar de período, o sistema sofre uma reconfiguração, onde o conjunto de participantes é atualizado para computar as entradas e saídas de nós no sistema. Além disso, alguns parâmetros também são reconfigurados. Estes parâmetros podem variar de aplicação para aplicação, mas alguns deles são comuns para a maioria das aplicações. Este é o caso do número máximo de processos que podem falhar ou deixar o sistema (o parâmetro f). No novo período, o limite f pode ser reconfigurado em um valor diferente daquele assumido no período anterior. Esta estratégia emprega o modelo de chegadas infinitas M_2^{finito} de participantes (Tabela 1), onde a concorrência é desconhecida mas limitada.

Outros trabalhos consideram um sistema formado por uma rede desconhecida (CAVIN et al., 2004, 2005; GREVE; TIXEUIL, 2007), onde inicialmente os processos não possuem conhecimento algum sobre a composição do sistema. No entanto, antes de realizar qualquer computação cada processo obtém uma visão parcial sobre a composição do sistema, que representa o conjunto de entidade com quem tal processo pode colaborar. Estas visões são obtidas no início de cada execução, definindo o modelo de chegada finita M_1 (Tabela 1), pois novas chegadas apenas serão consideradas nas execuções seguintes quando as visões parciais serão redefinidas.

Este capítulo apresentou alguns conceitos relacionados com os sistemas distribuídos dinâmicos. Estes conceitos são fundamentais para entender como estes sistemas funcionam e observar as diferenças existentes entre estes ambientes e os clássicos sistemas distribuídos estáticos. Para facilitar a compreensão, foram apresentados alguns exemplos de sistemas distribuídos dinâmicos, os quais também são úteis para entender quais são as tecnologias que estão relacionadas com estes sistemas.

3 SEGURANÇA DE FUNCIONAMENTO EM SISTEMAS DISTRIBUÍDOS DINÂMICOS

Este capítulo apresenta os principais conceitos sobre segurança de funcionamento em sistemas distribuídos dinâmicos (SDD). As principais vulnerabilidades a que estão sujeitas as aplicações distribuídas nestes ambientes também são descritas. Além disso, os problemas fundamentais, que formam a base para o desenvolvimento de aplicações distribuídas com atributos de segurança de funcionamento, são também objeto deste capítulo. Alguns trabalhos que propõem soluções para estes problemas em sistemas distribuídos dinâmicos são apresentados, evidenciando-se suas principais características.

3.1 INTRODUÇÃO

Para que um sistema seja seguro e confiável é necessário que o mesmo apresente propriedades de segurança de funcionamento (*dependability*) (AVIZIENIS et al., 2004). Um sistema com segurança de funcionamento possui um comportamento que evolui segundo suas especificações, mesmo diante da ocorrência de falhas, ataques e intrusões no sistema. A importância de um sistema ser seguro e confiável reside no fato de que, por exemplo, um sistema desprotegido pode ser completamente comprometido por um ataque bem sucedido resultando então numa intrusão neste sistema. Neste caso, o invasor poderia obter o completo controle do sistema.

Vários problemas comuns são encontrados no desenvolvimento de aplicações distribuídas que apresentam requisitos de segurança de funcionamento. Estes problemas foram definidos como blocos básicos de construção (*building blocks*), justamente por se tratarem de peças fundamentais e comuns no desenvolvimento de aplicações distribuídas seguras e confiáveis. Assim, a construção destas aplicações envolve a utilização de protocolos desenvolvidos para solucionar estes problemas.

A complexidade da solução destes problemas é variada e depende de uma série de fatores, geralmente definidos no modelo de sistema ao qual cada protocolo é endereçado. No entanto, esta complexidade aumenta na medida em que se consida um sistema distribuído dinâmico, que adiciona uma série de incertezas que devem ser suportadas por estes protocolos básicos. Em vista disso, a quantidade de soluções para estes problemas em ambientes estáticos é muito superior às encontradas para ambientes dinâmicos. No entanto, é imprescindível a utilização de protocolos que tratem destes problemas em ambientes dinâmicos, para que aplicações atendam aos requisitos de segurança

de funcionamento também nestes ambientes.

Este capítulo primeiramente descreve algumas vulnerabilidades inerentes aos sistemas distribuídos dinâmicos, para então discutir as propriedades que devem ser garantidas para um sistema (ou protocolo) apresentar segurança de funcionamento. Após isso, apresenta alguns conceitos sobre tolerância a faltas e intrusões, cujas técnicas baseadas em replicação formam a sustentação das propriedades de segurança de funcionamento. Além disso, este capítulo discute os principais problemas fundamentais encontrados no desenvolvimento de aplicações distribuídas. Estes problemas são abordados considerando as especificações e conceitos próprios e, também, o contexto de ambiente dinâmico considerado. As principais dificuldades encontradas na solução de cada problema nestes ambientes são apresentadas, onde são discutidas as principais soluções propostas para os mesmos.

3.2 VULNERABILIDADES EM SDD

Algumas características dos sistemas distribuídos dinâmicos os tornam vulneráveis a problemas de segurança em diversos aspectos. Estas vulnerabilidades estão principalmente relacionadas com suas composições e topologias de rede: entradas e saídas aleatórias de participantes no sistema tornam suas composições e topologias variáveis no tempo. Desta forma, não é possível fixar nós como responsáveis por determinadas tarefas durante todo o ciclo de vida do sistema. Assim, uma mesma tarefa pode ser realizada por diversos conjuntos de processos durante o ciclo de vida do sistema, sem mesmo que seja necessário que ocorram intersecções entre estes conjuntos.

A mobilidade dos nós, além da entrada e saída dos mesmos, causa também constantes mudanças na topologia da rede, o que introduz um elemento de incerteza, impondo que as soluções de confiabilidade e segurança se adaptem às novas composições que se formam no sistema.

Além disso, quando consideramos as tecnologias relacionadas com ambientes dinâmicos, outras vulnerabilidades particulares são encontradas. Por exemplo, em uma rede *ad hoc* pode-se encontrar os seguintes pontos a serem explorados por uma parte maliciosa (atacante): *(i)* necessidade de cooperação entre os nós na realização de determinada tarefa (ex.: roteamento); *(ii)* canais de comunicação *wireless*, usados nestas redes, não apresentam qualquer proteção aos dados transmitidos; e *(iii)* restrições de capacidade computacional e de energia inerentes aos nós da rede.

A necessidade de cooperação no roteamento de pacotes (roteamento *overlay* de pacotes) é maior em redes *ad hoc* multi-salto, onde a comunicação entre nós separados por uma distância superior a um salto ocorre através de

nós intermediários. O envio de pacotes de dados deve ser precedido pela operação de descoberta de rota, na qual será determinado o caminho através do qual pacotes de dados, para alcançar o destino, passam por nós intermediários no processo de encaminhamento de dados. Nós maliciosos podem estar envolvidos nestas operações (descoberta de rotas e encaminhamento de pacotes), colocando em risco o correto funcionamento das mesmas.

Em relação à camada física da rede, os canais de comunicação *wireless* são abertos e acessíveis tanto para usuários legítimos quanto para atacantes, não oferecendo qualquer resistência à interceptação ou introdução de informações. Com base nesta característica, ataques de negação de serviço podem ser conduzidos a partir da injeção de pacotes espúrios na rede, visando consumir recursos de rede como largura de banda ou recursos dos nós como memória e energia.

Além disso, comumente os dispositivos *wireless* possuem recursos restritos de: energia, capacidade de processamento, comunicação e armazenamento de dados. Estas restrições limitam, por exemplo, a adoção de mecanismos criptográficos de alto custo computacional.

Um dos maiores desafios de segurança encontrados no desenvolvimento de soluções para sistemas distribuídos dinâmicos refere-se a um ataque no qual determinado participante malicioso tenta entrar no sistema usando identidades falsas, podendo inclusive se fazer passar por vários nós do sistema. Esta ação maliciosa, discutida na próxima seção, é chamada de ataque *Sybil*.

3.2.1 Ataque *Sybil*

Um ataque *Sybil* (DOUCEUR, 2002) consiste na obtenção de múltiplas identidades por uma entidade, que torna-se capaz de comprometer as propriedades do sistema. Para o sistema, cada uma das identidades obtidas por esta entidade maliciosa é tratada como uma nova entidade.

Sistemas distribuídos geralmente utilizam redundância para prover garantias relacionadas com tolerância a faltas de processos do sistema. No entanto, caso uma entidade maliciosa consiga apresentar múltiplas identidades, esta entidade pode controlar uma grande parte dos recursos e serviços do sistema com o objetivo de ferir as propriedades de segurança do mesmo.

Uma forma simples de tratar este problema é através da utilização de uma autoridade centralizada e confiável responsável pela distribuição das identidades dos participantes do sistema (DOUCEUR, 2002). Outras soluções sugerem a presença de uma barreira na entrada do sistema, i.e., que a obtenção de novos identificadores seja dificultada (DINGER; HARTENS-

TEIN, 2006). No entanto, a dificuldade para entrar no sistema pode desencorajar ou desestimular o uso do mesmo por parte de novos participantes.

Dentre vários outros trabalhos, ainda existem soluções que não utilizam uma parte do sistema centralizada, como em (VORA et al., 2008) que apresenta um algoritmo para resolver este problema em MANETs, assumindo algumas premissas relacionadas com a densidade mínima dos nós (distribuição dos nós) bem como com o alcance mínimo de transmissão destes nós. Ainda neste trabalho, é apresentada a ideia de *detectores de universo*, que indicam qual universo é real, i.e., não apresenta nós fictícios. Esta abstração de detectores de universo é inspirada e definida da mesma forma dos, já conhecidos, detectores de falhas (CHANDRA; TOUEG, 1996).

3.3 SEGURANÇA DE FUNCIONAMENTO

Segurança de funcionamento é uma característica fundamental dos sistemas computacionais (AVIZIENIS et al., 2004). Para um sistema ser seguro de funcionamento é necessário que o mesmo suporte as seguintes propriedades (AVIZIENIS et al., 2004):

- **Confiabilidade** (*reliability*): as operações realizadas no sistema fazem com que seu estado se modifique de acordo com sua especificação;
- **Disponibilidade** (*availability*): o sistema sempre está pronto para executar as operações requisitadas por partes autorizadas;
- **Integridade** (*integrity*): nenhuma alteração imprópria no estado do sistema pode ocorrer, i.e., o estado do sistema só pode ser alterado através da correta execução de suas operações;
- **Confidencialidade** (*confidentiality*): dados confidenciais não podem ser revelados a partes não autorizadas.

Como destacado em (AVIZIENIS et al., 2004), nem sempre é requerido que um sistema contemple todos estes atributos, pois isto depende das necessidades da aplicação suportada. Em nossos estudos nos preocuparemos principalmente com os três primeiros atributos listados: confiabilidade, disponibilidade e integridade.

Vários mecanismos e técnicas foram desenvolvidos para que as aplicações pudessem prover estas propriedades. Estes mecanismos são normalmente divididos em quatro grupos (AVIZIENIS et al., 2004):

- **Prevenção de Faltas**: mecanismos ou técnicas que visam prevenir a ocorrência de faltas;

- **Tolerância a Faltas**: através da inclusão de redundâncias, visa evitar que um serviço falhe na presença de componentes faltosos;
- **Remoção de Faltas**: técnicas que procuram reduzir o número e a severidade das faltas;
- **Previsão de Faltas**: métodos para estimar o número presente e futuro de faltas, bem como a provável consequência das mesmas (semântica das falhas).

Prevenção e tolerância a faltas envolvem técnicas e mecanismos que têm como objetivo fornecer por construção um serviço que seja seguro. A remoção e previsão de faltas tentam atender os atributos de segurança de funcionamento pela justificativa de que as especificações são adequadas para o sistema e que o mesmo provavelmente atenderá estas especificações. O presente trabalho se concentra em mecanismos de tolerância a faltas para o fornecimento da segurança de funcionamento.

3.3.1 Tolerância a Faltas e Intrusões

A crescente dependência por serviços confiáveis, que operem sem interrupções, demanda sistemas com alta disponibilidade e que forneçam serviços de forma correta mesmo na presença de faltas e intrusões. Estes sistemas, capazes de permanecerem corretos mesmo em circunstâncias adversas, são ditos sistemas tolerantes a faltas e intrusões (VERíSSIMO et al., 2003; FRAGA; POWELL, 1985).

Como já comentado, tolerância a faltas é um dos mecanismos usados para prover segurança de funcionamento. Tolerar faltas sempre envolve a utilização de replicação do serviço fornecido, i.e., o sistema deve ser replicado em um conjunto de servidores, sendo que falhas que possam ocorrem em alguns deles é mascarada pelos demais servidores corretos. Desta forma, o sistema como um todo permanece correto mesmo que uma parte de seus componentes falhe (alguns processos que compõem o sistema replicado), por isso o sistema é dito tolerante a faltas.

Quando consideramos um ambiente sujeito a faltas Bizantinas (LAMPORT et al., 1982), o sistema deve ser tolerante a intrusões, i.e., o sistema deve continuar correto mesmo que uma parte dos processos que o compõem exiba qualquer comportamento malicioso. Este comportamento malicioso que algum servidor (processo) pode apresentar geralmente está relacionado com a invasão e corrupção deste servidor. Sistemas que devem conviver com elementos corrompidos (com comportamentos maliciosos) e mesmo assim manter a qualidade de seus serviços, são ditos sistemas tolerantes a intrusões.

Técnicas de replicação também são usadas para tornar um sistema tolerante a intrusões.

A seção seguinte descreve os principais problemas clássicos de sistemas distribuídos, considerando os mesmos em ambientes dinâmicos e analisando os mesmos sob a ótica da tolerância a faltas e a intrusões.

3.4 PROBLEMAS DE SEGURANÇA DE FUNCIONAMENTO

Esta seção apresenta os principais problemas clássicos encontrados no desenvolvimento de aplicações distribuídas. Algumas propostas de soluções para estes problemas em ambientes dinâmicos são discutidas na sequência.

3.4.1 Filiação

Um aspecto fundamental em sistemas distribuídos é o uso de técnicas de replicação de tal forma que operações possam ser executadas com sucesso, mesmo em caso de falhas de alguns dos componentes replicados no sistema. O uso de replicações visa manter as propriedades dos serviços providos (ver Seção 3.3) mesmo quando da ocorrência de faltas. O principal desafio envolvido na implementação destas replicações é suportar o conceito de **grupo** (réplicas de serviços ou de objetos) e gerenciar falhas e intrusões nestes grupos mantendo a integridade dos estados de seus membros de acordo com requisitos de segurança.

Neste sentido, o problema de **filiação**, também conhecido como *membership*, pode ser definido como um acordo realizado entre os processos de um grupo para saber, em tempo de execução, quem pertence e quem não pertence ao grupo, determinando assim a composição do mesmo (CRISTIAN, 1988, 1991). A lista de processos pertencentes a um grupo, chamada **visão**, pode ser estática ou dinâmica. Em um grupo estático os membros são conhecidos previamente, no início da computação. Em grupos dinâmicos, processos entram e saem dos grupos, o quê representa uma pertinência que evolui, possuindo um número de membros variável em função do tempo.

O **serviço de pertinência** é o componente do sistema distribuído responsável por manter a coordenação entre todos os processos a respeito de quem pertence ao grupo em um dado instante. Este serviço processa as mudanças em um grupo e altera a lista de membros do mesmo, a partir da execução de operações oferecidas aos processos, como por exemplo *join* (entrada) e *leave* (saída), e da ocorrência de falhas de processos, mantendo assim um *membership* consistente.

Formalmente o problema de filiação é definido por três primitivas:

- *join*(G,p): inclui o processo p no grupo G;
- *leave*(G,p): remove o processo p do grupo G.
- *view*(G,V^i): chamada pelo serviço de pertinência para enviar uma nova visão V^i para os processos do grupo G. Dizemos que o processo $p \in G$ que recebe V^i **instala** a visão V^i.

Estas primitivas devem atender as seguintes propriedades (RICCIARDI; BIRMAN, 1991; GUERRAOUI; RODRIGUES, 2003):

- **Própria inclusão:** Se um processo p instala a visão V^i então $p \in V^i$;
- **Monotonicidade local:** Se um processo p instala V^j depois de instalar V^i então $j > i$;
- **Visão inicial:** Todos os processos corretos instalam a visão $V^0 = G$ (configuração inicial);
- **Acordo:** Se um processo correto instala V^i então todos os processos corretos acabam por instalar V^i;
- **Completude:** Se um processo p é faltoso, então uma visão V^i instalada por algum processo correto terminará por não conter p, i.e. $p \notin V^i$;
- **Precisão:** Se um processo q instala uma visão V^i e sendo p um processo tal que $p \notin V^i$ então p é faltoso.

Estas propriedades dizem respeito a serviços de **pertinência não particionável** onde todos os processos corretos concordam com uma mesma visão do sistema, ao contrário dos sistemas com serviço de **pertinência particionável** onde admite-se várias visões diferentes do grupo, tolerando-se inclusive partições na rede, onde define-se operações de junção (*merge*) a serem usadas quando os casos de particionamento forem tratados, regenerando o grupo original (KEIDAR et al., 2002). A grande diferença, em termos de especificação do problema, diz respeito a inclusão de uma nova operação *merge*(G,V^i,V^j) para a junção de partições distintas e a modificação da propriedade de Acordo (uma vez que agora nem todos os processos instalam a mesma visão) e Precisão (nestas condições de sistema nem todo processo não pertencente a uma visão pode ser considerado faltoso).

Em termos de arquitetura, um serviço de pertinência pode ser distribuído ou centralizado. A abordagem distribuída emprega protocolos de filiação de tal forma que os próprios participantes da computação distribuída

gerenciam as visões do sistema. Já a solução centralizada utiliza um gerenciador de filiação para prover este serviço. Assim, não é necessário um protocolo distribuído de filiação para as decisões sobre os participantes dos grupos. No entanto, a simplicidade nas decisões deste gerenciador tem como custo a sua vulnerabilidade como ponto único de falha na arquitetura. Desta forma, o próprio gerenciador deve ser replicado para tolerar faltas.

3.4.1.1 Filiação em SDD

Os protocolos projetados para sistemas distribuídos dinâmicos trabalham com serviços de pertinência (filiação) dinâmica. Nestes sistemas, o problema de filiação toma dimensões maiores visto que a composição do sistema pode ser, a priori, desconhecida e que cada nó conhece apenas uma parcela do total de nós do sistema. Desta forma, não é possível determinar de maneira descentralizada um estado global contendo todos os nós presentes no sistema em um dado momento, o que implica que os nós devem trabalhar com visões locais parciais do sistema.

Seguindo esta abordagem, os trabalhos de (CAVIN et al., 2004, 2005; GREVE; TIXEUIL, 2007) utilizam *detectores de participação* para determinar quais são as entidades presentes no sistema em *redes desconhecidas*, i.e., sistemas distribuídos dinâmicos onde os participantes são, a priori, desconhecidos. Através destes detectores, cada participante do sistema obtêm um conjunto de entidades com as quais pode colaborar na determinação de uma visão parcial do sistema. Nos protocolos apresentados nestes trabalhos, não é necessário que a visão parcial obtida seja a mesma em todos os participantes do sistema. Desta forma, esta abordagem não satisfaz as propriedades de Acordo, Completude e Precisão (Seção 3.4.1).

Outra abordagem muito empregada em ambientes dinâmicos é a utilização de um serviço de filiação responsável por fornecer a visão global do sistema para os participantes do mesmo (ex.: (RODRIGUES et al., 2002; RODRIGUES; LISKOV, 2004; MARTIN; ALVISI, 2004; MARTIN; ALVISI, 2004)). Nestes trabalhos, o ciclo do vida do sistema é dividido em períodos (ou épocas), dentro dos quais é assumido que o conjunto de participantes não sofre alterações. No entanto, quando um período chega ao fim, o serviço de filiação deve reconfigurar o sistema através da atualização do conjunto de participantes do mesmo. Neste procedimento, as entradas e saídas de nós, bem como as falhas, são computadas por este serviço. Note que, para um participante entrar no sistema, é necessário que o mesmo conheça um determinado número de nós responsáveis pelo serviço de filiação no período corrente. Assim, este participante poderá executar a operação de entrada no sistema (*join*)

nestes nós que o incluirão na próxima visão do sistema.

3.4.2 Consenso

Em um sistema distribuído, formado por vários processos independentes, o *problema do consenso* consiste em fazer com que todos os processos corretos acabem por decidir o mesmo valor, o qual deve ter sido previamente proposto por algum dos processos do sistema. Formalmente, este problema é definido em termos de duas primitivas (HADZILACOS; TOUEG, 1994):

- *propose(G,v)*: o valor *v* é proposto ao conjunto de processos *G*.
- *decide(v)*: executado pelo protocolo de consenso para notificar ao(s) interessado(s) (geralmente alguma aplicação) que *v* é o valor decidido.

Para estas primitivas satisfazerem a correção (*safety*) e a vivacidade (*liveness*), as mesmas devem verificar as seguintes propriedades (ASPNES, 2003; ATTIYA; WELCH, 2004):

- **Acordo**: Se um processo correto decide *v*, então todos os processos corretos terminam por decidir *v*.
- **Validade**: Um processo correto decide *v* somente se *v* foi previamente proposto por algum processo.
- **Terminação**: Todos os processos corretos terminam por decidir.

A propriedade de acordo garante que todos os processos corretos decidem o mesmo valor. A validade relaciona o valor decidido com os valores propostos e sua alteração dá origem a outros tipos de consensos. As propriedades de acordo e validade definem os requisitos de correção (*safety*) do consenso, já a propriedade de terminação define o requisito de vivacidade (*liveness*) deste problema.

Uma variante deste problema é o *consenso uniforme* (CHARRON-BOST; SCHIPER, 2000), onde todos os processos (sendo eles corretos ou não) precisam decidir o mesmo valor. No entanto, consenso uniforme não faz muito sentido em sistemas bizantinos, pois não se pode exigir qualquer tipo de comportamento de processos faltosos bizantinos.

Existem muitos trabalhos sobre consenso, principalmente por este problema ser equivalente ao problema de difusão atômica (Seção 3.4.3), que é a base para a replicação Máquina de Estados (Seção 3.4.5), que, por sua vez, é a técnica de replicação mais usada na implementação de sistemas distribuídos

tolerantes a faltas e intrusões. No entanto, grande parte destes trabalhos se concentra na solução do consenso em sistemas distribuídos estáticos, onde o conjunto de participantes do sistema não sofre alterações. Para estes sistemas estáticos, existem protocolos para resolver o consenso considerando-se tanto faltas apenas por parada (*crash*) (LAMPORT, 1998; LAMPSON, 2001; LAMPORT, 2001) quanto faltas bizantinas (LAMPORT et al., 1982; DOLEV, 1982; CASTRO; LISKOV, 2002; MARTIN; ALVISI, 2006).

Um dos trabalhos mais importantes envolvendo o problema do consenso é (FISCHER et al., 1985), onde é provada a impossibilidade de se resolver este problema em um sistema distribuído completamente assíncrono onde pelo menos um processo pode ser faltoso (qualquer tipo de falta). Em vista disso, a maioria das propostas encontradas na literatura consideram algum comportamento temporal do sistema, onde componentes (processos e/ou canais) obedecem a requisitos temporais. Estes requisitos geralmente estão relacionados com *timeouts* encapsulados em *detectores de falhas* (CHANDRA; TOUEG, 1996) ou são atendidos a partir da ocorrência de possíveis comportamentos síncronos do sistema subjacente.

3.4.2.1 Consenso em SDD

Contrastando com a literatura sobre sistemas distribuídos estáticos, poucos trabalhos envolvendo consenso em sistemas distribuídos dinâmicos já foram desenvolvidos. Isto se deve principalmente ao pouco tempo de consolidação destes sistemas e a falta de uma padronização para o mesmo (ver capítulo anterior). As soluções para o problema do consenso em ambientes dinâmicos podem ser divididas em duas categorias: (*1*) soluções que utilizam uma parte do sistema fixa (BADACHE et al., 1999; SEBA et al., 2002; CAO et al., 2007) e (*2*) soluções sem componentes fixos (CAVIN et al., 2004, 2005; GREVE; TIXEUIL, 2007).

Nesta primeira linha de trabalhos (BADACHE et al., 1999; SEBA et al., 2002; CAO et al., 2007) encontram-se soluções para o consenso em ambientes móveis (redes móveis), onde as entidades que compõem o sistema apenas podem falhar por parada (*crash*) e são divididas em dois grupos: estações móveis (*mobile hosts* - MH) e estações fixas, que atuam dando suporte às estações móveis e são chamados de estações base (*base stations* ou *mobile support stations* - MSSs). As estações base são conectadas entre si e formam um sistema distribuído estático. Cada estação base cobre uma determinada região (célula) onde se localizam as estações móveis, que formam a parte dinâmica do sistema. Para diminuir as necessidades de comunicação e processamento nas estações móveis, o consenso apenas é executado pelas estações

fixas (sistema distribuído estático). Assim, primeiramente as estações móveis enviam os valores a serem propostos para as estações base (cada estação base recebe valores apenas das estações móveis localizadas dentro da sua área de cobertura). Após isso, as estações base executam um consenso usando, como valor de proposição, estes valores recolhidos das estações móveis e enviam a decisão para estas estações. Nesta solução, a propriedade da validade é modificada para permitir que a decisão seja um conjunto de valores (propostos pelas estações móveis) e não apenas um único valor.

Já na segunda abordagem para consenso em ambientes dinâmicos, este problema é estudado em redes desconhecidas, i.e., redes onde os processos, a priori, não possuem nenhuma informação sobre a composição do sistema. Os trabalhos nesta área (CAVIN et al., 2004, 2005; GREVE; TIXEUIL, 2007) buscam determinar as condições necessárias e suficientes para resolver o consenso no modelo de sistema considerado (redes desconhecidas). Estas condições podem ser divididas em: condições relacionadas com o sincronismo da rede; e condições de conectividade entre os participantes do sistema, encapsuladas na abstração de detectores de participação.

Um dos aspectos mais interessante destes trabalhos é que, durante a execução dos protocolos para resolver o consenso, os participantes presentes em determinada computação obtêm uma visão parcial sobre a composição do sistema que não é necessariamente a mesma para todos os participantes envolvidos neste processamento. Sendo assim, estas soluções seguem uma abordagem modular: primeiramente algoritmos identificam um subconjunto de participantes que compartilham a mesma visão parcial do sistema, para então executar qualquer algoritmo de consenso para sistemas distribuídos estáticos entre estes participantes. Após a decisão ser obtida, a mesma é encaminhada para o restante do sistema.

Inicialmente, (CAVIN et al., 2004) propuseram uma solução para o problema do consenso em redes desconhecidas assíncronas, desconsiderando-se a possibilidade de falhas de processos. Neste problema, chamado *consensus with unknown participants* (CUP), o principal desafio é a falta prévia de conhecimento sobre a composição do sistema. Como é impossível resolver o consenso caso cada processo tenha conhecimento apenas sobre si próprio, os processos obtêm informações sobre os demais participantes do sistema através da noção de detectores de participação (CAVIN et al., 2004), os quais são oráculos distribuídos que fornecem "dicas" aos processos sobre quais destes estão presentes na computação distribuída. A partir destas informações, é possível realizar uma pesquisa onde os participantes obtêm o máximo possível de conhecimento sobre a composição do sistema, gerando um grafo de conhecimento. De acordo com as características observadas neste grafo, são definidas algumas classes de detectores de participação.

No trabalho seguinte, *Cavin et all.* (CAVIN et al., 2005) estendem o modelo anterior adicionando a possibilidade de falhas por parada de participantes do sistema. Neste problema, chamado *Fault-Tolerant Consensus with Unknown Participants* - FT-CUP, encontramos dois desafios principais: a falta de conhecimento sobre a composição do sistema e a possibilidade de falhas por parada de participantes do sistema. Neste trabalho é provado que utilizando o mínimo de conectividade requerida para resolver o FT-CUP é necessário enriquecer o sistema com um detector de falhas perfeito ($\mathscr{P}$) (CHANDRA; TOUEG, 1996), o que representa a adição necessária de um nível elevado de sincronia, pois estes detectores possuem propriedades como (Seção 3.4.6.1): completude forte - todo processo faltoso acabará por ser suspeito, para sempre, por todos os processos corretos; e exatidão forte - nenhum processo é suspeito antes de falhar.

Um estudo comparando as condições de conectividade do grafo de conhecimento (necessidades de conhecimento sobre a composição do sistema) e as condições de sincronia do sistema é apresentado em (GREVE; TIXEUIL, 2007), onde é estabelecido que o FT-CUP admite solução em redes assíncronas enriquecidas com o mais fraco detector de falhas capaz de resolver o consenso ($\diamondsuit\mathscr{S}$ (CHANDRA; TOUEG, 1996)), desde que se aumente o grau de conhecimento entre os participantes. Desta forma, é provado que existe uma relação entre o nível de sincronia fornecido pelo sistema e o grau de conhecimento adquirido pelos participantes, i.e., aumentando-se o conhecimento sobre a composição do sistema o FT-CUP admite solução em sistemas com um nível menor de sincronia.

Como podemos perceber, todos os trabalhos sobre consenso em ambientes onde o conjunto de participantes pode sofrer alterações, tanto as soluções que utilizam partes fixas no sistema quanto as soluções completamente descentralizadas, apenas consideram a possibilidade de faltas por parada. No entanto, dada a natureza destes ambientes (ambiente aberto), é grande a probabilidade de entidades maliciosas estarem presentes no sistema.

3.4.3 Difusão Atômica

O problema da *difusão atômica* (HADZILACOS; TOUEG, 1994), conhecido também como *difusão com ordem total*, consiste em fazer com que todos os processos corretos, membros de um grupo, entreguem todas as mensagens difundidas neste grupo na mesma ordem.

A difusão atômica é definida sobre duas primitivas básicas (HADZILACOS; TOUEG, 1994):

- A-*multicast(G,m)* ou TO-*multicast(G,m)*: primitiva utilizada para di-

fundir a mensagem *m* no grupo *G*.

- A-*deliver(p,m)* ou TO-*deliver(p,m)*: chamada pelo protocolo de difusão atômica para entregar à aplicação a mensagem *m*, difundida pelo processo *p*.

Formalmente, um protocolo de difusão atômica deve satisfazer as seguintes propriedades (HADZILACOS; TOUEG, 1994):

- **Validade:** Se um processo correto difundiu *m* no grupo *G*, então algum processo correto pertencente à *G* terminará por entregar *m* ou nenhum processo pertencente à *G* está correto;
- **Acordo:** Se um processo correto pertencente a um grupo *G* entregar a mensagem *m*, então todos os processos corretos pertencentes a *G* acabarão por entregar *m*;
- **Integridade:** Para qualquer mensagem *m*, cada processo correto pertencente ao grupo *G* entrega *m* no máximo uma vez e somente se *m* foi previamente difundida em *G*;
- **Ordem Total Local:** Se dois processos corretos *p* e *q* entregam as mensagens *m* e m' difundidas no grupo *G*, então *p* entrega *m* antes de m' se e somente se *q* entregar *m* antes de m'.

Quando consideramos que os processos estão sujeitos a faltas Bizantinas, a propriedade de integridade precisa ser reformulada (HADZILACOS; TOUEG, 1994):

- **Integridade:** Para qualquer mensagem *m*, cada processo correto pertencente ao grupo *G* entrega *m* no máximo uma vez e somente se o emissor de *m* (*sender(m)*) é correto, então *m* foi previamente difundida em *G*.

Com esta definição, a propriedade de integridade se refere apenas a difusão e entrega de mensagens por processos corretos[1]. Porém, é difícil diferenciar um processo correto de um processo faltoso que está se comportando corretamente em relação ao protocolo. As outras propriedades não precisam sofrer alterações por estarem relacionadas apenas com processos corretos.

Um resultado teórico interessante é que a difusão atômica e o consenso são problemas equivalentes em sistemas onde os processos então sujeitos tanto a falhas de parada (CHANDRA; TOUEG, 1996) quanto a falhas bizantinas (CORREIA et al., 2006).

[1] O problema com a definição normal de integridade está em determinar se o processo que se diz emissor de *m* (*sender(m)*) realmente é este emissor.

3.4.3.1 Difusão Atômica em SDD

Não existem muitos trabalhos específicos sobre difusão atômica em ambientes dinâmicos. No entanto, como difusão atômica e consenso são problemas equivalentes (CHANDRA; TOUEG, 1996), podemos dizer que os trabalhos sobre consenso, apresentados na seção anterior, também estão relacionados com difusão atômica.

Além destes trabalhos, *Joseph et all.* (BAR-JOSEPH et al., 2002) propuseram um protocolo para difusão atômica em ambientes dinâmicos, considerando apenas a possibilidade de falhas por parada de processos. Este protocolo funciona em uma variante do modelo de sistema síncrono, onde os processos possuem relógios sincronizados (aproximadamente). Assim, cada processo divide o tempo em *slots*, usando seu relógio local, e relaciona cada mensagem a ser enviada com um destes *slots*. Os processos entregam as mensagens na ordem dos *slots*, e dentro de cada *slot* na ordem do identificador do emissor. Para cada *slot*, é determinado o *membership* deste *slot* e somente são entregues as mensagem de membros do *slot* em questão. Para garantir consistência, os processos precisam concordar com o *membership* de cada *slot*, o que é realizado através de um protocolo de consenso.

3.4.4 Memória Compartilhada

Algumas operações realizadas em sistemas distribuídos podem ser coordenadas através de alterações em dados compartilhados. Para garantir a disponibilidade destes dados utiliza-se replicação. O grande desafio é manter a consistência destes dados compartilhados, a fim de assegurar a correta coordenação nas operações realizadas no sistema (GIFFORD, 1979; BESSANI et al., 2008).

Uma abstração que pode ser utilizada como memória compartilhada são os *espaços de tuplas* (GELERNTER, 1985). Um espaço de tuplas pode ser visto (conceitualmente) como um objeto de memória compartilhada que fornece operações para armazenar e recuperar conjuntos de dados ordenados chamados de tuplas. Sendo assim, os processos de um sistema distribuído podem interagir através desta abstração de memória compartilhada. No entanto, poucos trabalhos foram desenvolvidos com o objetivo de fornecer um espaço de tuplas dinâmico. O *middleware* LIME (MURPHY et al., 2001) e sua extensão para redes de sensores, chamada TINYLIME (CURINO et al., 2005), são gratas exceções.

Por outro lado, a maneira mais comum de prover memória compartilhada em sistemas distribuídos é através do emprego de sistemas de quóruns.

Como sistemas de quóruns são objetos de estudo deste trabalho, os mesmos são detalhados a seguir.

3.4.4.1 Sistemas de Quóruns

Sistemas de quóruns (GIFFORD, 1979) são ferramentas usadas para garantir consistência e disponibilidade de dados, que são armazenados de forma replicada em um conjunto de servidores. O grande atrativo destes sistemas está relacionado com seu poder de escalabilidade e balanceamento de carga, uma vez que as operações não precisam ser executadas por todos os servidores que compõem o sistema, mas apenas por um quórum dos mesmos. A consistência de um sistema de quóruns é assegurada pela propriedade de intersecção dos quóruns do mesmo. Estes sistemas concretizam um registrador que permite operações de leitura e escrita com diferentes semânticas de implementação (LAMPORT, 1986): segura, regular ou atômica.

- Semântica segura: quando não houver uma escrita concorrente, garante que uma operação de leitura retorna o último valor escrito no registrador; caso contrário, quando houver escrita concorrente com leitura, a operação de leitura pode retornar qualquer valor do domínio de valores do registrador.

- Semântica regular: garante a semânctica segura e, quando houver escritas concorrentes com uma leitura, o valor retornado pela operação de leitura pode ser o último valor escrito no sistema ou um dos valores que estão sendo escritos.

- Semântica atômica: garante a semântica regular e uma ordenação das operações de leitura e escrita seguindo a relação *happens before* (LAMPORT, 1978). Deste modo, dado duas operações de leitura l_1 e l_2 executadas por clientes corretos, sendo que l_1 completa antes de l_2 iniciar, garante que caso l_1 leia um valor v do registro, então l_2 também lê v ou um valor mais atual escrito neste registrador.

Servidores em um sistema de quóruns organizam-se em subconjuntos denominados **quóruns**. Cada dois quóruns de um sistema mantêm um número suficiente de servidores corretos em comum (garantia de consistência), sendo que existe pelo menos um quórum no sistema formado somente por servidores corretos (garantia de disponibilidade) (MALKHI; REITER, 1998a). Os clientes realizam operações de leitura e escrita em registradores replicados por estes quóruns, cujas dimensões para operações de leitura e escrita podem ser iguais (quóruns simétricos) ou não (quóruns assimétricos).

O conceito de sistemas de quóruns foi inicialmente estudado em um modelo estático onde os servidores apenas falham por parada (GIFFORD, 1979). Posteriormente, o modelo foi estendido para tolerar comportamento malicioso dos servidores (MALKHI; REITER, 1998a). No entanto, o grande desafio em sistemas de quóruns é desenvolver protocolos eficientes para tolerar comportamento malicioso por parte dos clientes, pois os mesmos podem executar uma série de ações maliciosas com o objetivo de ferir as propriedades do sistema, como por exemplo realizar uma escrita incompleta, atualizando apenas alguns servidores. Soluções que consideram clientes maliciosos são importantes, na medida em que os sistemas de quóruns foram desenvolvidos para operar em ambientes de larga escala, como a Internet, que com sua natureza de sistema aberto aumenta significativamente a possibilidade destes clientes estarem presentes no sistema. Considerando um ambiente dinâmico, esta necessidade se torna ainda mais evidente, visto que com a mobilidade dos processos (entrada e saída no sistema) aumenta a possibilidade de clientes (e até mesmo servidores) maliciosos estarem presentes no sistema.

No entanto, os únicos protocolos que suportam a presença de clientes maliciosos foram projetados para ambientes estáticos. Primeiramente, estes protocolos necessitavam que o sistema fosse replicado em $4f + 1$ servidores para suportar até f falhas, não utilizando assinaturas digitais (MALKHI; REITER, 1998a, 1998b). Porém, os protocolos propostos nestes trabalhos ainda permitem que um cliente execute ações maliciosas, como preparar várias escritas para serem executadas por um aliado após sua exclusão do sistema, interferindo assim no funcionamento dos clientes corretos. Recentemente, estes problemas foram tratados no protocolo BFT-BC (LISKOV; RODRIGUES, 2006), que requer apenas $3f + 1$ servidores e permite que um cliente prepare uma escrita somente após terminar a escrita anterior. Para isso, os servidores utilizam assinaturas assimétricas (principal fonte de atrasos em protocolos tolerantes a faltas bizantinas (CASTRO; LISKOV, 2002)) no controle das ações dos clientes e na garantia da integridade dos dados armazenados.

3.4.4.2 Sistemas de Quóruns em SDD

Os primeiros trabalhos sobre sistemas de quóruns em ambientes dinâmicos consideram apenas a possibilidade de faltas por parada de processos pertencentes ao sistema. Assim, o protocolo RAMBO I (LYNCH; SHVARTSMAN, 2002) representa um sistema de quóruns tolerante a faltas por parada onde processos podem entrar e sair do sistema. Um consenso é utilizado para a reconfiguração do sistema que é fracamente acoplada ao algoritmo principal. Isso torna possível a existência de mais de uma configuração

(visão) ao mesmo tempo, sendo que todas estas configurações são utilizadas nas operações de leitura e escrita. O protocolo também contempla um serviço para remoção de configurações antigas, que é modificado em (GILBERT et al., 2003) para acelerar o processo de reconfiguração do sistema.

Além destes trabalhos que consideram apenas falhas por parada, alguns outros protocolos foram desenvolvidos com o intuito de prover aos sistemas de quóruns suporte a falhas maliciosas. A seguir são apresentadas as principais características de uma série destes trabalhos que consideram a possibilidade de servidores maliciosos estarem presentes no sistema.

Primeiramente, com o objetivo de superar a limitação dos sistemas de quóruns estáticos em sempre considerar o pior caso para o número de faltas apresentadas pelo sistema[2], (ALVISI et al., 2000) propuseram protocolos que permitem aumentar e diminuir o número máximo de faltas suportadas pelo sistema, i.e., permitem ajustar o valor de f (*threshold* de faltas). Porém, os protocolos discutidos em (ALVISI et al., 2000) não suportam a inclusão e/ou remoção de processos no sistema. Além disso, estes protocolos requerem mais servidores do que é exigido na abordagem estática ($n \geq 6f_{max} - 2f_{min} + 1$, para variar o número de faltas em um intervalo $[f_{min}, f_{max}]$). Note que a expressão que limita o n é uma generalização de $n \geq 4f + 1$ necessário para quóruns estáticos (*b-masking quorum system* (MALKHI; REITER, 1998a, 1998b)), onde $f_{min} = f_{max}$. Estes protocolos garantem a semântica segura (*safe*) das variáveis compartilhadas.

Em (KONG et al., 2003, 2005) são apresentados protocolos para sistemas de quóruns que permitem remover processos do sistema. No entanto, tais protocolos não contemplam a entrada de processos, não sendo possível adicionar novos processos no sistema, que deve possuir no mínimo $n \geq 4f + 1$ servidores para o correto funcionamento. Além disso, um modelo probabilista de detecção de faltas maliciosas para quóruns é apresentado em (KONG et al., 2003), o qual utiliza um conjunto de servidores para diagnosticar este comportamento malicioso. Este trabalho assume que este conjunto de servidores, bem como os clientes, não podem se comportar maliciosamente.

Uma arquitetura para armazenamento tolerante a faltas bizantinas que se preocupa com aspectos relacionados com a escalabilidade e o balanceamento de carga é apresentada em (RODRIGUES et al., 2002; RODRIGUES; LISKOV, 2004). Nesta arquitetura os dados não são armazenados em todos os servidores, mas através de um protocolo (*consistent hashing*) é determinado quais servidores são responsáveis por determinado dado. O modelo assume clientes corretos e requer $n \geq 3f + 1$ servidores, armazenando dados auto-

[2]No modelo estático o número máximo de faltas suportadas pelo sistema (f) é definido de forma pessimista e em todas as execuções é considerado que f processos são faltosos. Porém, não é verdade que necessariamente teremos f processos faltosos em cada execução do sistema.

verificáveis. Neste trabalho, alguns nós do sistema são usados para formar um *serviço de configuração* que utiliza um protocolo de consenso bizantino para trocar a visão do sistema (reconfiguração). As trocas de visões são executadas apenas no final de uma época, i.e., todas as mudanças ocorridas no sistema (entradas e saídas de processos) apenas são computadas no final de uma época. Estes nós monitoram todo o sistema para determinar nós falhos (apenas por *crash*) a fim de removê-los do sistema. A troca de configuração (visão) necessita de *loosely synchronized clocks* entre as réplicas do serviço de configuração, o qual permite que os processos adotem tempo similar para a duração de uma época. Caso mais de f réplicas do serviço de configuração estejam mais lentas do que as outras, o tempo que certas condições devem ser satisfeitas pelo sistema é aumentado, i.e., a troca de visão é atrasada.

Um *framework* para adaptar vários protocolos baseados em quóruns aos ambientes dinâmicos (suportar entradas e saídas de processos) é apresentado em (MARTIN; ALVISI, 2004; MARTIN; ALVISI, 2004). Para transformar os protocolos estáticos em dinâmicos, é definida uma primitiva chamada DQ-RPC (a versão dinâmica da primitiva Q-RPC empregada nos protocolos estáticos (MALKHI; REITER, 1998a)), onde as chamadas remotas consideram a possibilidade de mudança no conjunto de processos do sistema. As propriedades do sistema são definidas em termos dos dados retornados pelas operações e não mais pelas propriedades da intersecção dos quóruns. Além disso, neste modelo uma terceira parte confiável (administrador correto) é responsável pela troca de visões (reconfiguração do sistema). Considerando que as tarefas deste administrador são executadas por um processo, em (MARTIN; ALVISI, 2004) é discutido um protocolo que mantém a consistência do sistema mesmo que este administrador apresente falha por parada. Porém, após a falha do administrador, não é mais possível reconfigurar o sistema.

Como podemos perceber, a maioria das propostas para sistemas de quóruns dinâmicos adota um algoritmo de consenso para reconfigurar o sistema (ou utiliza um administrador centralizado) (LYNCH; SHVARTSMAN, 2002; MARTIN; ALVISI, 2004; RODRIGUES; LISKOV, 2004). No entanto, é impossível resolver o consenso em um sistema assíncrono (CHANDRA; TOUEG, 1996) e sistemas de quóruns estáticos foram desenvolvidos para operar nestes ambientes assíncronos (GIFFORD, 1979; MALKHI; REITER, 1998a). Desta forma, estas abordagens para sistemas de quóruns dinâmicos fazem com que seja necessário aumentar o grau de sincronismo do sistema quando passamos de um ambiente estático para um ambiente dinâmico. Este fato está diretamente relacionado com a necessidade de reconfiguração em um sistema dinâmico.

Neste sentido, recententemente foi proposto um protocolo para sistemas de quóruns dinâmicos que não utiliza consenso nas reconfigurações. Este

protocolo, chamado DynaStore (AGUILERA et al., 2009, 2011), foi desenvolvido para sistemas que apresentam apenas falhas por parada de processos. Em vez de consenso, no DynaStore cada visão possui um *weak snapshot object* associado e os processos atualizam uma visão através da escrita de *updates* (atualizações) no seu objeto associado. O protocolo de reconfiguração do DynaStore é fortemente acoplado aos protocolos de leitura e escrita no registrador. Além disso, os processos podem executar uma reconfiguração no sistema em qualquer momento. Esta abordagem pode reduzir o desempenho do sistema, gastando a maior parte do tempo executando reconfigurações e atrasando as operações de leitura e escrita. As reconfigurações do DynaStore geram um grafo de visões, sendo que operações dos clientes são executadas apenas em determinadas visões, que formam uma sequência de visões. Por outro lado, o DynaStore não especifica quando um processo pode deixar o sistema e nem como executar os *weak snapshot objects*, uma vez que estes objetos são uma espécie de registradores replicados que devem permanecer disponíveis no sistema.

3.4.5 Replicação Máquina de Estados

A replicação Máquina de Estados (SCHNEIDER, 1990) é a abordagem mais abrangente e também a mais usada na implementação de sistemas tolerantes a faltas, tanto por parada (SCHNEIDER, 1990) quanto bizantinas (CASTRO; LISKOV, 2002). Esta abordagem consiste em replicar os servidores e coordenar as interações entre os clientes e as réplicas dos servidores, com o intuito de que as várias réplicas apresentem a mesma evolução em seus estados, i.e., em qualquer ponto da execução do sistema distribuído todas as réplicas devem possuir o mesmo estado.

Para isso, a replicação Máquina de Estados exige que todas as réplicas, *(i)* partindo de um mesmo estado e *(ii)* executando o mesmo conjunto de requisições na mesma ordem, *(iii)* cheguem ao mesmo estado final, o que define o determinismo de réplicas.

O item *(i)* é facilmente garantido, bastando iniciar todas as réplicas com o mesmo estado (i.e., iniciar todas as variáveis que representam o estado com os mesmos valores nas diversas réplicas). Para prover o item *(ii)*, é necessário a utilização de um protocolo de difusão atômica (ver seção 3.4.3). Já para alcançar o item *(iii)*, é necessário que as operações executadas pelas réplicas sejam deterministas, i.e., que a execução de uma mesma operação (com os mesmos parâmetros) resulte no mesmo resultado nas diversas réplicas. Além disso, o estado produzido (a mudança no estado) por esta operação deve ser o mesmo nas várias réplicas do sistema.

3.4.5.1 Replicação Máquina de Estados em SDD

Praticamente não existem trabalhos que tratem diretamente de replicação Máquina de Estados em sistemas distribuídos dinâmicos, como por exemplo em MANETs. Isto deve-se principalmente ao fato de que este modelo de replicação geralmente é construído a partir da aplicação direta de protocolos de difusão atômica e, se considerarmos um nível mais inferior, de protocolos de consenso.

É importante notar que, protocolos desenvolvidos para replicação Máquina de Estados em ambientes dinâmicos devem suportar a entrada e saída aleatória de processos no sistema. Assim, além das questões já destacadas, um esquema para atualização do estado dos participantes que entram no sistema torna-se indispensável. É necessário também sincronizar estas entradas no sistema com a execução dos protocolos de difusão atômica a fim de permitir que todos os participantes executem a mesma sequência de operações, preservando assim a consistência de seus estados.

Recentemente uma discussão superficial (sem apresentar protocolos e nem mesmo se aprofundar nas questões envolvidas em uma reconfiguração) sobre como reconfigurar uma replicação Máquina de Estados foi apresentada em (LAMPORT et al., 2010). Nesta abordagem, a reconfiguração da replicação Máquina de Estados é realizada através da execução do protocolo de consenso que implementa sua camada de difusão atômica. Neste sentido, as requisições de reconfiguração são ordenadas normalmente (como se fossem uma requisição de um cliente) mas não são entregues para a aplicação. Pelo contrário, estas requisições fazem com que todos os processos do sistema atualizem suas visões no mesmo ponto de execução (uma vez que estas requisições são ordenadas juntamente com as requisições dos clientes). Duas formas de reconfigurações são abordadas: a primeira utiliza uma única máquina de estados onde as requisições de reconfiguração atualizam o conjunto de membros que implementam esta máquina; a segunda abordagem utiliza uma sequência de máquina de estados onde as requisições de reconfiguração param a máquina em execução e inicializam uma nova máquina de estados a partir do estado final da máquina anterior.

3.4.6 Outros

Esta seção apresenta outros conceitos e problemas, cujos protocolos desenvolvidos para tratá-los também são utilizados na construção de aplicações distribuídas mais elaboradas. No entanto, tais protocolos ganham menos destaque neste texto por tratar de problemas que não serão abordados em nos-

sos estudos. Apesar disso, estes mecanismos são úteis no desenvolvimento da maioria dos protocolos já discutidos. Por exemplo, o roteamento é necessário na elaboração de qualquer um destes protocolos, pois é essencial para a comunicação entre dois processos.

3.4.6.1 Detecção de Falhas

Os detectores de falhas (CHANDRA; TOUEG, 1996) foram introduzidos com o objetivo de contornar a impossibilidade da realização do consenso em redes assíncronas (FISCHER et al., 1985). Em essência, detectores de falhas enriquecem o sistema com algum grau de sincronismo, geralmente representado por um *timeout*, e utilizam alguma técnica, normalmente *heartbeats*, com o objetivo de identificar processos falhos ou suspeitos de terem falhado. Note que estes detectores funcionam apenas para o modelo de falha por parada, pois um processo malicioso pode enganar os detectores (por exemplo, respondendo aos *heartbeats*), fazendo-se parecer como um processo correto.

Informalmente, detectores de faltas são oráculos distribuídos que fornecem dicas aos processos sobre quais deles estão falhos. Formalmente, as características dos detectores de falhas são descritas em termos das propriedades de *completude* e de *exatidão* (CHANDRA; TOUEG, 1996). A completude estabelece que todos os processos falhos acabarão por ser permanentemente suspeitos por algum processo correto do sistema. Já a propriedade de exatidão limita os erros (suspeição de processos corretos) que o detector possa cometer, estabelecendo que um processo correto acabará por nunca ser suspeito por nenhum outro processo correto. As variações destas propriedades são apresentadas abaixo:

- **Completude forte**: todos os processos corretos acabarão por suspeitar permanentemente de todos os processos faltosos;
- **Completude fraca**: todos os processos faltosos acabarão por ser suspeitos por algum processo correto;
- **Exatidão forte**: nenhum processo é suspeito antes de falhar;
- **Exatidão fraca**: algum processo correto nunca será suspeito;
- **Exatidão forte após um tempo**: há um tempo depois do qual todos os processos corretos não serão suspeitos por qualquer processo correto;
- **Exatidão fraca após um tempo**: há um tempo depois do qual algum processo correto não será considerado suspeito por qualquer processo correto.

A combinação destas propriedades define oito classes de detectores de falhas (CHANDRA; TOUEG, 1996): *Perfeito* ($\mathscr{P}$), *Forte* ($\mathscr{S}$), *Perfeito Após um Tempo* ($\diamond\mathscr{P}$), *Forte Após um Tempo* ($\diamond\mathscr{S}$), *Quase-Perfeito* ($\mathscr{Q}$), *Fraco* ($\mathscr{W}$), *Quase-Perfeito Após um Tempo* ($\diamond\mathscr{Q}$), *Fraco Após um Tempo* ($\diamond\mathscr{W}$). A Tabela 3 apresenta estas classes em termos das propriedades atendidas.

Tabela 3 – Classes de detectores de falhas (CHANDRA; TOUEG, 1996).

Completude	**Exatidão**			
	Forte	Fraca	Forte Após um Tempo	Fraca Após um Tempo
Forte	*Perfeito* $\mathscr{P}$	*Forte* $\mathscr{S}$	*Perfeito Após um Tempo* $\diamond\mathscr{P}$	*Forte Após um Tempo* $\diamond\mathscr{S}$
Fraca	*Quase-perfeito* $\mathscr{Q}$	*Fraco* $\mathscr{W}$	*Quase-perfeito Após um Tempo* $\diamond\mathscr{Q}$	*Fraco Após um Tempo* $\diamond\mathscr{W}$

Detectores pertencentes às classes $\mathscr{P}$ e $\mathscr{Q}$ são considerados *confiáveis*, por satisfazerem a propriedade de exatidão forte. A implementação deste tipo de detectores requer que o sistema seja síncrono, uma vez que dependem fortemente de hipóteses de tempo para identificação de processos falhos.

Já os detectores que não satisfazem a propriedade de exatidão forte são considerados *não-confiáveis*, i.e., estes detectores podem cometer erros na detecção. Assim, estes detectores não são capazes de darem respostas deterministas sobre o estado de um processo, fornecendo apenas dicas sobre tal estado através da identificação de processos *suspeitos* e *não-suspeitos*. Deste modo, um processo *suspeito* não necessariamente é realmente falho, da mesma forma que um processo *não-suspeito* pode ter falhado logo após ter recebido essa classificação.

Considerando um sistema distribuído dinâmico, a principal dificuldade no projeto de detectores de faltas é a visão restrita que cada entidade possui do sistema como um todo (visão parcial do sistema). Assim, os detectores de faltas locais relacionados com uma entidade serão capazes de atuar apenas no escopo restrito da visão desta entidade, e nada poderão afirmar sobre entidades fora desta visão.

Um detector de faltas assíncrono para redes móveis e auto-organizáveis é proposto em (SENS et al., 2008; GREVE et al., 2011a, 2011b). O modelo assume uma rede desconhecida, onde, a priori, um processo não conhece os demais participantes do sistema. O princípio básico de funcionamento destes detectores é a inundação da rede através de um mecanismo de pergunta-resposta (*query-response*). Durante a execução do protocolo, os participantes do sistema podem continuamente se deslocar (desconectar) e se reconectar ou falhar. No entanto, para o correto funcionamento destes detectores, a rede deve possuir uma determinada conectividade mínima. Posteriormente, este

protocolo foi estendido para tolerar participantes maliciosos presentes no sistema (LIMA; GREVE, 2009; LIMA et al., 2011).

3.4.6.2 Protocolos de Roteamento

Protocolos de roteamento são usados para determinar a rota (ou direção imediata) a ser usada no envio de informações entre quaisquer dois nós. Existem muitas propostas de protocolos de roteamento na literatura. No entanto, como este não é o foco deste trabalho, a seguir são brevemente descritos apenas alguns protocolos de roteamento para redes MANETs, os quais propõem mecanismos para tolerar comportamento malicioso de participantes do sistema. Além de considerar as mudanças que ocorrem na topologia da rede, protocolos de roteamento para MANETs ainda devem se preocupar com as características de escassez de recursos das entidades que compõem tais redes.

O protocolo proposto em (AWERBUCH et al., 2002) implementa um serviço de roteamento onde qualquer grupo intermediário de nós pode se comportar maliciosamente, bastando que o nó emissor e o nó destino de uma mensagem sejam corretos (confiáveis). Qualquer nó intermediário (entre o nó emissor e o nó destino) pode se autenticar e participar do protocolo, mesmo sendo um nó comprometido que exibirá comportamento malicioso. O protocolo utiliza um esquema de atribuição de pesos para os caminhos, que é utilizado na descoberta das rotas para evitar a escolha de caminhos comprometidos. Além disso, este protocolo apresenta uma fase de detecção de comportamento bizantino onde um *link* faltoso é encontrado após $log(n)$ faltas, sendo que n é o tamanho do caminho (número de nós presentes no caminho).

De acordo com o número de caminhos descobertos, os protocolos de roteamento podem ser divididos em: *caminho simples*, quando apenas um caminho é descoberto como no protocolo apresentado em (AWERBUCH et al., 2002); e *múltiplos caminhos*, quando múltiplos caminhos são descobertos como nos protocolos apresentados em (PAPADIMITRATOS; HAAS, 2002; KOTZANIKOLAOU et al., 2005). Os protocolos de roteamento que utilizam (descobrem) múltiplos caminhos são mais resistentes à ataques, por exemplo ataques de negação de serviço, do que os outros. Além disso, esta classe de protocolos possibilita proteger a disponibilidade da rede inclusive na presença de processos maliciosos.

O protocolo de roteamento SRP (*Secure Routing Protocol*) (PAPADIMITRATOS; HAAS, 2002) utiliza um mecanismo de descoberta de rotas onde o nó emissor consegue determinar múltiplos caminhos até o nó destino. A partir disso, o emissor pode utilizar uma ou mais destas rotas para enviar

seus dados, sendo necessária apenas uma associação de segurança entre o emissor e o destino (isto é, uma chave de sessão compartilhada entre emissor e destino). Este protocolo garante que o nó emissor obtém informação correta sobre a topologia do sistema mesmo na presença de participantes maliciosos. No entanto, apesar de suportar que um conjunto de nós seja malicioso, o protocolo SRP não suporta um conluio de nós maliciosos contra o protocolo dentro de um período de descoberta da rota.

O protocolo SECMR (*Secure Multipath Routing*) (KOTZANIKOLAOU et al., 2005) também utiliza múltiplos caminhos. Este protocolo estabelece as rotas sob demanda e garante que todos os caminhos até o nó de destino são descobertos pelo nó emissor. As rotas (caminhos) descobertas não são cíclicas e são disjuntas nos nós. Este protocolo é dividido em duas fases: *neighborhood authentication*, nesta fase os nós vizinhos autenticam-se através da utilização de criptografia assimétrica (cada nó possui uma par de chaves); e *route discovery and maintenance*, esta fase envolve o estabelecimento e a manutenção de rotas.

3.5 CONSIDERAÇÕES FINAIS

A busca de soluções para estes problemas clássicos é importante na medida em que estes protocolos são utilizados no desenvolvimento de qualquer aplicação distribuída que possua requisitos de segurança de funcionamento. Os protocolos desenvolvidos para tratar destes problemas apresentam complexidade variada, dependendo do modelo de sistema considerado, principalmente do tipo de falta a ser suportada pelo sistema. No entanto, a solução para qualquer um destes problemas torna-se ainda mais complexa quando consideramos um ambiente dinâmico e todas as suas incertezas.

Este capítulo apresentou os principais conceitos sobre segurança de funcionamento em sistemas distribuídos dinâmicos, destacando as principais vulnerabilidades encontradas no desenvolvimento de aplicações para estes ambientes. Além disso, alguns problemas clássicos encontrados no desenvolvimento de aplicações distribuídas foram descritos, sendo que as principais soluções para estes problemas em ambientes dinâmicos foram discutidas. A partir destas análises podemos perceber que existe um amplo campo de pesquisa a ser explorado em sistemas distribuídos dinâmicos, principalmente se considerarmos a possibilidade de entidades maliciosas estarem presentes no sistema.

4 CONSENSO BIZANTINO ENTRE PARTICIPANTES DESCONHECIDOS

Este capítulo aborda o problema do consenso em um novo ambiente, onde os participantes de determinada computação distribuída são inicialmente desconhecidos. Neste modelo, o conjunto de participantes pode sofrer alterações entre as execuções do protocolo, caracterizando o modelo de chegadas finitas de processos. Neste sentido, este capítulo apresenta as condições necessárias e suficientes para resolver o consenso nestes ambientes. Primeiramente, um padrão seguro de falhas bizantinas e um protocolo de *broadcast* são definidos, para então os protocolos para solução do consenso serem abordados. Então, são apresentadas as provas de que as hipóteses assumidas representam os requisitos mínimos para que o consenso admita solução. Por fim, algumas simulações com os protocolos propostos são analisadas.

4.1 INTRODUÇÃO

O problema do *consenso* (LAMPORT et al., 1982; FISCHER et al., 1985; TOUEG, 1984; CHANDRA; TOUEG, 1996; CORREIA et al., 2006), e de um modo mais geral os algoritmos de *acordo*, formam a base para a solução da maioria dos problemas encontrados no desenvolvimento de sistemas distribuídos confiáveis, pois possibilitam que os participantes da computação distribuída coordenem suas ações de forma a manter a consistência em seus estados e garantir o progresso do sistema. Este problema foi estudado longamente em redes clássicas, onde o conjunto de processos que participam de determinada computação é estático e conhecido por todos os participantes do sistema. Mesmo nestes ambientes, o problema do consenso não admite solução determinista na presença de um único processo faltoso (simples falta por parada), quando as entidades possuem comportamento assíncrono (FISCHER et al., 1985).

Em sistemas auto-organizáveis, como redes *ad-hoc*, as dificuldades encontradas na elaboração de protocolos para resolver o problema do consenso aumentam significativamente. Nestes ambientes, o conhecimento inicial sobre a composição do sistema é uma premissa muito forte para ser adotada, pois o número de participantes e seus conhecimentos sobre a composição do sistema não podem ser previamente determinados com precisão e aceitos como parâmetros estáticos. De fato, estes ambientes definem um novo modelo de sistemas distribuídos com diferenças essenciais em relação ao modelo clássico. Desta forma, novos desafios são encontrados na especificação

e resolução de problemas fundamentais. No caso do consenso, a maioria dos protocolos existentes não pode ser usada nestes ambientes dinâmicos, pois seus modelos de computação consideram um conjunto estático de participantes previamente conhecidos. Para o modelo de comunicação por passagem de mensagens (*message passing model*), algumas notáveis exceções são os trabalhos de Cavin *et al.* (CAVIN et al., 2004, 2005) e Greve *et al.* (GREVE; TIXEUIL, 2007).

Cavin *et al.* (CAVIN et al., 2004, 2005) definiu um novo problema chamado FT-CUP (*Fault-Tolerant Consensus with Unknown Participants*), que mantém a definição do consenso mas considera que os participantes não conhecem o conjunto de processos que compõem o sistema. Então, as condições necessárias e suficientes para resolver o FT-CUP foram identificadas, onde consideraram o grau de conhecimento a respeito da composição do sistema e os requerimentos de sincronia relativo à detecção de faltas. Neste sentido, concluíram que para resolver o FT-CUP com o menor grau de conectividade possível é necessário enriquecer o sistema com o maior grau de sincronismo possível, i.e., detectores de faltas perfeitos $\mathscr{P}$ (CHANDRA; TOUEG, 1996).

Greve e Tixeuil (GREVE; TIXEUIL, 2007) mostraram que, para resolver o consenso em redes desconhecidas sujeitas a faltas, realmente existe um *trade-off* entre o grau de conhecimento sobre a composição do sistema e as condições de sincronismo. Neste sentido, propõem uma solução alternativa para o FT-CUP, a qual possui requisitos mínimos de sincronia (mesmo grau já identificado como necessário para resolver o consenso em redes clássicas, que é representado por detectores de faltas da classe $\diamondsuit\mathscr{S}$ (CHANDRA; TOUEG, 1996)) mas exige que os participantes obtenham um maior grau de conhecimento sobre a composição do sistema. A abordagem utilizada no protocolo proposto é modular: inicialmente, algoritmos identificam um conjunto de participantes que compartilham a mesma visão do sistema; na sequência, qualquer algoritmo de consenso clássico (inclusive os projetados para redes tradicionais) é executado por estes participantes que difundem o valor de decisão do consenso para todo o sistema.

Neste contexto, nossos esforços concentram-se na extensão destes resultados, através do estudo do problema do consenso em redes desconhecidas considerando a possibilidade de processos maliciosos (LAMPORT et al., 1982) estarem presentes no ambiente. Definimos este problema como BFT-CUP: *Byzantine Fault-Tolerant Consensus with Unknown Participants*. Neste sentido, estabelecemos as condições necessárias e suficientes para resolver o BFT-CUP. Um algoritmo para solucionar este problema é apresentado, identificando-se assim as condições suficientes. Além disso, é provado que este algoritmo é ótimo em termos de sincronia e grau de conhecimento necessário para resolver o BFT-CUP, identificando-se assim as condições ne-

cessárias. O algoritmo proposto não requer a utilização de assinaturas digitais, que é a principal fonte de latência em protocolos que toleram faltas bizantinas (CASTRO; LISKOV, 2002). Adicionalmente, mostramos que os requerimentos definidos como necessários para resolver o BFT-CUP não diminuem mesmo com a utilização de assinaturas digitais. As seções seguintes apresentam os protocolos utilizados na solução do BFT-CUP.

4.2 DEFINIÇÕES PRELIMINARES

Esta seção apresenta algumas definições preliminares, abordando o modelo de sistema considerado e o conceito de detectores de participação.

4.2.1 Modelo de Sistema

Consideramos um sistema distribuído composto por um conjunto finito Π de processos (também chamados de participantes ou nós) extraídos do conjunto universo U. Em uma *rede conhecida*, Π é conhecido por todos os processos do sistema, enquanto que em uma *rede desconhecida*, um processo $i \in \Pi$ conhece apenas um subconjunto $\Pi_i \subseteq \Pi$. A chegada dos processos no sistema segue o modelo de chegadas finitas (M_1 - Tabela 1).

Os processos do sistema estão sujeitos a *faltas bizantinas* (LAMPORT et al., 1982), i.e., processos faltosos podem exibir qualquer comportamento, podendo parar, omitir envio ou entrega de mensagens, ou desviar de suas especificações arbitrariamente e trabalhar em conjunto com o objetivo de corromper o sistema. Um processo que apresenta comportamento de falha é dito falho (ou faltoso), de outra forma é dito correto. Apesar de não conhecerem todos os participantes que compõem o sistema, os processos conhecem o número máximo de faltas suportadas pelo mesmo, denotado por f. Além disso, consideramos que os processos possuem identificadores únicos, sendo inviável a obtenção de identificadores adicionais por processos faltosos com o objetivo de executar um ataque *Sybil* (Seção 3.2.1) contra o sistema.

Os processos comunicam-se através do envio e recebimento de mensagens por meio de *canais ponto-a-ponto autenticados e confiáveis*, estabelecidos entre processos conhecidos[1]. A autenticidade das mensagens disseminadas para processos ainda desconhecidos é verificada através de redundância no recebimento das mesmas, como explicado na Seção 4.4. Um processo i

[1] O uso de canais autenticados é fundamental para tolerar processos maliciosos em sistemas assíncronos, pois sem estes canais um processo faltoso pode se fazer passar por outros processos para algum outro processo vítima.

apenas pode enviar mensagens diretamente para outro processo j se $j \in \Pi_i$, i.e., i conhece j. Obviamente, caso i envie uma mensagem para j sendo que $i \notin \Pi_j$, quando j receber esta mensagem o mesmo pode adicionar i em Π_j, i.e., j passa a conhecer i e torna-se capaz de enviar mensagens para i. Consideramos a existência de uma camada de roteamento subjacente resistente à faltas bizantinas (Seção 3.4.6.2) de tal modo que se $j \in \Pi_i$ e existe conectividade suficiente na rede, então i é capaz de enviar uma mensagem de forma confiável para j. Por exemplo, o SECMR (KOTZANIKOLAOU et al., 2005) é um protocolo de roteamento seguro que descobre múltiplos caminhos. Desta forma, a comunicação correta entre dois participantes é garantida desde que exista pelo menos um caminho correto entre estes participantes, i.e., nenhum participante ou canal que forma este caminho é faltoso.

Não existe nenhuma suposição relacionada com a tempo necessário para transmissões de mensagens e para computações locais nos processos, i.e., o sistema é assíncrono. No entanto, o protocolo apresentado nesta seção utiliza um algoritmo de consenso tradicional (para redes estáticas) tolerante a faltas bizantinas, o qual pode ser implementado sobre um sistema parcialmente síncrono (DWORK et al., 1988) (ex.: Paxos Bizantino (CASTRO; LISKOV, 2002)) ou sobre um sistema completamente assíncrono (ex.: protocolos randômicos (CORREIA et al., 2006; BRACHA, 1984; BEN-OR, 1983)). Assim, nosso protocolo requer o mesmo nível de sincronia exigido pelo algoritmo de consenso bizantino tradicional utilizado pelo mesmo.

4.2.2 Detectores de Participação

Para resolver qualquer problema distribuído que exige cooperação entre os participantes, os processos precisam obter de alguma forma uma visão (mesmo que parcial) da composição do sistema. Os *detectores de participação*, chamados de PD, foram propostos com o objetivo de se determinar este subconjunto de processos conhecidos (CAVIN et al., 2004). Os detectores de participação podem ser vistos como oráculos distribuídos que fornecem dicas sobre os processos que estão participando em determinada computação.

Seja $i.PD$ o detector de participação do processo i. Quando consultado por i, $i.PD$ retorna um subconjunto de processos, contidos em Π, com os quais i pode colaborar. Seja $i.PD(t)$ a consulta de i realizada no tempo t. A informação fornecida por $i.PD$ pode evoluir entre as consultas, mas deve satisfazer às seguintes propriedades:

- *Inclusão da Informação*: A informação retornada pelos detectores de participação não diminui entre as consultas (com o transcorrer do tempo), i.e., $\forall i \in \Pi, \forall t' \geq t : i.PD(t) \subseteq i.PD(t')$;

- *Exatidão da Informação*: Os detectores de participação não cometem erros no sentido de retornar processos que não pertencem a Π, i.e., $\forall i \in \Pi, \forall t : i.PD(t) \subseteq \Pi$.

Detectores de participação fornecem um contexto inicial sobre a composição do sistema através do qual é possível expandir o conhecimento sobre Π. Assim, esta abstração enriquece o sistema com um grafo de conectividade por conhecimento. Este grafo é orientado, pois o conhecimento fornecido pelos detectores de participação não é necessariamente bidirecional (CAVIN et al., 2004).

Definição 1 ***Grafo de Conectividade por Conhecimento****: Seja $G_{di} = (V, \xi)$ um grafo orientado representando a relação de conhecimento determinada pelo oráculo PD. Então, $V = \Pi$ e $(i, j) \in \xi$ se e somente se $j \in i.PD$, i.e., i conhece j.*

Definição 2 ***Grafo Não-Orientado de Conectividade por Conhecimento****: Seja $G = (V, \xi)$ um grafo não orientado representando a relação de conhecimento determinada pelo oráculo PD. Então, $V = \Pi$ e $(i, j) \in \xi$ se e somente se $j \in i.PD$ ou $i \in j.PD$, i.e., i conhece j ou j conhece i.*

Baseado nas propriedades apresentadas pelos grafos de conectividade por conhecimento, algumas classes de detectores de participação foram propostas para resolver o CUP (CAVIN et al., 2004) e o FT-CUP (CAVIN et al., 2005; GREVE; TIXEUIL, 2007). O detector de participação k-OSR (*k-One Sink Reducibility*) é o mais fraco detector definido para resolver o FT-CUP em sistemas assíncronos (GREVE; TIXEUIL, 2007). Neste trabalho, redefinimos este detector de participação a fim de estabelecer condições mais fracas para a solução do BFT-CUP, considerando o grau de conectividade por conhecimento que deve ser adquirido pelos participantes.

Antes de definir como o detector de participação k-OSR encapsula o conhecimento sobre a composição do sistema, vamos definir algumas notações sobre grafos:

- Uma componente G_c de G_{di} é *k-fortemente conexa* se para qualquer par (v_i,v_j) de nós em G_c, v_i pode alcançar v_j através de k caminhos disjuntos nos nós.

- Uma componente G_{sink} de G_{di} é uma *componente poço* quando não existe caminhos partindo de nós pertencentes à G_{sink} para outros nós de G_{di}, com exceção dos nós na própria componente G_{sink}.

Definição 3 ***(k-One Sink Reducibility PD (k-OSR PD)).*** *O grafo de conectividade por conhecimento G_{di}, que representa a relação de conhecimento induzida pelo oráculo PD, satisfaz às seguintes condições:*

1. *O grafo não-orientado de conectividade por conhecimento G, obtido de G_{di}, é conexo;*

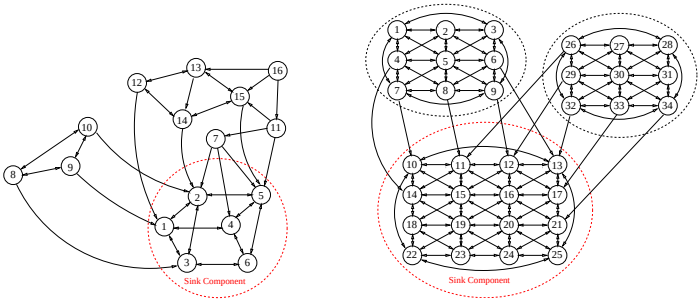
2. *O grafo direcionado acíclico, obtido pela redução de G_{di} às suas componentes fortemente conexas, tem uma e somente uma componente poço, chamada G_{sink};*

3. *A componente poço G_{sink} é k-fortemente conexa.*

4. *Para qualquer par p_i, p_j, tal que $p_i \notin G_{sink}$ e $p_j \in G_{sink}$, existem pelo menos k caminhos disjuntos nos nós de p_i para p_j.*

Esta nova definição de detector de participação também é abordada no trabalho recente de Greve e Tixeuil (GREVE; TIXEUIL, 2010), que apresenta novas discussões sobre a possibilidade de solução do FT-CUP, redefinindo o detector de participação *k*-OSR mas mantendo o mesmo nome. Para manter a coerência com aquele trabalho, optamos por continuar chamando este detector de participação de *k*-OSR.

Para melhor ilustrar a Definição 3, a Figura 1 apresenta dois grafos G_{di} induzidos por um detector de participação pertencente à classe *k*-OSR. As Figuras 1(a) e 1(b) mostram relações de conhecimento induzidas por detectores de participação das classes 3-OSR e 5-OSR, respectivamente. Por exemplo, na Figura 1(a), o valor retornado por $1.PD$ é o subconjunto $\{2,3,4\}$, que está contido em Π.



(a) 3-OSR (b) 5-OSR

Figura 1 – Grafos de conectividade por conhecimento induzidos por detectores de participação da classe *k*-OSR.

Em nossos algoritmos, consideramos que para cada processo *i*, seu detector de participação $i.PD$ é consultado exatamente uma vez no início da

execução do protocolo. Esta premissa pode ser implementada através do armazenamento do resultado da primeira consulta executada em $i.PD$ e do retorno deste valor em consultas subsequentes. Isto garante que a visão parcial sobre a composição inicial do sistema é consistente em todos os nós, o que define um grafo de conectividade por conhecimento G_{di} único e comum para todos os participantes do sistema. Neste trabalho usamos a seguinte notação: um participante p é *vizinho* de outro participante i se e somente se $p \in i.PD$; G_{di} é o grafo de conectividade por conhecimento induzido por um detector de participação k-OSR; G_{sink} é a única componente poço k-fortemente conexa de G_{di}; e um participante $p \in G_{di}$ é um *participante do poço* se e somente se $p \in G_{sink}$, caso contrário p é um *participante fora do poço*.

4.3 PADRÃO SEGURO DE FALHAS BIZANTINAS

Trabalhos anteriores mostraram que para resolver o consenso tolerante a faltas entre participantes desconhecidos em um sistema com o menor nível de sincronia, é necessário que os participantes obtenham um conhecimento sobre a composição do sistema representado por um detector de participação k-OSR (GREVE; TIXEUIL, 2007; GREVE; TIXEUIL, 2010; ALCHIERI et al., 2008b).

Nestes trabalhos, o valor do parâmetro de conectividade k é escolhido de forma conservativa, sempre considerando o pior cenário para todos os participantes do sistema. No entanto, este valor pode ser flexível de acordo com a posição dos processos faltosos no grafo de conectividade por conhecimento induzido por um PD tanto da classe k-OSR. Por exemplo, em um trabalho anterior (ALCHIERI et al., 2008b) propomos uma solução para o BFT-CUP que define o grau de conectividade como $k \geq 2f+1$ para tolerar até f participantes maliciosos. Isto significa que são necessários pelo menos $2f+1$ caminhos disjuntos nos nós ligando cada par de participantes de G_{sink}. Porém, o BFT-CUP admite solução caso existam pelo menos $f+1$ caminhos disjuntos nos nós *formados apenas por processos corretos* ligando estes participantes. Esta diferença é mais facilmente notada se considerarmos os requisitos de conectividade entre participantes fora do poço com participantes do poço.

Para melhor ilustrar este cenário, considere o grafo 5-OSR apresentado na Figura 1(b) (que também representa a conectividade por conhecimento induzida por um PD 5-OSR definido em (GREVE; TIXEUIL, 2007; ALCHIERI et al., 2008b)). A solução para o BFT-CUP apresentada em (ALCHIERI et al., 2008b) define que é possível tolerar até 2 participantes maliciosos neste cenário ($k \geq 2f+1$, $k=5$, $f=2$). Porém, considerando que os dois processos maliciosos são os nós 4 e 10, a Figura 2 apresenta um exem-

plo de conectividade por conhecimento suficiente para resolver o BFT-CUP. Note que, o grau de conhecimento que os nós obtém sobre a composição do sistema é maior no grafo da Figura 1(b). Não é necessário agrupar os participantes fora do poço em componentes k-fortemente conexas (Definição 3), porém utilizamos este exemplo para melhor visualização das diferenças visto que as soluções anteriores para o consenso tolerante a faltas entre participantes desconhecidos adotam uma definição para o PD k-OSR, onde além de ser definido que $k \geq 2f+1$ também é especificado que participantes fora do poço devem ser agrupados em componentes k-fortemente conexas (GREVE; TIXEUIL, 2007; GREVE; TIXEUIL, 2010; ALCHIERI et al., 2008b).

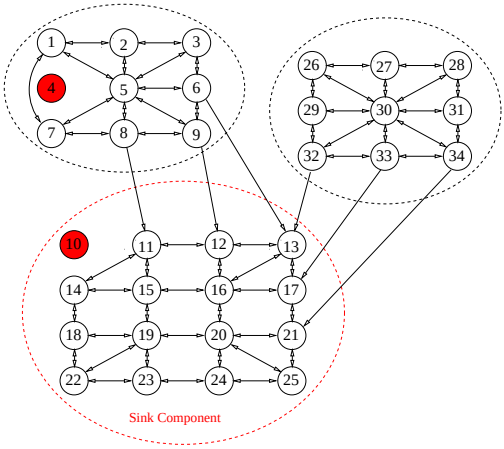


Figura 2 – k-OSR + Padrão Seguro de Falhas Bizantinas, $f = 2$ e $k = 3$.

Desta forma, para representar esta diminuição na necessidade de conhecimento sobre a composição do sistema, introduzimos um *padrão seguro de falhas bizantinas* que define o grau de conectividade necessário e suficiente para resolver o BFT-CUP, em um modelo de sistema não síncrono, pela imposição de uma restrição na combinação de participantes maliciosos. Esta restrição é bastante simples e define que o grau de conectividade deve ser de pelo menos $k \geq f+1$ caminhos disjuntos nos nós formados apenas por processos corretos.

Definição 4 ***(Padrão Seguro de Falhas Bizantinas).*** *Considere um grafo de conectividade por conhecimento G_{di} induzido por um PD k-OSR. Seja f o número de participantes em G_{di} que podem falhar e F o conjunto de participantes em G_{di} cujas falhas são permitidas, então $|F| \leq f$. O padrão seguro de falhas bizantinas especifica que $\{G_{di} \setminus F\} \in$ k-OSR, com $k \geq f+1$.*

Este padrão também exige que cada participante realize uma única chamada em seu PD (CAVIN et al., 2005), garantindo que a visão parcial sobre a composição inicial do sistema é consistente em todos os nós, o que define um grafo de conectividade por conhecimento G_{di} único e comum para todos os participantes do sistema.

As seções seguintes apresentam um protocolo que resolve o BFT-CUP usando um PD k-OSR e considera que o padrão seguro de falhas bizantinas é verificado para os processos retornados por este PD, provando que estes requisitos são suficientes para resolver o BFT-CUP. Na Seção 4.5.4 é provado que estes requisitos também são necessários para a solução deste problema.

4.4 REACHABLE RELIABLE BROADCAST

Esta seção introduz uma nova primitiva, chamada *reachable reliable broadcast*, usada para comunicação entre os processos que compõem o sistema. Esta primitiva é genérica suficiente para ser usada em qualquer sistema onde processos não conhecem todos os participantes presentes em determinada computação e necessitam realizar difusões (*broadcasts*) de mensagens de forma confiável. No contexto deste trabalho, esta primitiva será usada na solução do BFT-CUP. Antes de definir como os processos invocam esta primitiva, vamos apresentar o conceito de participante alcançável (*reachable*).

Definição 5 ***(Participante Alcançável).*** *Considere um grafo de conectividade por conhecimento G_{di} induzido por qualquer detector de participação e que f é o número de nós em G_{di} que podem falhar. Para qualquer dois participantes $p,q \in G_{di}$, q é alcançável a partir de p em G_{di} caso existam pelo menos $f+1$ caminhos disjuntos nos nós de p para q em G_{di}, formados apenas por processos corretos.*

Seja m uma mensagem, os processos acessam esta primitiva através de duas operações básicas:

- reachable_send (m,p) – através desta operação um participante p difunde m para todos os participantes alcançáveis a partir de p em G_{di}.
- reachable_deliver (m,p) – operação invocada no receptor para entrega de m, difundida pelo participante p.

Considerando a Definição 5, a primitiva *reachable reliable broadcast* deve satisfazer as seguintes propriedades:

- *RB_Validade*: Caso um participante correto p executa reachable_send (m,p) então *(i)* um participante correto q, alcançável a partir de p em

G_{di}, eventualmente executa reachable_deliver (m,p) *(ii)* ou não existe nenhum participante correto alcançável a partir de p em G_{di}.

- *RB_Integridade*: Para qualquer mensagem m, se um participante correto q executa reachable_deliver (m,p), então o participante p executou reachable_send (m,p) (*No spurious message*).

- *RB_Acordo*: Caso um participante correto q executa reachable_deliver (m,p), m difundida por um participante correto p, então todos os participantes corretos alcançáveis a partir de p em G_{di} executam reachable_deliver (m,p).

Note que estas propriedades estabelecem uma primitiva de comunicação com especificação semelhante às usualmente utilizadas em protocolos de difusão confiável (CHANDRA; TOUEG, 1996; CORREIA et al., 2006; BRACHA, 1984). No entanto, a primitiva proposta somente garante a entrega de mensagens em todos os processos corretos alcançáveis a partir dos nós que iniciam as difusões em G_{di} (Definição 5).

4.4.1 Protocolo

O Algoritmo 1 apresenta uma implementação para a primitiva de *reachable reliable broadcast*. A principal ideia de nossa implementação é que os participantes executem uma inundação (*flooding*) de suas mensagens no sistema, fazendo com que as mesmas cheguem em todos os participantes alcançáveis em G_{di} (Definição 5), os quais entregarão estas mensagens assim que suas autenticidades forem comprovadas.
Notações. O algoritmo utiliza as seguintes notações.

- $i.received\_msgs$ – saco contendo tuplas da forma $\langle m, m.route \rangle$, onde m é uma mensagem recebida pelo processo i e $m.route$ é uma lista ordenada de nós por onde m passou. A cauda de $m.route$ contém o identificador do emissor que iniciou a difusão de m.

- $computeRoutes(m, i.received\_msgs)$ – função que computa o número de caminhos disjuntos nos nós através dos quais a mensagem m foi recebida no participante i.

- $appendRoute(m.route, i)$ – função que adiciona i no final de $m.route$;

- $getFirstElement(m.route)$, $getLastElement(m.route)$ – estas funções retornam o primeiro e o último participante de $m.route$, respectivamente.

Descrição. Um processo i difunde uma mensagem m no sistema pela execução de reachable_send (m,i) (linha 6). Neste caso, através da mensagem RC_FLOODING$(m, m.route = i)$, i envia m para seus vizinhos, i.e., participantes retornados pelo seu detector de participação. A lista $m.route$ é anexada a esta mensagem, sendo inicializada com apenas i mas sempre contendo a rota acumulada de acordo com o caminho percorrido por m desde o emissor i até seus receptores.

Algoritmo 1 *Reachable Reliable Broadcast* (participante i).

```
constant:
1)  f : int                                          // upper bound on the number of failures

variables:
2)  i.received_msgs : bag of ⟨m, m.route⟩ tuples

message:
3)  RC_FLOODING:                                     // struct of this message
4)      m : message to flood                         // value to be disseminated
5)      route : ordered list of nodes                // path traversed by m

** Initiator Only **
upon invocation of reachable_send (m, i)
6)  ∀j ∈ i.PD, send RC_FLOODING(m, m.route = i) to j;
** All Nodes **
INIT:
7)  i.received_msgs ← ∅;

upon receipt of RC_FLOODING(m, m.route) from j
8)  if getLastElement(m.route) = j ∧ i ∉ m.route then
9)      appendRoute(m.route, i);
10)     i.received_msgs ← i.received_msgs ∪ {⟨m, m.route⟩};
11)     routes ← computeRoutes(m, i.received_msgs);
12)     if routes ≥ f + 1 then
13)         initiator ← getFirstElement(m.route);
14)         trigger reachable_deliver (m, initiator);
15)         i.received_msgs ← i.received_msgs \ {⟨m, ∗⟩};
16)     end if
17)     ∀z ∈ i.PD \ {j}, send RC_FLOODING(m, m.route) to z;
18) end if
```

Quando RC_FLOODING$(m, m.route)$, enviada por j, é recebida em i, primeiramente o conteúdo de m é analisado (linhas 8-16). Caso este conteúdo seja válido, i encaminha m para seus vizinhos, com exceção de j, e assim por diante (linha 17). Este procedimento implementa uma inundação (*flooding*) de m no sistema de tal forma que esta mensagem é recebida por todos os participantes alcançáveis, em G_{di}, a partir do emissor que originou m.

Na avaliação do conteúdo de RC_FLOODING$(m, m.route)$, i inicialmente certifica-se de que m realmente foi enviada por j e de que esta mensagem ainda não foi recebida (linha 8). Então, i adiciona seu identificador em $m.route$ (linha 9) e armazena m, juntamente com $m.route$, em $i.received\_msgs$ (linha 10). Finalmente, i entrega m se e somente se i recebeu m através de

$f+1$ caminhos disjuntos nos nós, de tal forma que a autenticidade de m é verificada. Esta verificação é realizada com a ajuda da função *computeRoutes* (linha 11). Se este for o caso, i executa o procedimento $reachable\_deliver(m, initiator)$, para a entrega de m enviada por $initiator$ e, então, remove os dados relacionados com esta mensagem de $i.received\_msgs$ (linha 15).

A solução apresentada nesta seção é baseada na abordagem de Dolev (DOLEV, 1982) e exige que cada participante adicione o seu identificador no final da rota percorrida por uma mensagem, antes de enviar ou encaminhar a mesma. Um participante somente processa uma mensagem recebida caso o participante que está enviando (ou encaminhando) a mesma apareça no final da rota acumulada. No entanto, um participante malicioso é capaz de modificar uma rota acumulada (removendo ou adicionando participantes), além de modificar ou bloquear uma mensagem que está sendo difundida. Porém, o grau de conectividade exigido (pelo menos $f+1$ caminhos disjuntos nos nós formados apenas por participantes corretos) garante que esta mensagem é recebida por todos os participantes alcançáveis a partir do emissor em G_{di}. Além disso, como um participante entrega uma mensagem apenas após o recebimento da mesma através de $f+1$ caminhos disjuntos nos nós, mensagens forjadas por participantes maliciosos não são entregues, pois, nestes casos, a autenticidade destas mensagens não são verificadas nos processos corretos.

O Algoritmo 1 apresenta um inconveniente de que uma mesma mensagem, difundida por algum participante, poderá ser entregue mais de uma vez pelos receptores. Este fenômeno não afeta a corretude deste algoritmo, o qual satisfaz as propriedades definidas para esta primitiva. Além disso, estas propriedades são suficientes para garantir a corretude dos algoritmos de consenso apresentados na seção seguinte. Desta forma, não tratamos esta limitação do algoritmo. No entanto, isto pode ser facilmente resolvido através da utilização de *buffers* que armazenam as mensagens entregues, as quais devem possuir identificadores únicos.

4.4.1.1 *Corretude*

A seguir apresentamos as provas de corretude do *reachable reliable broadcast* implementado pelo Algoritmo 1, onde consideramos que G_{di} é o grafo de conectividade por conhecimento induzido por qualquer PD e f é o número de nós em G_{di} que podem falhar.

Lema 1 *(RB_Validade) Caso um participante correto p executa* reachable_send *(m,p) então (i) um participante correto q, alcançável a partir de p em G_{di}, eventualmente executa* reachable_deliver *(m,p) (ii) ou não existe nenhum participante correto alcançável a partir de p em G_{di}.*

Prova. Pela Definição 5, uma vez que q é alcançável a partir de p em G_{di}, existe pelo menos $f+1$ caminhos disjuntos nos nós em G_{di} formados apenas por nós corretos de p para q. Seja $P : p = 0, 1, ..., k = q$ um destes caminhos, vamos provar por indução em k que q receberá a mensagem m através de P e executará as linhas 8-18. O caso base ($k = 1$) é trivial, uma vez que p é correto e então executa reachable_send (m,p) para todos os seus vizinhos retornados em $p.PD$ (linha 6). Pelo passo da indução, esta afirmação é válida para o participante i ($k = i$) em P. Então, na recepção de m por i, o predicado da linha 8 é satisfeito, pois os participantes no caminho P são corretos. Então, i executa a linha 17 e envia m para seus vizinhos incluindo $i+1 \in i.PD$ ($k = i+1$). Como os canais são confiáveis, $i+1$ receberá m e esta afirmação permanece válida. Isto garante que q recebe m através destes $f+1$ caminhos disjuntos nos nós (incluindo P), satisfazendo o predicado da linha 8 pelo menos $f+1$ vezes. Então, a autenticidade de m é verificada em q pela redundância no seu recebimento. Isto é realizado pela execução das linhas 9-11, que são responsáveis pelas informações sobre as diferentes rotas através das quais m foi recebida em q. Quando a autenticidade de m é provada (linha 12), a entrega de m é autorizada em q pela execução de reachable_deliver (m,p) (linha 14). Isto prova que se q é um participante correto alcançável a partir de p em G_{di}, então q entrega m pelo menos uma vez. Como consequência, caso m não seja entregue por algum participante correto em G_{di}, então não existe nenhum participante correto alcançável a partir de p em G_{di}. $\square_{Lema\ 1}$

Lema 2 *(RB_Integridade) Para qualquer mensagem m, se um participante correto q executa reachable_deliver (m,p), então o participante p executou reachable_send (m,p) (No spurious message).*

Prova. A mensagem m é entregue por um participante correto q apenas após seu recebimento através de $f+1$ caminhos disjuntos nos nós de p para q (linhas 8-16). Desta forma, um participante malicioso não é capaz de forjar que m foi difundida por p, pois mesmo com um conluio de até f participantes maliciosos, q obterá no máximo f caminhos disjuntos nos nós através dos quais m foi recebida "de p" (cada um destes caminhos conterá um participante malicioso). Isto garante que a mensagem m entregue por q é autêntica e realmente foi difundida por p. Além disso, isso também assegura que q é alcançável a partir de p em G_{di}. $\square_{Lema\ 2}$

Lema 3 *(RB_Acordo) Caso um participante correto q executa reachable_deliver (m,p), m difundida por um participante correto p, então todos os participantes corretos alcançáveis a partir de p em G_{di} executam reachable_deliver (m,p);*

Prova. Pelo Lema 2, se um participante correto q executa reachable_deliver (m,p), então algum participante p executou reachable_send (m,p). Pelo Lema 1, se um participante correto p executa reachable_send (m,p), então um participante correto q, alcançável a partir de p em G_{di}, eventualmente executa reachable_deliver (m,p). Desta forma, todos os participantes corretos alcançáveis a partir de p em G_{di} executam reachable_deliver (m,p). $\square_{Lema\ 3}$

***Reachable Reliable Broadcast*, PD k-OSR e o Padrão Seguro de Falhas Bizantinas.**

Na Seção 4.5, mostramos que um PD da classe k-OSR e o padrão seguro de falhas bizantinas são suficientes para resolver o BFT-CUP. Considere que G_{di} é um grafo de conectividade por conhecimento induzido por um PD k-OSR, onde G_{sink} é a única componente poço de G_{di}. A seguir, mostramos que estes requisitos também são suficientes para implementar um *reachable reliable broadcast* para os participantes em G_{sink}.

Observação 1 *(Alcançabilidade dos Participantes do Poço) Considerando um PD k-OSR e o padrão seguro de falhas bizantinas, cada participante $q \in G_{sink}$ é alcançável a partir de qualquer participante $p \in G_{di}$. Pelas definições do PD k-OSR (Definição 3) e do padrão seguro de falhas bizantinas (Definição 4), existem pelo menos $k \geq f+1$ caminhos disjuntos nos nós formados apenas por participantes corretos de qualquer participante $p \in G_{di}$ para cada participante $q \in G_{sink}$. Então, pela definição de participante alcançável (Definição 5), esta Observação é válida.*

Corolário 1 *(Entrega de Mensagens pelos Participantes do Poço) Considerando um PD k-OSR e o padrão seguro de falhas bizantinas, o Algoritmo 1 implementa um reachable reliable broadcast a partir de qualquer participante $p \in G_{di}$ para todos os participantes $q \in G_{sink}$.*

Prova. Pela Observação 1, cada participante correto $q \in G_{sink}$ é alcançável a partir de qualquer participante $p \in G_{di}$. Deste modo, esta prova é derivada diretamente a partir dos lemas 1 (*RB_Validade*), 2 (*RB_Integridade*) e 3 (*RB_Acordo*). $\square_{Corolário\ 1}$

4.5 BFT-CUP: CONSENSO BIZANTINO ENTRE PARTICIPANTES DESCONHECIDOS

Esta seção apresenta uma solução para o BFT-CUP. O protocolo é baseado na primitiva *reachable reliable broadcast* (Seção 4.4), a qual, juntamente com a camada subjacente de roteamento tolerante a faltas bizantinas,

englobam todos os detalhes relacionados com a comunicação entre os participantes. A Figura 3 apresenta as relações entre os protocolos usados para resolver o BFT-CUP.

O protocolo é dividido em três fases. Na primeira fase – chamada *descoberta dos participantes* (protocolo DISCOVERY) – cada participante aumenta seu conhecimento sobre a composição do sistema, descobrindo o maior número possível de participantes que estão presentes em determinada computação. A segunda fase – chamada *determinação da componente poço* (protocolo SINK) – define quais são os participantes que pertencem à componente poço G_{sink} do grafo de conhecimento G_{di} induzido por um PD k-OSR. No final desta fase, cada participante é capaz de determinar se pertence ou não à G_{sink}. Na última fase – chamada *realização do consenso* (protocolo CONSENSUS) – os participantes do poço executam um consenso tolerante a faltas bizantinas clássico e propagam o valor de decisão para os demais participantes do sistema.

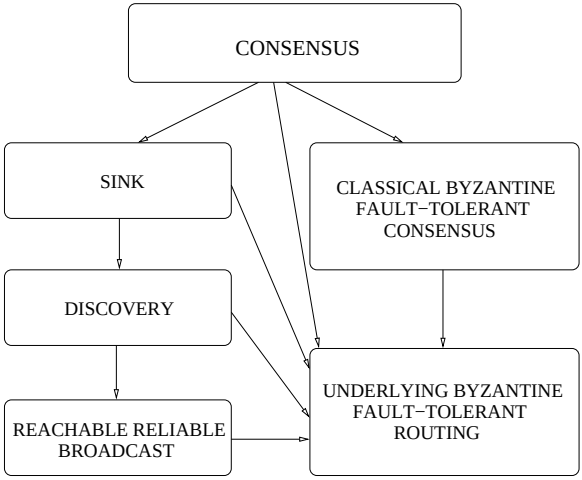


Figura 3 – Protocolos do BFT-CUP.

Suposição 1 *(Modelo de Sistema) Nos protocolos apresentados nas seções seguintes, consideramos que G_{di} é um grafo de conectividade por conhecimento induzido por um PD k-OSR e $G_{sink} = (V_{sink}, E_{sink})$ é a única componente poço de G_{di}, f é o número de participantes em G_{di} que podem falhar e F é o conjunto de participantes em G_{di} cujas falhas são permitidas, $|F| \leq f$. Além disso, o padrão seguro de falhas bizantinas que define $\{G_{di} \setminus F\} \in k$-OSR, $k \geq f+1$, é verificado durante a execução dos protocolos.*

4.5.1 1ª Fase: Descoberta dos Participantes

O primeiro passo necessário para resolver o consenso entre participantes desconhecidos é fazer com que os processos obtenham o máximo possível de conhecimento sobre a composição do sistema. Note que, através de seu detector de participação local, um processo obtém um conhecimento inicial sobre a composição do sistema que pode não ser suficiente para resolver o BFT-CUP. Sendo assim, um processo expande seu conhecimento através da execução do protocolo DISCOVERY, apresentado no Algoritmo 2.

A principal ideia deste algoritmo é que cada participante i realize uma busca em largura no grafo de conhecimento G_{di} induzido por um PD k-OSR. Para isso, i difunde uma mensagem solicitando informações sobre os vizinhos de cada participante alcançável a partir de i em G_{di}. Uma característica importante deste algoritmo é que somente é garantido o término do mesmo nos participantes do poço, i.e., participantes fora do poço podem não terminar a execução deste protocolo, ficando bloqueados na espera do valor de decisão. De fato, mesmo que um participante fora do poço não termine este protocolo, tal participante é capaz de descobrir todos os participantes do poço, o que é suficiente para resolver o BFT-CUP e terminar. Um participante i que termina o algoritmo obtém uma visão parcial do sistema composta pelo conjunto maximal de participantes alcançáveis a partir i em G_{di}.

Notações. O algoritmo utiliza as seguintes notações:

1. $i.known$ – conjunto contendo os identificadores de todos os participantes conhecidos por i;
2. $i.received$ – conjunto contendo os identificadores dos participantes cujas respostas foram recebidas por i;
3. $i.msg\_pend$ – conjunto contendo os identificadores dos participantes que devem enviar uma mensagem (resposta) para i, i.e., para cada $j \in i.msg\_pend$, i deve receber uma mensagem de j;
4. $i.nei\_pend$ – conjunto de tuplas $\langle j, j.neighbor \rangle$, onde $j.neighbor$ contém os identificadores dos vizinhos de j. Esta tupla representa um participante j conhecido por i, onde i ainda não coletou informações suficientes para ter certeza que todos os participantes em $j.neighbor$ realmente existem.
5. $\#_{\langle *, \langle j \rangle \rangle} i.nei\_pend$ – número de vezes que um participante $j \in *.neighbor$ para qualquer tupla $\langle *, *.neighbor \rangle \in i.nei\_pend$.
6. $i.asked$ – conjunto global contendo os identificadores de participantes que solicitaram o valor de decisão para i. Este conjunto também é usado

no Algoritmo 4.

7. *i.decision* – variável global que armazena o valor de decisão. Esta variável também é usada no Algoritmo 4.

Algoritmo 2 DISCOVERY executado pelo participante *i*.

```
constant:
 1: f : int                                                  {upper bound on the number of failures}

variables:
 2: i.known : set of nodes                                   {known nodes}
 3: i.nei_pend : set of ⟨node, node.neighbor⟩ tuples         {i does not know all neighbors of node}
 4: i.msg_pend : set of nodes                                {nodes that i is waiting for messages (replies)}
 5: i.received : set of nodes                                {nodes that i has received a reply}
 6: i.asked : set of nodes                                   {nodes that have required the decision value (also used in Algorithm 4)}
 7: i.decision : value                                       {decision value (also used in Algorithm 4)}

message:
 8: GET_NEIGHBOR:
 9: SET_NEIGHBOR:
        neighbor : set of nodes                              {neighbors of the sending node}
10: SET_DECISION:
        decision : value                                     {the decided value}

Task MAIN
11: i.known, i.msg_pend ← {i} ∪ i.PD;
12: i.nei_pend, i.received, i.asked ← ∅;
13: Fork DELIVER();
14: reachable_send(GET_NEIGHBOR, i);

Task DELIVER
15: upon execution of reachable_deliver(GET_NEIGHBOR, p);
16: if i.decision = ⊥ then
17:     i.asked ← i.asked ∪ {p};
18: else
19:     send SET_DECISION(i.decision) to p; {Decision already been taken by i.}
20: end if
21: send SET_NEIGHBOR(i.PD) to p;

22: upon receipt of SET_NEIGHBOR(p.neighbor) from p
23: i.received ← i.received ∪ {p};
24: i.msg_pend ← i.msg_pend \ {p};
25: i.nei_pend ← i.nei_pend ∪ {⟨p, p.neighbor⟩};
26: for all j: (#⟨∗,⟨j⟩⟩ i.nei_pend > f) ∧ (j ∉ i.known) do
27:     i.known ← i.known ∪ {j};
28:     if {j} ∉ i.received then
29:         i.msg_pend ← i.msg_pend ∪ {j};
30:     end if
31: end for
32: for all ⟨j, j.neighbor⟩ ∈ i.nei_pend do
33:     if (∀z ∈ j.neighbor: z ∈ i.known) then
34:         i.nei_pend ← i.nei_pend \ {⟨j, j.neighbor⟩};
35:     end if
36: end for
37: if (|i.nei_pend| + |i.msg_pend|) ≤ f then
38:     return i.known;
39: end if
```

Descrição. Na inicialização deste algoritmo no participante i, os conjuntos $i.known$ e $i.msg\_pend$ são atualizados com o próprio i e seus vizinhos retornados por $i.PD$ (linha 11). Então, com a ajuda do Algoritmo 1, i difunde (*reliable broadcasts*) uma mensagem GET_NEIGHBOR requisitando informações sobre a composição do sistema para todos os participantes alcançáveis a partir de i em G_{di} (linha 14). A tarefa DELIVER é lançada na linha 13 para lidar com a entrega desta mensagem e difundir o valor decisão, caso o mesmo já tenha sido estabelecido.

Quando i entrega uma mensagem GET_NEIGHBOR de um participante p (linha 15), i responde com uma mensagem SET_NEIGHBOR, indicando seus vizinhos (linhas 21). Além disso, esta mensagem também representa uma solicitação do valor de decisão: caso i já conheça o valor de decisão, i envia este valor para p; caso contrário, i armazena o identificador de p em $i.asked$ e envia o valor de decisão para p assim que determinar tal valor (veja o Algoritmo 4).

Quando o participante i recebe uma resposta SET_NEIGHBOR de p (linha 22), i atualiza com p o conjunto de respostas recebidas, vizinhos pendentes e mensagens pendentes (linhas 23-25). O passo seguinte é verificar se i adquiriu conhecimento sobre um novo participante (linhas 26 - 31). Para garantir segurança, i descobre um novo participante j se pelo menos $f+1$ outros processos conhecidos informaram que j é um vizinho (linha 26). Após esta verificação, o conjunto de vizinhos pendentes é atualizado (linhas 32–36) de acordo com os novos participantes descobertos.

Para determinar se ainda existem participantes a serem descobertos, i utiliza os conjuntos $i.msg\_pend$ e $i.nei\_pend$, que armazenam as pendências relacionadas com as respostas recebidas por i. Assim, o algoritmo termina quando permanecerem no máximo f pendências (linhas 37- 39). A intuição por trás desta condição é que, considerando o padrão seguro de falhas bizantinas, se existem no máximo f pendências no processo i, então i já descobriu todos os processos alcançáveis a partir dele em G_{di}, pois existem pelo menos $k \geq f+1$ caminhos disjuntos nos nós, de i para cada participante alcançável a partir de i, formados apenas por nós corretos. Deste modo, o algoritmo termina retornando o conjunto de participantes descobertos por i, que contém todos os participantes (corretos ou faltosos) alcançáveis a partir de i em G_{di}.

Terminação de participantes fora do poço. Como mencionado, o Algoritmo 2 pode não terminar em participantes que não pertencem à componente poço G_{sink} de G_{di}. Considere dois nós p,q tal que $p,q \in G_{di}$, $p,q \notin G_{sink}$ e q é alcançável a partir de p em G_{di} (Definição 5). O fato de p alcançar q não implica que p também alcance todos os vizinhos de q de forma confiável, seguindo a Definição 5. Neste caso, algum vizinho de q pode não entregar a mensagem GET_NEIGHBOR de p. Então, é possível que q nunca seja re-

movido do conjunto $p.nei\_pend$, mesmo q sendo correto. Desta forma, o número de pendências em p pode nunca ser menor ou igual ao limiar f (linha 37). Felizmente, mesmo nestes casos, p pode aguardar pelo valor de decisão, que será recebido a partir dos participantes alcançáveis em G_{di} (pelo menos os participantes em G_{sink} – Observação 1).

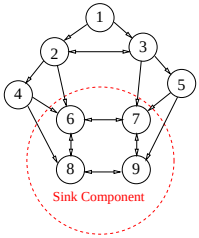


Figura 4 – 2-OSR sem falhas, onde o algoritmo DISCOVERY não termina no participante 1.

A Figura 4 apresenta um cenário para um PD 2-OSR ($f = 1$, com todos os participantes corretos), onde o Algoritmo 2 não termina no participante 1, que está fora do poço. Isto acontece pelo fato de que, apesar do participante 1 alcançar os participantes 2 e 3 (de fato, $\{2,3\} \in 1.PD$), 1 não alcança os participantes 4 (vizinho de 2) e 5 (vizinho de 3). Assim, os nós 4 e 5 nunca entregarão a mensagem GET_NEIGHBOR difundida a partir do participante 1. Desta forma, $\{2,3\}$ nunca é removido de $1.nei\_pend$ e, como $f = 1$, o algoritmo não termina em 1. Felizmente, isto não acontece com participantes do poço, visto que todos os participantes em G_{sink} são alcançáveis a partir de qualquer participantes em G_{di} (Observação 1) e pelas propriedades do PD k-OSR (Definição 3), os participantes em G_{sink} apenas possuem vizinhos que também pertencem à G_{sink}.

4.5.1.1 *Corretude*

O Algoritmo 2 satisfaz algumas propriedades expostas pelos lemas 4 e 5. Antes de apresentar as provas, faremos algumas observações importantes. **Obs 1**: Pela Observação 1 (Seção 4.4) e pelas propriedades de $\{G_{di} \setminus F\} \in k$-OSR, temos que: *(i)* todo participante $z \in G_{sink}$ é alcançável a partir de qualquer participante $p \in G_{di}$ através de pelo menos $f+1$ caminhos disjuntos nos nós formados apenas por participantes corretos (Definição 5); *(ii)* se $p \in G_{sink}$ então apenas nós em G_{sink} são alcançáveis a partir de p e *(iii)* qualquer $z \in G_{sink}$ (correto ou faltoso) é vizinho conhecido de pelo menos $f+1$ nós

corretos, assim, $z \in PD$ de pelo menos $f+1$ nós corretos.
Obs 2: O Algoritmo termina com a execução da linha 38 quando o número de pendências ($|i.nei\_pend| + |i.msg\_pend|$) não ultrapassar f (linha 37). O conjunto $i.msg\_pend$ contém os processos conhecidos por i, cujas respostas (SET_NEIGHBOR) ainda não foram recebidas. Então, $i.msg\_pend = i.known \setminus i.received$ (pelas linhas 11-12, 23-24, 27-29). O conjunto $i.nei\_pend$ contém tuplas de processos que cujas respostas foram recebidas por i, mas nem todos os seus vizinhos já são conhecidos por i. Então, $\forall j : \langle j, * \rangle \in i.nei\_pend, j \in i.received$ (pelas linhas 12, 23, 25, 34).
Obs 3: Um processo $j \in i.known$ se ($j \in i.PD$) (pela linha 11) ou ($j \in \bigcup q.PD$ e $|\bigcup q.PD| > f$, q é alcançável a partir de i) (pelas linhas 11, 14-15, 21–22, 26–27).

Lema 4 *(Participantes do Poço –* DISCOVERY*) O Algoritmo 2 (*DISCOVERY*) executado por um participante correto $i \in G_{sink}$ satisfaz as seguintes propriedades, considerando o modelo da Suposição 1:*

- *Terminação: i termina a execução;*
- *Exatidão: i retorna o conjunto $i.known = V_{sink}$.*

Prova. Pela Obs 1 e Corolário 1, se i executa reachable_send (GET_NEIGHBOR, i) (linha 14), então todo nó correto $q \in G_{sink}$, executa reachable_deliver (GET_NEIGHBOR, i) (line 15) e envia uma mensagem SET_NEIGHBOR (linha 21) que é recebida por i (linha 22), uma vez que os canais são confiáveis. Então, o seguinte cenário é atingido no final do algoritmo:

(A): Para cada participante correto $q \in G_{sink}$, $q \in i.received \wedge q \notin i.msg\_pend$ (pelas linhas 11-12, 23-24, 27-29).

(B): Para um dado participante malicioso (ou parado) $x \in G_{sink}$, x pode não responder com uma mensagem SET_NEIGHBOR para i. Então, $x \notin i.received \wedge x \in i.msg\_pend$.

Em ambas situações, (A) e (B), i recebe mensagens SET_NEIGHBOR (*neigh*) de pelo menos $f+1$ vizinhos corretos de x, q (participantes que conhecem estes processos), onde x,$q \in neigh$. Então, o predicado da linha 26 é satisfeito, e assim, todo $x, q \in i.known$ (linha 27) (Obs 2). Em consequência, o predicado da linha 33 é satisfeito para todo processo correto $j \in i.received$, isto é ($\forall z \in j.neighbor : z \in i.known$), e $\langle j, * \rangle$ é removido de $i.nei\_pend$ (linha 34).

(C): Para um participante malicioso $x \notin G_{sink}$ (não alcançável a partir de i) que de alguma forma descobre a presença de i no sistema – considere que x envia uma mensagem SET_NEIGHBOR ($x.neighbor$) para i indicando sua presença no sistema e/ou a presença de outros nós ($k \in x.neighbor, k \notin G_{sink}$). Neste caso, $x \in i.received$, $\langle x, x.neighbor \rangle \in i.nei\_pend$ (linhas 23–25), mas

$x,k \notin i.known$ e $x,k \notin i.msg\_pend$, uma vez que, pela Obs 3, o predicado da linha 26 é satisfeito somente para processos alcançáveis.

Por (A), (B), (C), no final do algoritmo, sabendo que $G_{sink} \subseteq G_{di} \subseteq \Pi$ é finito, e segue as observações Obs 1, 2, 3, conclui-se que:

1. $i.known = V_{sink}$ (satisfazendo *Exatidão*);
2. $V_{sink} \setminus F \supseteq i.received \subseteq V_{sink} \cup F$;
3. $i.msg\_pend \subseteq F$, $i.msg\_pend = i.known \setminus i.received$, $|i.known \setminus i.received| \leq f$, $|i.msg\_pend| \leq f$;
4. $i.nei\_pend \subseteq F$, $|i.nei\_pend| \leq f$.

Agora, provaremos que realmente $|i.nei\_pend| + |i.msg\_pend| \leq f$ (o predicado de terminação da linha 37). Note que, no final do algoritmo, $\forall j$ correto: $\langle j, * \rangle \notin i.nei\_pend$ e $j \notin i.msg\_pend$. Assim, $i.nei\_pend \cup i.msg\_pend \subseteq F$. Além disso, se $\langle j, * \rangle \in i.nei\_pend$ então $j \notin i.msg\_pend$ (pelas linhas 11-12, 23-24). Então, $|i.nei\_pend| + |i.msg\_pend| \leq f$, satisfazendo *Terminação*. Isto conclui nossa prova e este lema é válido. $\square_{Lema\ 4}$

Lema 5 *(Participantes Fora do Poço –* DISCOVERY*) O Algoritmo 2 (*DISCOVERY*) executado por um participante correto $i \notin G_{sink}$ satisfaz as seguintes propriedades, considerando o modelo da Suposição 1:*

- *Exatidão: Eventualmente, $i.known \supset V_{sink}$;*
- *Terminação Condicional: Se i termina o algoritmo, i retorna $i.known \supset V_{sink}$.*

Prova. Esta prova é derivada diretamente a partir do Lema 4, uma vez que todo participante $q \in G_{sink}$ é alcançável a partir de $i \notin G_{sink}$. Assim, no início da execução $i.known = \{i\} \cup i.PD$ (linha 11), então, eventualmente $i.known \supseteq \{i\} \cup V_{sink}$ antes da condição de terminação da linha 37 ser satisfeita. $\square_{Lema\ 5}$

4.5.2 2ª Fase: Determinação da Componente Poço

Na fase anterior é garantido que todos os participantes do poço terminam o algoritmo DISCOVERY, descobrindo todos os participantes em G_{sink}. Os participantes fora do poço podem ou não terminar aquele algoritmo. Neste sentido, uma fase intermediária é necessária para determinar quais participantes, destes que terminam a fase anterior, pertencem à G_{sink}. Evidentemente, um participante que não termina a fase anterior não pertence à G_{sink} (Lema 4).

Esta fase intermediária é representada pelo Algoritmo 3 (SINK), que é executado pelos participantes para determinar se pertencem ou não a componente G_{sink}. Este algoritmo explora o fato de que após terminar o algoritmo DISCOVERY, os membros de G_{sink} obtém a mesma visão parcial do sistema (exatamente os membros em G_{sink} - Lema 4), enquanto que os outros participantes obtêm necessariamente mais conhecimento sobre a composição do sistema, i.e., cada nó conhece pelo menos a si próprio mais os membros em G_{sink} (Lema 5).

Algoritmo 3 SINK executado pelo participante i

```
constant:
 1: f : int                                   {upper bound on the number of failures}

variables:
 2: i.known : set of nodes                                   {known nodes}
 3: i.nacked : set of nodes                  {not in the same component of i}
 4: i.acked : set of nodes                   {in the same component of i}
 5: i.in_the_sink : boolean                  {is i in the sink?}

message:
 6: REQUEST:
 7:     known : set of nodes
 8: RESPONSE:
 9:     ack/nack : boolean

** All Nodes **
Task MAIN
10: i.known ← DISCOVERY();
11: i.acked ← {i};
12: i.nacked, ← ∅;
13: for all j ∈ i.known: j ≠ i do
14:     send REQUEST(i.known) to j;
15: end for

16: upon receipt of REQUEST(p.known) from p
17: if i.known = p.known then
18:     send RESPONSE(ack) to p;
19: else
20:     send RESPONSE(nack) to p;
21: end if

22: upon receipt of RESPONSE(m) from p
23: if m = nack then
24:     i.nacked ← i.nacked ∪ {p};
25:     if |i.nacked| ≥ f + 1 then
26:         i.in_the_sink ← false;
27:         return ⟨i.in_the_sink, i.known⟩;
28:     end if
29: else
30:     i.acked ← i.acked ∪ {p};
31:     if |i.acked| ≥ |i.known| − f then
32:         i.in_the_sink ← true;
33:         return ⟨i.in_the_sink, i.known⟩;
34:     end if
35: end if
```

Notações. O algoritmo utiliza as seguintes notações:

1. *i.nacked* – conjunto contendo os identificadores dos participantes que não estão na mesma componente de *i* em G_{di};
2. *i.acked* – conjunto contendo os identificadores dos participantes que estão na mesma componente de *i* em G_{di};
3. *i.in_the_sink* – variável booleana que determina se *i* é um participante do poço.

Descrição. Na inicialização, o participante *i* executa o algoritmo DISCOVERY a fim de obter sua visão parcial sobre a composição do sistema (linha 10), onde os participantes fora do poço poderão não terminar aquele algoritmo enquanto os participantes do poço sempre terminam descobrindo exatamente todos os participantes de G_{sink}. Caso *i* terminar o algoritmo DISCOVERY, *i* envia a visão obtida para todos os participantes conhecidos (linha 14) a fim de determinar se pertence ou não a componente poço. Quando esta mensagem é entregue em algum participante *j*, *j* responde com um *ack* caso possua o mesmo conhecimento de *i* (i.e., *j* e *i* são membros da mesma componente). Caso contrário, *j* responde com um *nack* (linhas 16-21). É importante notar que *j* somente responde para *i* após completar o algoritmo DISCOVERY. Isto significa que pelo menos todos os participantes de G_{sink} enviam respostas.

Quando o participante *i* recebe uma resposta de um processo *p* (linha 22), duas situações são possíveis: (1) caso a resposta seja um *nack*, *i* adiciona *p* no conjunto *i.nacked* de processos pertencentes a outras componentes; caso o número de processos em *i.nacked* seja maior do que f, *i* conclui que não pertence à G_{sink} e retorna *false* (linhas 23–29). Esta condição é válida porque os participantes fora de G_{sink} conhecem pelo menos todos os participantes de G_{sink} (lemas 4 e 5). Caso contrário, (2) se a resposta for um *ack*, *i* adiciona *p* no conjunto *i.acked*. Então, se *i* recebeu respostas com *ack* de todos os processos conhecidos, excluindo-se os possíveis f faltosos (linha 31), *i* conclui que pertence a G_{sink}. Esta condição é válida porque processos membros da componente poço recebem respostas apenas de processos que pertencem a esta mesma componente. Além disso, em ambos os casos, um conluio de até f participantes maliciosos não é capaz de fazer um processos concluir erroneamente.

4.5.2.1 *Corretude*

Os lemas 6 e 7 definem as propriedades satisfeitas pelo Algoritmo 3.

Lema 6 *(Participantes do Poço –* SINK*) O Algoritmo 3 (*SINK*) executado por um participante correto $i \in G_{sink}$ satisfaz as seguintes propriedades, considerando o modelo da Suposição 1:*

- *Terminação: i termina a execução;*
- *Exatidão: i retorna* $\langle true, V_{sink} \rangle$.

Prova. Pelo Lema 4, cada processo correto $j \in G_{sink}$ termina o algoritmo DISCOVERY; além disso, $\forall j, j.known = V_{sink}$. Então, i recebe, como resposta de sua mensagem REQUEST (linha 14), mensagens RESPONSE (*ack*) (linha 22) de cada nó correto $j \in G_{sink}$, pois os canais são confiáveis. Mesmo com a ocorrência de um conluio de até f processos maliciosos que respondem com *nack*, no máximo $|i.nacked| \leq f$. Assim, o predicado da linha 25 nunca é satisfeito. Uma vez que o número de participantes corretos é pelo menos $|i.known| - f$, eventualmente o predicado da linha 31 é satisfeito e i retorna $\langle true, V_{sink} \rangle$, satisfazendo *Terminação* e *Exatidão*. $\square_{Lema\ 6}$

Lema 7 *(Participantes fora do Poço –* SINK*) O Algoritmo 3 (*SINK*) executado por um participante correto* $i \notin G_{sink}$ *satisfaz as seguintes propriedades, considerando o modelo da Suposição 1:*

- *Terminação Condicional: se i terminar o algoritmo* DISCOVERY*, então i também termina o algoritmo* SINK*;*
- *Exatidão: se i terminar o algoritmo, então i retorna* $\langle false, i.known \rangle$.

Prova. Caso i termine a execução do algoritmo DISCOVERY, i envia a mensagem REQUEST para todos os nós em $i.known$. Pelo Lema 5, $i.known \supset V_{sink}$. Então, cada processo correto $j \in G_{sink}$ receberá a mensagem REQUEST de i e responderá com uma mensagem RESPONSE (*nack*), pois $(j.known = V_{sink}) \neq i.known$, pelo Lemma 4, e pelos canais serem confiáveis. Pelas propriedades de $\{G_{di} \setminus F\} \in k$-OSR e pela Observação 1 (Seção 4.4), existem pelo menos $f+1$ processos corretos em G_{sink}. Assim, i receberá na linha 22 pelo menos $f+1$ respostas com *nack* e o predicado da linha 25 ($|i.nacked| \geq f+1$) eventualmente é satisfeito. Além disso, i nunca receberá um número de respostas RESPONSE (*ack*), tal que $|i.acked| \geq |i.known| - f$, mesmo com um conluio de até f processos maliciosos do poço que respondem *ack* para i, e então o predicado da linha 31 nunca é satisfeito. Assim, eventualmente i retorna $\langle false, i.known \rangle$, satisfazendo *Terminação Condicional* e *Exatidão*. $\square_{Lema\ 7}$

4.5.3 3ª Fase: Realização do Consenso

Esta é a última fase do protocolo para resolver o BFT-CUP. Como os participantes de G_{sink} compartilham a mesma visão parcial do sistema, a ideia deste algoritmo é fazer com que estes participantes executem um consenso

bizantino clássico e difundam o valor de decisão para os outros participantes do sistema. O Algoritmo 4 (CONSENSUS) apresenta este protocolo.

Suposição 2 *(Modelo de Consenso Bizantino) Para resolver o BFT-CUP, consideramos que existem pelo menos* $2f+1$ *participantes corretos em* G_{sink}.

Esta suposição é adotada pelo protocolo de consenso bizantino clássico, que exige pelo menos $3f+1$ processos para tolerar até f faltas maliciosas (CASTRO; LISKOV, 2002; MARTIN; ALVISI, 2006; ZIELINSKI, 2004).

Algoritmo 4 CONSENSUS executado pelo participante i

```
constant:
 1: f : int                                          {upper bound on the number of failures}

input:
 2: i.initial : value                                {proposal value (input)}

variables:
 3: i.in_the_sink : boolean                          {is i in the sink?}
 4: i.known : set of nodes                           {partial view of i}
 5: i.decision : value                               {decision value}
 6: i.asked : set of nodes                           {nodes requiring the decision}
 7: i.values : set of ⟨node, value⟩ tuples           {reported decisions}

message:
 8: SET_DECISION:
 9:     decision : value                             {the decided value}

** All Nodes **
Task MAIN
10: i.decision, i.in_the_sink ← ⊥;
11: i.values, i.known ← ∅;
12: i.asked ← Its value comes from Algorithm 2 (DISCOVERY)
13: Fork GATHER_DECISION;
14: (i.in_the_sink, i.known) ← SINK();

** Node In Sink **
15: if i.in_the_sink then
16:     Consensus.propose(i.initial)        {underlying Byzantine consensus with all p ∈ i.known}
17:     upon Consensus.decide(v)
18:     i.decision ← v;
19:     ∀j ∈ i.asked, send SET_DECISION(i.decision) to j;
20:     return i.decision;
21: end if

** Node Not In Sink **
Task GATHER_DECISION
22: upon receipt of SET_DECISION(v) from p
23: if i.decision = ⊥ then
24:     i.values ← i.values ∪ {⟨p, v⟩};
25:     if #⟨*,v⟩ i.values ≥ f + 1 then
26:         i.decision ← v;
27:         return i.decision;
28:     end if
29: end if
```

Notações. O algoritmo utiliza as seguintes notações:

- $i.known$ – conjunto contendo os identificadores de todos os processos conhecidos por i;

- $i.in\_the\_sink$ – variável booleana que determina se i pertence ou não a G_{sink};

- $i.asked$ – conjunto global que contém os identificadores dos participantes que solicitaram o valor de decisão para i. Este conjunto é formado durante a execução do Algoritmo 2 (DISCOVERY);

- $i.decision$ – variável global que armazena o valor de decisão;

- $i.values$ – conjunto de tuplas do tipo $\langle nodeid, value \rangle$;

- $\#_{\langle *,v \rangle} i.values$ – número de vezes que o valor de decisão v aparece em qualquer tupla $\langle *, v \rangle \in i.values$.

Descrição. Na inicialização, cada participante executa o algoritmo SINK (linha 14) para obter sua visão parcial sobre a composição do sistema e determinar se pertence ou não à componente poço. Note que um participante $i \notin G_{sink}$ pode ficar bloqueado no algoritmo DISCOVERY (Lema 7). De qualquer forma, i aguardará pelo valor de decisão através da execução da tarefa GATHER_DECISION, que é lançada na linha 13. Caso $i \in G_{sink}$, i termina o algoritmo SINK (Lema 6) e, então, executa o algoritmo CONSENSUS. Desta forma, dependendo do resultado de um participante i pertencer ou não a G_{sink}, dois comportamentos distintos são possíveis:

1. Caso $i \in G_{sink}$, então i executa um consenso bizantino clássico (linha 16) com os participantes de sua visão. Quando uma decisão for estabelecida, i envia esta decisão para todos os nós que solicitaram este valor (linha 19). Estes nós estão armazenados em $i.asked$ e foram identificados no algoritmo DISCOVERY. Note que, durante a execução do algoritmo DISCOVERY, a decisão pode já ter sido estabelecida, e, neste caso, na execução da tarefa DELIVER, i envia esta decisão diretamente para o processo solicitante.

2. Caso $i \notin G_{sink}$, então i não participa do consenso bizantino clássico. Durante a execução do algoritmo DISCOVERY, i enviou mensagens solicitando o valor de decisão para todos os participantes alcançáveis a partir de i em G_{di}. Uma vez que todos os participantes de G_{sink} são alcançáveis a partir de i, é garantido que cada processo correto $j \in G_{sink}$ envia uma mensagem SET_DECISION($j.decision$) para i, quando a decisão for estabelecida, pois $i \in j.asked$. O participante i decide por um

valor v somente após receber v de pelo menos $f+1$ outros participantes, garantindo que v foi obtido de pelo menos 1 participante correto (linhas 22-29).

4.5.3.1 *Corretude*

O Teorema 1 mostra que o Algoritmo 4 resolve o BFT-CUP. Porém, antes de apresentar o teorema, vamos fazer a seguinte observação.
Obs 4: Todo processo correto $j \in G_{di}$ é tal que $j \in i.asked$ se j conseguir se comunicar com $i \in G_{sink}$ antes da decisão ser definida na linha 18 do algoritmo CONSENSUS. Pela Observação 1 (Seção 4.4) e pelo Corolário 1, toda mensagem difundida (*reachable broadcast*) por j é entregue por i. Deste modo, como a tarefa DELIVER do algoritmo DISCOVERY sempre permanece executando, assim que a mensagem de j for entregue em i, i adiciona j em $i.asked$ (linhas 15-17 do algoritmo DISCOVERY).

Teorema 1 *O Algoritmo 4 (*CONSENSUS*) resolve o BFT-CUP, considerando os modelos das Suposições 1 e 2.*

Prova. Dependendo de um participante i pertencer ou não a G_{sink}, dois comportamentos distintos são possíveis:

1. Caso $i \in G_{sink}$: Na execução do algoritmo SINK (linha 14), i obtém $\langle true, V_{sink} \rangle$ (Lema 6). Então, i executa um protocolo subjacente de consenso bizantino clássico (linha 16) com os nós em V_{sink}. Pela Suposição 2, V_{sink} contém pelo menos $2f+1$ participantes corretos, então todas as propriedades do consenso bizantino subjacente (CASTRO; LISKOV, 2002; MARTIN; ALVISI, 2006; ZIELINSKI, 2004) são satisfeitas, i.e., *Validade*, *Integridade*, *Acordo* e *Terminação*. Assim, i eventualmente executa a linha 17 e decide. O valor de decisão é então enviado para todos os processos em $i.asked$ (linha 19). Finalmente, o valor de decisão é retornado para a aplicação (linha 20). Pela Obs 4, para todo participante correto $j \in G_{di}$ que se comunicou com i, temos que $j \in i.asked$. No entanto, devido aos assincronismos, caso as mensagens de $j \in G_{di}$ ainda não tiverem chegado em i quando i está executando o algoritmo CONSENSUS, então temos que $j \notin i.asked$. Neste caso, uma vez que a tarefa DELIVER do algoritmo DISCOVERY sempre permanece executando, assim que estas mensagens cheguem em i, o mesmo envia o valor de decisão diretamente para j (linha 19 do algoritmo DISCOVERY).

2. Caso $i \notin G_{sink}$: Na execução do algoritmo SINK (linha 14), i pode fi-

car bloqueado ou obter $\langle false, i.known \rangle$ (Lema 7). Em qualquer caso, i não participa do protocolo de consenso bizantino clássico (linha 15). No entanto, i aguardará pelo valor de decisão através da execução da tarefa GATHER_DECISION (linhas 22-29) lançada na linha 13. Quando a mensagem SET_DECISION(v) é entregue em i (linha 22), caso i ainda não tenha decidido, i armazena v no conjunto $i.values$ (linha 24) e, assim que $f+1$ valores iguais forem recebidos (linha 25), i retorna o valor de decisão para a aplicação (linha 27). Pelo comportamento dos participantes em G_{sink} (descritos no caso anterior), existem pelo menos $2f+1$ participantes corretos em G_{sink} que enviam o valor de decisão (quando o mesmo for estabelecido) para i, pela execução tanto da linha 19 do algoritmo CONSENSUS quanto da linha 19 do algoritmo DISCOVERY (tarefa DELIVER). Isto garante que pelo menos $f+1$ mensagens com o mesmo valor de decisão são entregues em i, eventualmente satisfazendo o predicado da linha 25. Esta condição evita que um conluio de até f participantes maliciosos faça com que i decida por um valor incorreto. Desta forma, quando o predicado da linha 25 for satisfeito, i retorna o valor de decisão para a aplicação (linhas 26-27).

Deste modo, em qualquer um dos casos, as propriedades do consenso são garantidas pelo algoritmo CONSENSUS. $\square_{Teorema\ 1}$

O Teorema 1 prova que o PD k-OSR e o padrão seguro de falhas bizantinas são suficientes para resolver o BFT-CUP.

4.5.4 Necessidade de PD k-OSR e Padrão Seguro de Falhas Bizantinas para Resolver o BFT-CUP

Utilizando um PD k-OSR e o padrão seguro de falhas bizantinas, o protocolo apresentado nas seções anteriores resolve o BFT-CUP (Teorema 1). Esta seção apresenta a prova de que estas suposições também são necessárias para resolver este problema (Teorema 2).

Lema 8 *Para resolver o BFT-CUP em um sistema assíncrono, estendido com um PD, o DAG (directed acyclic graph) obtido pela redução de G_{di}, induzido por PD, às suas componentes fortemente conexas deve ter uma e apenas uma componente poço.*

Prova. Seja $Dag(G_{di})$ o DAG obtido pela redução de $G_{di} = (V,E)$ às suas componentes fortemente conexas. Assuma, por contradição, que $Dag(G_{di})$ obtido de $G_{di} \in \mathsf{PD}$ possui mais do que um poço e existe um protocolo $\mathscr{A}$

que resolve o BFT-CUP em um sistema assíncrono. Agora, mostraremos que $\mathscr{A}$ admite uma execução que viola a propriedade de *acordo* do consenso.

Considere o sistema $\mathscr{X}$ onde $Dag(G_{di})$ possui mais de um poço. Sejam $G_1 = (V_1, E_1)$ e $G_2 = (V_2, E_2)$ duas destas componentes poço. Considere que todos os participantes em G_1 têm valor de proposição v_1 enquanto que todos os participantes em G_2 têm valor v_2, com $v_1 \neq v_2$. Seja um sistema $\mathscr{X}_1$, derivado de $\mathscr{X}$, composto apenas por processos em G_1 que possuem valor de proposição v_1. Considere uma execução e_1 de $\mathscr{A}$ para $\mathscr{X}_1$. Pela propriedade de terminação, processos em G_1 eventualmente decidem em um tempo t_1. Pela propriedade de validade, estes processos decidem pelo valor v_1. Similarmente, é possível cronstruir um sistema $\mathscr{X}_2$, derivado de $\mathscr{X}$, composto apenas por processos em G_2 que possuem valor de proposição v_2. Considere uma execução e_2 de $\mathscr{A}$ para $\mathscr{X}_2$. Pela propriedade de terminação, processos em G_2 eventualmente decidem em um tempo t_2. Pela propriedade de validade, estes processos decidem pelo valor v_2.

Considere agora o sistema original $\mathscr{X}$ contendo todos os processos de G_1 e G_2. Pelas suposições, a cardinalidade n de processos no sistema não é conhecida e o único conhecimento fornecido aos processos é através de PD e representado por G_{di}. Seja $K_1 = \bigcup_{i \in V_1} i.\mathsf{PD}$ o conjunto de processos conhecidos por todos os processos de G_1. Similarmente, seja $K_2 = \bigcup_{j \in V_2} j.\mathsf{PD}$ o conjunto de processos conhecidos por todos os processos de G_2. Pelas propriedades dos grafos, não existe nenhuma aresta saindo de algum processo de uma componente poço para um outro processo de outra componente poço. Assim, $K_1 \subseteq V_1$ e $K_2 \subseteq V_2$ e os processos em $G_1(G_2)$ não conhecem os outros processos de G_{di} (incluindo os processos em outras componentes opço). Como o sistema é assíncrono, considere que um processo i fora do poço, $i \in \{V \setminus \{V_1 \cup V_2\}\}$, não executa nenhum processamento até o tempo $t = max\{t_1, t_2\}$; ou, alternativamente, caso i envie uma mensagem para um processo $j \in \{V_1 \cup V_2\}$, tal mensagem é entregue após o tempo t.

Claramente, é possível ocorrer uma execução e do algoritmo $\mathscr{A}$ no sistema $\mathscr{X}$ onde os processos de G_1 evoluem exatamente como na execução e_1 do sistema $\mathscr{X}_1$ até o tempo t. Então, na execução e, processos em G_1 decidem por v_1 no tempo $t_1 \leq t$. Similarmente, na mesma execução e do algoritmo $\mathscr{A}$ no sistema $\mathscr{X}$, os processos em G_2 podem evoluir exatamente como na execução e_2 do sistema $\mathscr{X}_2$ até o tempo t. Então, na execução e, processos em G_2 decidem por v_2 no tempo $t_2 \leq t$. Como os processos em G_1 decidem por v_1 e os processos em G_2 decidem por v_2, $v_1 \neq v_2$, a propriedade de *acordo* é violada na execução e, chegando-se assim a uma contradição que $\mathscr{A}$ resolve o BFT-CUP no sistema $\mathscr{X}$. $\square_{Lema\ 8}$

Lema 9 *Para resolver o BFT-CUP em um sistema assíncrono, estendido com um* PD*, o grafo não orientado de conectividade por conhecimento G obtido*

de $G_{di} \in$ PD deve ser conectado.

Prova. A prova deste lema é derivada diretamente da prova do Lema 8, uma vez que se G não é conectado, então existem pelo menos duas componentes poço no DAG obtido pela redução de G_{di} às suas componentes fortemente conexas. $\square_{Lema\ 9}$

Corolário 2 *Para resolver o BFT-CUP em um sistema assíncrono, estendido com um PD, um processo fora da componente poço deve aguardar pelo valor de decisão enviado pelos processos da componente poço.*

Prova. A prova deste lema é derivada diretamente da prova do Lema 8, uma vez que processos da componente poço devem obter uma decisão unilateralmente. $\square_{Corolário\ 2}$

Lema 10 *Para resolver o BFT-CUP em um sistema assíncrono, estendido com um PD, a componente poço obtida pela redução de G_{di} às suas componentes fortemente conexas deve possuir pelo menos $2f+1$ participantes corretos.*

Prova. Pelo Corolário 2, as decisões devem ser obtidas na componente poço. Adicionalmente, $2f+1$ é o número mínimo de participantes corretos necessários para executar um consenso bizantino clássico (CASTRO; LISKOV, 2002; MARTIN; ALVISI, 2006). $\square_{Lema\ 10}$

Observação 2 *Seguindo os resultados de Dolev (DOLEV, 1982), em uma rede assíncrona onde os participantes são desconhecidos, o número de faltas maliciosas deve ser menor do que a metade do grau de conectividade k para que os processos possam comunicar-se corretamente, i.e., para garantir autenticidade. Além disso, sem autenticidade não é possível tolerar processos maliciosos em sistemas assíncronos, pois um processo faltoso pode se fazer passar por outros processos para algum outro processo vítima.*

Lema 11 *O PD k-OSR e o padrão seguro de falhas bizantinas são necessários para resolver o BFT-CUP em um sistema assíncrono com participantes desconhecidos, apesar de $f < k$ faltas bizantinas.*

Prova. Seja G_{di} o grafo de conectividade por conhecimento induzido por PD e seja $G_{sink} = (V_{sink}, E_{sink})$ a única componente poço de G_{di}. O padrão seguro de falhas bizantinas estabelece que $G_{di} \setminus F \in k$-OSR, $k \geq f+1$. Sabendo que F é o conjunto de participantes em G_{di} que podem falhar, $|F| \leq f$. Isso garante que dado qualquer conjunto de falhas, $G_{di} \setminus F$ satisfas as propriedades de k-OSR, $k \geq f+1$. Como resultado, existem pelo menos $k \geq f+1$

caminhos disjuntos nos nós compostos apenas por processos corretos entre os processos de G_{sink} e também entre um processo fora de G_{sink} para um processo pertencente a esta componente.

Agora, assuma por contradição que existe um algoritmo $\mathscr{A}$ que resolve o BFT-CUP com um *PD* mais fraco do que o k-OSR ou que não necessite do padrão seguro de falhas bizantinas. Os quatro seguintes cenários são possíveis: ou (1) o grafo não direcionado G obtido de $G_{di} \setminus F$ não é conectado; ou (2) o DAG obtido pela decomposição de $G_{di} \setminus F$ às suas componentes fortemente conexas possui mais do que um poço; ou (3) a única componente poço G_{sink} de $G_{di} \setminus F$, não é k-fortemente conexa, com $k \geq f+1$, e então existem menos do que $f+1$ caminhos disjuntos nos nós, formados apenas por processos corretos, entre os participantes; ou (4) existem dois participantes i, j, tal que $i \notin G_{sink}$ e $j \in G_{sink}$, e existem menos do que $f+1$ caminhos disjuntos nos nós de i para j em $G_{di} \setminus F$.

Cenário (1): Conectividade é uma condição necessária para resolver o BFT-CUP (Lema 9).

Cenário (2): Uma única componente poço é condição necessária para resolver o BFT-CUP (Lema 8).

Cenário (3): Pela Observação 2, G_{sink} não possui conectividade suficiente e a presença de f participantes maliciosos pode dividir G_{sink} em pelo menos duas componentes de G_{di}, G_1 e G_2, de forma que nenhuma mensagem dos processos em G_1 (respectivamente, G_2) podem ser autenticadas pelos processos em G_2 (respectivamente, G_1). Neste caso, os participantes acreditam que G_{di} possui pelo menos duas componentes poço: G_1 e G_2. Pelo Lema 8, uma única componente poço é necessária.

Cenário (4): caso existam menos do que $f+1$ caminhos disjuntos nos nós de $i \notin G_{sink}$ para $j \in G_{sink}$ em $G_{di} \setminus F$, então i pode alcançar no máximo f participantes de G_{sink} através de $f+1$ caminhos disjuntos nos nós corretos, uma vez que G_{sink} é k-fortemente conexo, com $k \geq f+1$ caminhos disjuntos nos nós corretos. Seja $R \subset V_{sink}$ o conjunto composto por estes f processos. Então, $|R| \leq f$ e $j \notin R$. Pela Observação 2, o comportamento malicioso de f participantes de G_{di} pode impedir a autenticação das mensagem de i para j. Desta forma, i não consegue certificar-se da presença de j, e vice-versa, pois n não é conhecido e o sistema é assíncrono. Por contrução, nenhum processo em $\{V_{sink} \setminus R\}$ consegue autenticar as mensagens de i e então não conhecem i. Pelo Corolário 2, i deve aguardar pelo valor de decisão enviado pelos processos de G_{sink}. Além disso, como o sistema está sujeito a faltas bizantinas, i deve aguardar por pelo menos $f+1$ valores de decisão iguais, a fim de preservar as propriedades de segurança (acordo). No entanto, processos em $\{V_{sink} \setminus R\}$ nunca enviarão o valor de decisão para i e, como $|R| \leq f$, i receberá no máximo f mensagens, compromentendo a propriedade de terminação do

BFT-CUP.

Pelos cenários 1,2,3 e 4, chegamos a uma contradição. $\square_{Lema\ 11}$

Teorema 2 *O PD k-OSR e o padrão seguro de falhas bizantinas são necessários para resolver o BFT-CUP em um sistema assíncrono com participantes desconhecidos, apesar de $f < k$ faltas bizantinas. Além disso, a única componente poço obtida de $G_{di} \in PD$ deve possuir pelo menos $2f+1$ participantes corretos.*

Prova. A prova deste teorema é derivada diretamente a partir dos lemas 10 e 11. $\square_{Teorema\ 2}$

4.6 SEPARANDO FALHAS DE SAÍDAS DO SISTEMA

O modelo adotado neste estudo, bem como o utilizado nas outras soluções para o consenso entre participantes desconhecidos (CAVIN et al., 2005; GREVE; TIXEUIL, 2007; ALCHIERI et al., 2008a), suporta a mobilidade dos nós, mas não é forte o suficiente para tolerar um arbitrário *churn* (chegada e saída de processos) durante a execução do protocolo. Isto acontece porque, após o estabelecimento das relações de conhecimento (primeira fase do protocolo), novos participantes apenas serão considerados em execuções futuras do consenso. De fato, estes protocolos empregam o modelo de chegada finita de participantes (M_1 - Tabela 1).

Em nossos algoritmos, a saída de processos do sistema é considerada como uma falta. No entanto, esta não é a melhor abordagem uma vez que nossos protocolos toleram faltas bizantinas e a saída de um processo do sistema é semelhante a uma falta simples de parada (*crash*). Uma solução alternativa consiste em estender o padrão seguro de falhas bizantinas para prever saídas de processos, separando assim este comportamento de uma falha maliciosa. Neste sentido, o grau de conectividade no grafo de conhecimento deve ser suficiente para tolerar faltas maliciosas e também saídas de participantes do sistema.

Definição 6 ***(Padrão Seguro Estendido de Falhas Bizantinas).*** *Considere uma grafo de conectividade por conhecimento G_{di} induzido por um PD k-OSR e um conjunto L de participantes em G_{di} cujas saídas do sistema são permitidas. Seja f o número de participantes em G_{di} que podem falhar e F o conjunto de participantes em G_{di} cujas falhas são permitidas, então $|F| \leq f$. O padrão seguro estendido de falhas bizantinas especifica que $\{G_{di} \setminus \{L \cup F\}\} \in$ k-OSR, com $k \geq f+1$.*

O padrão estendido especifica que o grau de conectividade deve ser

pelo menos $k \geq f+1$ caminhos disjuntos nos nós formados apenas por participantes corretos que não deixam o sistema. No entanto, esta não é a única mudança para tolerar saídas de participantes, pois com este comportamento devemos especificar a necessidade da componente poço G_{sink} permanecer com uma quantidade de participantes suficiente para executar um consenso bizantino clássico. Neste sentido, o número de participantes em G_{sink} deve ser $n_{sink} \geq 3f+2l+1$ para tolerar até f faltas bizantinas e até l saídas. O parâmetro l não precisa ser previamente conhecido pelos processos do sistema, i.e., l é apenas um número que representa a quantidade de participantes do poço cujas saídas são permitidas e não comprometem as propriedades do sistema.

A seguir, é apresentada uma explicação sobre a necessidade de $n_{sink} \geq 3f+2l+1$ participantes em G_{sink} para tolerar até f faltas bizantinas e até l saídas. Considere o protocolo de consenso clássico *Paxos Bizantino* (CASTRO; LISKOV, 2002; ZIELINSKI, 2004), que requer $n \geq 3f+1$ processos para tolerar até f falhas e utiliza quóruns de $\frac{n+f}{2}$ processos em cada fase do protocolo, aguardando por mensagens destes quóruns. Então, temos o seguinte cenário na execução do consenso clássico pelo participantes de G_{sink}: este protocolo utiliza quóruns de $\frac{n_{sink}+f}{2}$ participantes do poço; como $n_{sink} \geq 3f+2l+1$, temos quóruns de tamanho $qsize \geq \frac{3f+2l+1+f}{2}$ participantes, que é equivalente a $qsize \geq 2f+l+1$; para o sistema evoluir é necessário processar mensagens de pelo menos $qsize$ participantes, então o número de participantes que podem sair do sistema é $n_{sink} - qsize$, que é equivalente a $((3f+2l+1)-(2f+l+1)) = f+l$. Neste sentido, o sistema permanece correto enquanto apresentar no máximo f participantes maliciosos no poço e progredirá enquanto no máximo l participantes de poço deixarem o mesmo. Este resultado também é aplicável para quóruns de $\frac{n+3f}{2}$ processos usados no algoritmo de consenso *Paxos at War* (ZIELINSKI, 2004) para decisões rápidas.

4.7 DISCUSSÃO

Esta seção discute alguns aspectos relacionados com o modelo de consenso apresentado, bem como da solução proposta para este problema.

4.7.1 Assinaturas Digitais

O grau de conectividade necessário (mínimo) para resolver o BFT-CUP, representado pelo PD k-OSR e pelo padrão seguro de falhas bizantinas,

permanece válido mesmo usando assinaturas digitais nos protocolos. Com a utilização de assinaturas digitais, é possível que um participante p envie uma mensagem para outro participante q desde que exista pelo menos 1 caminho de p para q formado apenas por processos corretos. No entanto, mesmo utilizando assinaturas digitais, o padrão seguro de faltas bizantinas continua sendo necessário para uma correta descoberta dos participantes presentes no sistema (protocolo DISCOVERY).

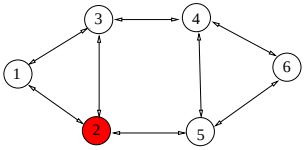


Figura 5 – 2-OSR com o participante 2 faltoso.

De fato, caso o padrão seguro de falhas bizantinas não seja verificado no sistema, um participante malicioso pode fazer com que um participante correto p não descubra todos os nós alcançáveis a partir de p em G_{di}, o que impossibilita a solução do BFT-CUP (a visão parcial obtida por p será inconsistente - Teorema 2). Por exemplo, a Figura 5 apresenta um grafo de conectividade por conhecimento induzido por um PD 2-OSR ($k = 2$), onde o participante 2 é malicioso e o padrão seguro de falhas bizantinas não é verificado para $f = 1$, uma vez que este padrão especifica que $k \geq f + 1$ caminhos formados apenas por processos corretos (Definição 4) e neste exemplo temos $k = 1$. Considere que o participante 1 está executando a fase DISCOVERY. Neste cenário, o participante malicioso 2 pode informar para o participante 1 que conhece apenas o participante 3. Após receber esta informação, o participante 1 irá interromper sua busca por novos participantes, pois o mesmo está aguardando apenas uma mensagem do participante 3 (o participante 1 não pode aguardar por esta mensagem, pois o mesmo não sabe se o participante 3 é correto). Assim, o participante 1 obtém uma visão parcial inconsistente do sistema $\{1,2,3\}$, pois a mesma deveria ser $\{1,2,3,4,5,6\}$.

4.7.2 Limitações do Protocolo

O modelo adotado neste estudo, bem como o utilizado nas soluções para o FT-CUP (CAVIN et al., 2005; GREVE; TIXEUIL, 2007), suporta a mobilidade dos nós, mas não é forte o suficiente para tolerar um arbitrário *churn* (chegada e saída de processos) durante a execução do protocolo. Isto

acontece porque, após o estabelecimento das relações de conhecimento (primeira fase do protocolo), novos participantes apenas serão considerados em execuções futuras do consenso. De fato, estes protocolos empregam o modelo de chegadas finitas de participantes (M_1 - Tabela 1).

Outra limitação destes modelos, quando consideramos um sistema assíncrono, é que os participantes da componente poço G_{sink} de G_{di} não conhecem a composição completa do sistema, pois estes participantes apenas descobrem os nós desta componente. Desta forma, é necessário que estes participantes guardem o valor de decisão enquanto o mesmo for útil para a aplicação, pois não é possível saber quando algum processo solicitará este valor. De fato, a tarefa DELIVER do algoritmo DISCOVERY deve permanecer ativa para processamento destas solicitações.

4.7.3 Outros Detectores de Participação

Apesar do PD k-OSR adotado neste trabalho ser o detector de participação mais fraco capaz de resolver o FT-CUP, existe outros detectores de participação mais fortes que também resolvem este problema (CAVIN et al., 2004; GREVE; TIXEUIL, 2007; ALCHIERI et al., 2008a):

Definição 7 *(**PD k-Redutível a Único Poço segundo (GREVE; TIXEUIL, 2007)**). A definição deste PD é muito semelhante à adotada neste trabalho (Definição 3). A diferença fundamental é que na definição apresentada em (GREVE; TIXEUIL, 2007), os participantes de fora do poço também são agrupados em componentes k-fortemente conexas.*

Definição 8 *(**PD k-Fortamente Conexo (k-Strong Connectivity – k-SCO)**). O grafo de conectividade por conhecimento $G_{di} = (V, \xi)$ induzido pelo oráculo PD é k-fortemente conexo.*

Definição 9 *(**PD Completamente Conexo (Full Connectivity – FCO)**). O grafo de conectividade por conhecimento $G_{di} = (V, \xi)$ induzido pelo oráculo PD é de tal modo que $\forall p, q \in \Pi$, temos $(p, q) \in \xi$.*

Uma característica comum a todos os detectores de participação que resolvem o BFT-CUP (com exceção do PD FCO que é completamente conexo) é o grau de conectividade k, o qual possibilita o correto funcionamento do protocolo mesmo na presença de faltas.

Através do PD k-OSR definido em (GREVE; TIXEUIL, 2007), é possível resolver o BFT-CUP com o mesmo algoritmo apresentado neste trabalho, com a diferença de que até mesmo os participantes fora do poço terminam todas as fases do protocolo. Por outro lado, através dos detectores de

participação FCO e k-SCO, a visão parcial obtida por cada processo contém exatamente todos os processos do sistema (primeira fase do protocolo). Depois disso, o problema do consenso é trivialmente resolvido usando qualquer protocolo de consenso bizantino clássico, pois todos os processos obtêm a mesma visão (completa) do sistema.

Vale ressaltar que o PD adotado neste trabalho (Definição 3) é o detector de participação capaz de resolver o BFT-CUP que apresenta a menor necessidade de conhecimento (conectividade no grafo de conhecimento) entre os participantes do sistema.

4.8 SIMULAÇÕES

Alguns trabalhos da literatura buscam analisar analiticamente o grau de conectividade apresentado por uma rede MANET em função de suas configurações (BETTSTETTER, 2002; SANTI, 2005). Estes trabalhos buscam soluções para controlar a topologia da rede e podem ser empregados na obtenção de uma rede com características (principalmente relacionadas com o grau de conectividade) que possibilitem a solução do BFT-CUP. Também encontramos alguns trabalhos que buscam analisar o comportamento do FT-CUP (consenso tolerante a faltas por *crash*) (COSTA; GREVE, 2007; COSTA et al., 2009) através de simulações em uma rede MANET.

Neste sentido, buscando complementar os resultados teóricos obtidos até então sobre consenso entre participantes desconhecidos, esta seção discute uma abordagem prática sobre a realização do BFT-CUP, apresentada em (ALCHIERI et al., 2011; TOMELIN et al., 20010). Primeiramente, o comportamento dos protocolos que resolvem o BFT-CUP é avaliado em uma rede MANET (*Mobile Ad Hoc Networks*) de topologia arbitrária, onde o grau de conhecimento teoricamente necessário para resolver este problema nem sempre é obtido pelos participantes. Nossas avaliações são baseadas em simulações onde os parâmetros tanto do BFT-CUP quando da rede MANET são variados, possibilitando verificar quais são os parâmetros e configurações necessários para que os participantes do sistema consigam resolver o BFT-CUP com uma alta probabilidade. Por fim, também é apresentado um estudo acerca da utilização do BFT-CUP em redes VANETs (*Veicular Ad Hoc Networks*), através de uma análise analítica sobre a conectividade veicular apresentada pela cidade de Porto-Portugal.

4.8.1 BFT-CUP em Redes MANETs

Esta seção busca analisar, através de uma série de simulações, o comportamento dos protocolos do BFT-CUP quando executados em MANETs. Para isso, tais protocolos foram implementados e simulados no *Jist/SWANS*. O *Jist* (*Java in Simulation Time*) (BARR et al., 2004) é um simulador de alta performance baseado em eventos discretos que executa sobre uma máquina virtual *Java*. Já o *SWANS* (*Scalable Wireless Ad hoc Network Simulator*) (BARR et al., 2005) é um módulo para simulação de MANETs construído sobre o *Jist*, sendo portanto também implementado em *Java*.

4.8.1.1 Implementação dos Detectores de Participação

Como apresentado antes neste capítulo, os detectores de participação são abstrações que fornecem conhecimento sobre a composição do sistema, sendo que para a solução do BFT-CUP o mínimo exigido é o detector de participação da classe k-OSR (Seção 4.2.2). No entanto, não existe uma proposta de implementação para um detector de participação que garanta os atributos definidos para esta classe em uma rede MANET. A possibilidade de existência de tal implementação continua sendo uma questão em aberto (COSTA et al., 2009). Deste modo, em nossas simulações utilizamos o detector de participação **PD-1hop** (COSTA et al., 2009) e buscamos avaliar a possibilidade de solução do BFT-CUP em um cenário onde o detector de participação não garante as propriedades do PD k-OSR. Note que, caso o detector de participação garantisse as propriedades do PD k-OSR, então o BFT-CUP sempre admitiria solução, i.e., a conectividade por conhecimento fornecida pelo detector sempre seria suficiente para resolver este problema.

O detector de participação **PD-1hop** é muito simples e retorna para os processos uma lista contendo os nós que se encontram no seu alcance de transmissão (nós vizinhos). Sempre que este detector é iniciado em algum processo, o mesmo envia mensagens de "*hello*" através de um *broadcast* local, por meio da camada MAC da rede MANET, que atinge todos os seus vizinhos dentro de 1 salto. Ao receber estas mensagens, cada vizinho i adiciona a fonte (emissor) a sua lista de nós conhecidos, que é retornada em $i.PD$.

Esse procedimento de descoberta inicial termina quando um tempo é atingido. Durante este período, os processos podem difundir novas mensagens de "*hello*". Este detector é implementado de forma independente dos parâmetros do BFT-CUP e é ativado no início de cada simulação para retornar uma aproximação da vizinhança real na rede.

4.8.1.2 Configurações das Simulações

As configurações gerais utilizadas nas simulações foram adotadas com base nos trabalhos de Costa *et al.* (COSTA; GREVE, 2007; COSTA et al., 2009). Estas configurações podem ser divididas em configurações da rede MANET (Tabela 4) e configurações do BFT-CUP (Tabela 5).

Os principais parâmetros da rede MANET variados nas simulações e que influenciam na possibilidade de solução do BFT-CUP são: *(1)* o alcance de transmissão e *(2)* a densidade de nós no sistema (quantidade de nós). Estes dois parâmetros estão diretamente relacionados com o grau de conhecimento que os participantes obtêm sobre a composição do sistema, i.e., com a conectividade por conhecimento apresentada em G_{di}, que é o fator determinante para a possibilidade de solução do BFT-CUP.

Parâmetros das Simulações (Simulador)	
Simulador	*Jist/SWANS*
Quantidade de nós	de 0 à 50
Área	500 m *X* 500 m
Alcance *wireless*	de 0m à 300m
Repetições	10
Padrão de Mobilidade	*Randon WayPoint*
Velocidade dos nós	de 0m/s à 5m/s
Tempo de pausa	de até 2s

Tabela 4 – Configurações da rede MANET.

Parâmetros das Simulações (BFT-CUP)	
Tempo de PD	60s
Tempo para reenvio de "*hello*"	de até 20s
Número máximo de faltas suportadas	de 1 à 10

Tabela 5 – Configurações do BFT-CUP.

Os detectores de participação utilizados nestas simulações foram ativados durante 60s antes de cada execução do BFT-CUP. Durante este período, cada participante esperou um tempo aleatório de até 20s para reenviar a mensagem de "*hello*" indicando sua presença no sistema (Seção 4.8.1.1). Vale ressaltar que estes parâmetros fizeram com que cada participante enviasse em média de 3 a 4 destas mensagens.

Além disso, também vale destacar que apesar do sistema estar configurado para suportar um determinado número de faltas, que varia de acordo com as configurações utilizadas em determinada simulação (de 1 até 10 fal-

tas), não foram introduzidos nós faltosos no sistema. Simulações adicionais mostraram que a inclusão de nós faltosos no sistema, desde que respeitando-se o limite de até f nós, não altera os resultados apresentados nesta seção.

Por fim, o tempo de término das simulações foi definido como sendo o instante de tempo em que todos os eventos produzidos pelos nós foram processados pelo simulador, i.e., as simulações permaneceram em execução enquanto tinham eventos a serem processados.

4.8.1.3 Métricas Avaliadas

Estas simulações têm por objetivo verificar em quais condições os participantes do sistema conseguem adquirir um conhecimento suficiente para resolver o BFT-CUP. Para atingir este objetivo, não é importante analisar o tempo necessário para a execução do BFT-CUP, mas sim as métricas diretamente relacionadas com a convergência do BFT-CUP. Neste sentido, as seguintes métricas serão analisadas nestas simulações:

1. *Conhecimento retornado pelos detectores de participação*: Esta métrica define a quantidade de conhecimento sobre a composição do sistema que os nós recebem diretamente dos detectores de participação. Deste modo, esta métrica refere-se ao conhecimento adquirido pelos participantes antes da execução do algoritmo DISCOVERY.

2. *Convergência para Poço*: Esta métrica refere-se a quantidade de participantes que acabam por determinar que pertencem à G_{sink} de G_{di}. No entanto, como veremos a seguir, em determinadas configurações é possível ocorrer execuções do BFT-CUP onde diferentes participantes acreditam que pertencem a diferentes componentes G_{sink}. Neste cenário é teoricamente possível ocorrer uma de duas coisas: *(1)* ou os participantes não terminam a execução do BFT-CUP (porque cada G_{sink} não possui nós suficientes para executar o consenso clássico); *(2)* ou ocorre desacordo nos valores das decisões (participates em diferentes G_{sink} podem decidir por valores diferentes – Teorema 2).

3. *Terminação*: Esta métrica define a quantidade de participantes que terminam a execução dos protocolos do BFT-CUP. Rigorosamente falando, os nós em G_{sink} não podem terminar o BFT-CUP quando decidem por algum valor, pois os mesmos devem responder às requisições de outros nós solicitando este valor da decisão. Assim, tais nós devem aguardar por estas requisições enquanto o valor de decisão é importante para a aplicação que está utilizando o BFT-CUP como base para algum

algoritmo distribuído. Nestas simulações, os nós em G_{sink} aguardaram por requisições de valor de decisão enquanto tinha algum evento para ser processado pelo simulador, i.e., enquanto existia a possibilidade de algum processo ter solicitado este valor de decisão.

4. *Acordo*: Esta métrica especifica a quantidade de participantes que terminam a execução dos protocolos do BFT-CUP decidindo pelo mesmo valor.

4.8.1.4 Resultados e Análises

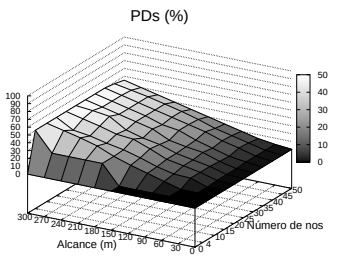
Esta seção apresenta os resultados obtidos através das simulações, bem como as análises sobre estes resultados. Primeiramente, a Figura 6 apresenta os resultados para simulações onde fixou-se o número de faltas suportadas pelo sistema (f) em 1 falta e variou-se a densidade e o alcance de transmissão dos nós. Como veremos a seguir, tanto o aumento da densidade de nós no sistema, quanto o aumento do alcance de transmissão dos nós, faz com que os participantes obtenham um melhor conhecimento sobre a composição do sistema, aumentando a conectividade em G_{di} e com isso também aumentando a probabilidade do BFT-CUP admitir solução.

Na Figura 6(a) é possível observar o grau de conhecimento sobre a composição do sistema que é obtido através dos detectores de participação (conhecimento obtido antes da execução do algoritmo DISCOVERY). Considerando a porcentagem de conhecimento inicial fornecido aos participantes (e não os números absolutos), é possível verificar que a porcentagem de conhecimento está diretamente relacionada com o alcance de transmissão dos nós, sendo que a densidade de nós presentes no sistema exerce pouca influência sobre este percentual. O conhecimento reportado pelos detectores de participação variou de aproximadamente 0% (para alcance de 30m) a aproximadamente 50% (para alcance de 300m). Considerando o cenário para alcance de 300m, isto significa que, em média, quando o sistema foi composto por 4 nós, o conhecimento sobre a presença de mais outro participante foi retornado pelos detectores de participação. Já para o cenário de 50 nós, este valor salta para 24 outros participantes.

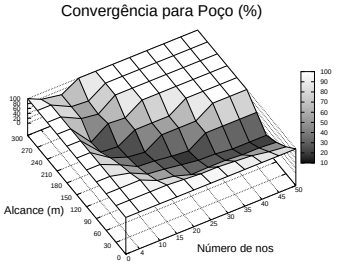
É importante notar que a união destes conhecimentos, obtidos em cada participante do sistema através de seus detectores de participação, gera o grafo de conectividade por conhecimento G_{di}, o qual indicará se a conectividade obtida é suficiente para resolver o BFT-CUP.

Após esta etapa de descoberta inicial, os nós executam o algoritmo DISCOVERY para expandir este conhecimento e na sequência o algoritmo SINK para determinar se pertencem à G_{sink} de G_{di}. Neste sentido, a Figura

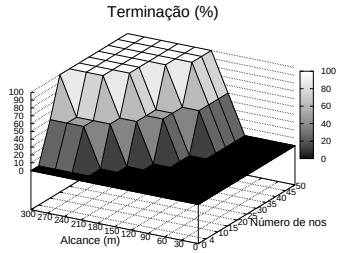
6(b) apresenta a porcentagem de participantes que acreditaram pertencer à G_{sink} nos mais variados cenários. Este gráfico apresenta três zonas distintas:



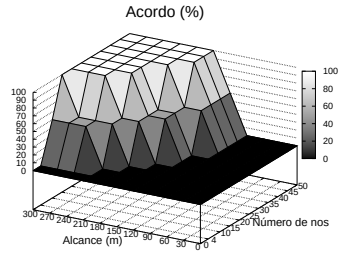
(a) Conhecimento adquirido pelos PDs.



(b) Convergência para Poço.



(c) Terminação.



(d) Acordo.

Figura 6 – Resultados das simulações para $f = 1$.

1. Alcance de transmissão e/ou densidade de nós baixos: Nestes cenários todos os nós acreditam pertencer à G_{sink}. No entanto, como os conhecimentos obtidos sobre a composição do sistema (depois do algoritmo DISCOVERY) são muito reduzidos, os nós acreditam pertencer a diferentes componentes G_{sink}. Em alguns cenários os nós descobriram apenas outro ou até mesmo nenhum outro participante.

2. Alcance de transmissão e/ou densidade de nós moderada: Estes cenários formam o fundo da "vala" na Figura 6(b), onde o conhecimento obtido pelos nós é tão divergente entre os mesmos (embaralhado) que nenhum nó (ou poucos) acreditam pertencer à G_{sink}.

3. Alcance de transmissão e/ou densidade de nós altos: Nestes cenários todos os nós decidiram que pertencem ao mesmo G_{sink}, pois adquiriram

um grande grau de conhecimento sobre a composição do sistema. Em praticamente todos estes cenários, os nós acabaram conhecendo todos os outros participantes do sistema.

As figuras 6(c) e 6(d) apresentam os gráficos para terminação e acordo, respectivamente. Como podemos perceber, estas métricas estão diretamente relacionadas com a convergência para um único G_{sink}, sendo que não é possível resolver o BFT-CUP nos cenários descritos anteriormente pelos itens 1 e 2. Nestes casos, o consenso tradicional não é executado, fazendo com que os protocolos do BFT-CUP não terminem, pelos seguintes motivos:

1. Cenários do item 1: Apesar de todos os nós acreditarem pertencer a G_{sink}, o consenso tradicional não é executado porque são diferentes componentes G_{sink} visto que os seus conhecimentos são muito reduzidos. De fato, nenhum nó consegue descobrir pelo menos outros 3 nós para executar o consenso clássico que exige $3f+1$ participantes (CASTRO; LISKOV, 2002) (como $f=1$, é necessário 4 participantes).

2. Cenários do item 2: O consenso clássico não é executado porque um número insuficiente de nós acreditam pertencer à G_{sink} (menos de $3f+1$ (CASTRO; LISKOV, 2002)).

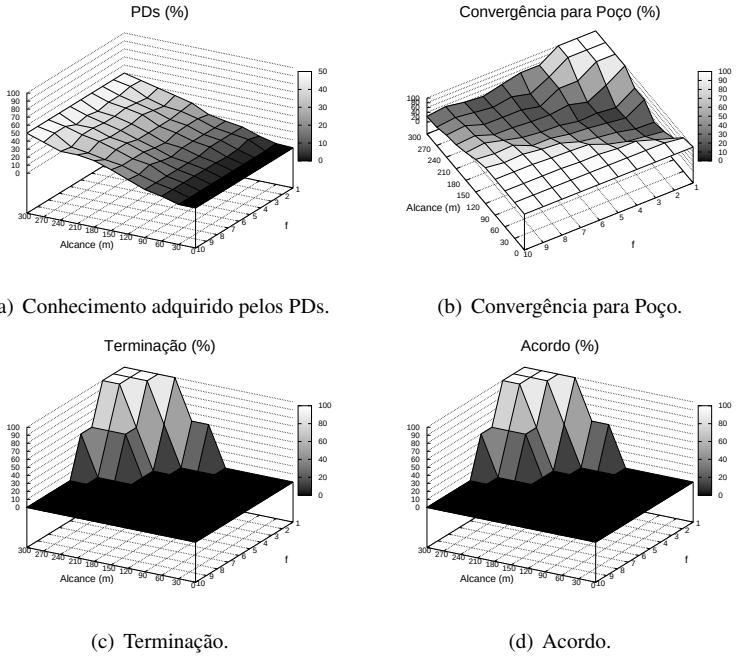
Por estes motivos, os gráficos de acordo e terminação são idênticos, representando que quando os algoritmos do BFT-CUP terminaram, a propriedade de acordo do consenso foi atendida. Este é um resultado importante, pois nestas simulações sempre que os nós terminaram a execução dos protocolos acabaram decidindo pelo mesmo valor.

A fim de analisar o comportamento do BFT-CUP em relação ao número de faltas suportadas pelo sistema, outros cenários foram simulados para o sistema composto por 30 nós. Os resultados são apresentados na Figura 7, onde é possível perceber que a probabilidade do BFT-CUP admitir solução em uma rede MANET é diretamente proporcional ao alcance de transmissão dos nós e inversamente proporcianal ao número de faltas toleradas pelo sistema.

A Figura 7(a) mostra que o número de faltas toleradas pelo sistema não têm relação com a quantidade de conhecimento obtida através dos detectores de participação. De fato, os detectores de participação trabalham de forma independente do BFT-CUP e este parâmetro não deve interferir em seu funcionamento.

Nestas simulações foi possível resolver o BFT-CUP em cenários para f de até 5 faltas e alcance de transmissão a partir de $150m$. O BFT-CUP nunca terminou nos cenários com f maior do que 5 faltas, pois nestes casos não foi possível formar um único G_{sink}. A Figura 7(b) apresenta a porcentagem de

nós que acreditaram pertencer à G_{sink}, cuja interpretação é semelhante ao experimento anterior (Figura 6(b)): *(1)* alcance de transmissão baixo – os nós acreditaram que pertenciam à diferentes componentes G_{sink}; *(2)* alcance de transmissão moderado e/ou f alto – poucos nós acreditaram pertencer à G_{sink};*(3)* alcance de transmissão alto e/ou f baixo – os nós decidiram que pertenciam à mesma componente G_{sink}.



(a) Conhecimento adquirido pelos PDs.

(b) Convergência para Poço.

(c) Terminação.

(d) Acordo.

Figura 7 – Resultados das simulações para 30 nós.

Novamente, sempre que os protocolos do BFT-CUP terminaram (Figura 7(c)) os nós acabaram decidindo pelo mesmo valor (Figura 7(d)), indicando que não ocorreu desacordo nos cenários simulados.

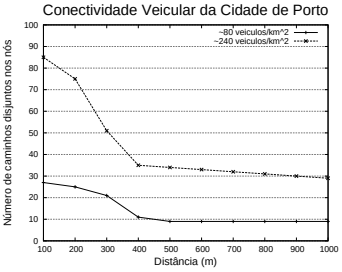
4.8.2 BFT-CUP em Redes VANETs: Exemplo Prático da Cidade de Porto - Portugal

Esta seção apresenta uma análise acerca da realização do BFT-CUP entre veículos (ou uma parte deles) que trafegam nas vias de uma cidade,

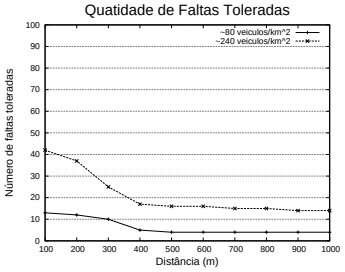
ou seja, em uma rede VANET. O cenário escolhido foi a cidade de Porto - Portugal, cujo tráfego veicular foi modelado por Ferreira *et al.* (FERREIRA et al., 2009). A mobilidade veicular e a comunicação entre os veículos são dois fatores fundamentais desta modelagem, que baseou-se no uso de fotografias aéreas (*stereoscopic aerial photography*) e mostrou-se mais realista que outros simuladores para VANETs (FERREIRA et al., 2009). De fato, a modelagem do tráfego urbano é uma tarefa árdua, sendo que cada cidade possui suas peculiaridades. Por exemplo, os motoristas preferem trafegar por determinadas vias em determinados horários, de acordo com o movimento dos pedestres e/ou de outros veículos nestes locais.

Nesta análise consideramos o grau de conectividade veicular apresentado por esta cidade, onde as rotas escolhidas pelos veículos constituem o fator que mais interfere nesta métrica, pois a concentração de veículos em um número reduzido de vias certamente afetará a conectividade de veículos que trafegam em outras vias.

A Figura 8(a) apresenta o grau de conectividade entre dois veículos em relação à distância entre os mesmos (dados extraídos de (FERREIRA et al., 2009)). Para este cálculo, fixou-se um veículo i e calculou-se a média de caminhos disjuntos nos nós que ligam os outros veículos ao i, de acordo com a distância que os mesmos se encontram. O gráfico apresenta o grau de conectividade para cenários com aproximadamente 80 e 240 veículos/km^2. Além disso, o alcance de transmissão foi especificado em 250m para comunicações diretas e 140m para comunicações onde alguma construção impediu a comunicação direta.



(a) Conectividade Veicular da Cidade de Porto (extraído de (FERREIRA et al., 2009))



(b) Quantidade de Faltas Toleradas

Figura 8 – O número de faltas toleradas está diretamente relacionado com o grau de conectividade apresentado pelo sistema.

A partir destes dados, podemos derivar o número máximo de faltas toleradas na execução do BFT-CUP entre estes veículos. Nesta analise consideraremos o pior caso para cada veículo, onde é previsto que o mesmo possuí até f vizinhos faltosos. Então, como é necessário $k \geq f+1$ caminhos corretos, onde k é o grau de conectividade, temos que $k \geq 2f+1$ e, assim, $f \leq \frac{k-1}{2}$. A Figura 8(b) mostra que é possível tolerar até aproximadamente 40 faltas em um cenário de 240 veículos/km^2, que se encontram em uma distância de até $100m$. Além disso, este número diminui bruscamente até os $400m$ de distância quando então tende a estabilizar.

Existem diversas aplicações para o consenso em redes VANETs. Por exemplo, existe uma proposta de substituição dos semáforos (infraestrutura localizada na via) pela execução de um consenso entre veículos (FERREIRA et al., 2010), que determinará qual destes veículos deve atravessar por primeiro em um cruzamento, implementando uma espécie de semáforo virtual. Esta abordagem visa otimizar o fluxo de veículos em cruzamentos, pois neste caso um veículo apenas irá parar no cruzamento quando realmente for necessário devido à presença de outros veículos competindo pelo cruzamento.

4.9 CONSIDERAÇÕES FINAIS

A maioria dos trabalhos sobre consenso encontrados na literatura considera um conjunto estático e conhecido de participantes envolvidos em uma computação (ex.: (CHANDRA; TOUEG, 1996; CORREIA et al., 2006; FRIEDMAN et al., 2005a; LAMPORT et al., 1982; MARTIN; ALVISI, 2006; TOUEG, 1984)). Recentemente, alguns trabalhos consideram que os participantes obtêm uma visão parcial sobre a composição do sistema (CAVIN et al., 2004, 2005; GREVE; TIXEUIL, 2007; ALCHIERI et al., 2008a). Neste sentido, estes trabalhos estudam as condições necessárias e suficientes para resolver o consenso quando o conjunto de participantes é desconhecido.

Nosso trabalho estende estes resultados apresentando algoritmos para resolver o FT-CUP em um sistema sujeito à faltas bizantinas (BFT-CUP). Neste sentido, mostramos que para resolver o BFT-CUP em um ambiente com pouca sincronia é necessário um maior grau de conectividade entre os participantes do sistema, quando comparado à solução para o FT-CUP (GREVE; TIXEUIL, 2007). O principal resultado deste trabalho é a constatação de que é possível resolver o BFT-CUP com a mesma classe de detector de participação (k-OSR) e o mesmo grau de sincronia ($\diamondsuit\mathscr{S}$) necessário para resolver o FT-CUP (GREVE; TIXEUIL, 2007). Além disso, especificamos o mais fraco PD capaz de resolver o BFT-CUP (e também o FT-CUP), onde adicionamos um padrão seguro de falhas bizantinas que impõe uma restrição na combinação

de falhas e também reduz o grau de conhecimento que deve ser obtido pelos participantes do sistema. Por fim, também especificamos uma primitiva de difusão, chamada *reachable reliable broadcast*, que pode ser usada em outros protocolos para redes desconhecidas.

A Tabela 6 apresenta uma comparação entre os resultados já estabelecidos sobre a resolução do consenso entre participantes desconhecidos. Nesta tabela, podemos observar quais são as condições necessárias e suficientes para a solução do consenso entre participantes desconhecidos, considerando diferentes domínios de falhas e sincronia.

Abordagem	Modelo de falhas	PD	k	participantes no poço
CUP Cavin *et al.* 2004	sem faltas	OSR	–	1
FT-CUP Cavin *et al.* 2005	parada (*crash*)	OSR	–	1
FT-CUP Greve *et al.* 2007	parada (*crash*)	k-OSR	$f+1$	$2f+1$
BFT-CUP Alchieri *et al.* 2008	bizantina	k-OSR	$2f+1$	$3f+1$
BFT-CUP (este trabalho)	bizantina	k-OSR	padrão seguro de faltas bizantinas	$2f+1$ participantes corretos

continuação...

Abordagem	conectividade entre os participantes	modelo de sincronia
CUP Cavin *et al.* 2004	OSR	assíncrono
FT-CUP Cavin *et al.* 2005	OSR + padrão seguro de falhas	assíncrono + $\mathscr{P}$
FT-CUP Greve *et al.* 2007	k caminhos disjuntos nos nós	assíncrono + $\diamondsuit\mathscr{S}$
BFT-CUP Alchieri *et al.* 2008	k caminhos disjuntos nos nós	mesmo nível requerido pelo protocolo de consenso subjacente
BFT-CUP (este trabalho)	k-OSR + padrão seguro de falhas bizantinas	mesmo nível requerido pelo protocolo de consenso subjacente

Tabela 6 – Soluções do consenso entre participantes desconhecidos.

Por fim, a partir das simulações discutidas neste capítulo, é possível identificar quais são os parâmetros e configurações do sistema que devem ser considerados quando estes protocolos forem utilizados no desenvolvimento de aplicações: *(1)* a escolha do parâmetro f é determinante para a convergência de BFT-CUP; *(2)* uma escolha adequada entre o alcance de transmissão e a densidade de nós no sistema também determinará a possibilidade de convergência de BFT-CUP, quando consideramos uma rede MANET.

Nestas simulações, nas configurações onde o BFT-CUP admitiu solução, geralmente o sistema apresentou um grafo de conectividade por conhecimento G_{di} k-fortemente conexo (apenas uma componente k-fortemente co-

nexa). Outro ponto a destacar é que a propriedade de acordo não foi violada em nenhum cenário analisado, i.e., sempre que os protocolos do BFT-CUP terminaram, os participantes acabaram decidindo pelo mesmo valor.

5 SISTEMAS DE QUÓRUNS BIZANTINOS DINÂMICOS

Este capítulo apresenta uma série de protocolos para reconfiguração de sistemas de quóruns bizantinos, discutindo vários problemas relacionados com a utilização dos mesmos em diversos modelos de sistemas, onde soluções para cada modelo são abordadas. Além disso, são apresentados protocolos para tolerar também clientes maliciosos nestes ambientes dinâmicos. Por fim, vários aspectos das soluções propostas são discutidos, onde algumas soluções alternativas são apontadas.

5.1 INTRODUÇÃO

Sistemas de quóruns (GIFFORD, 1979) são ferramentas usadas para garantir consistência e disponibilidade de dados, que são armazenados de forma replicada em um conjunto de servidores. Além de seu uso como blocos básico de construção (*building blocks*) para protocolos de sincronização (ex.: consenso), o grande atrativo destes sistemas está relacionado com seu poder de escalabilidade e balanceamento de carga, uma vez que as operações não precisam ser executadas por todos os servidores que compõem o sistema, mas apenas por um quórum dos mesmos. A consistência de um sistema de quóruns é assegurada pela propriedade de intersecção dos subconjuntos de servidores (quóruns). Estes sistemas concretizam um registrador que permite operações de leitura e escrita com diferentes semânticas de implementação (Seção 3.4.4.1).

Sistemas de quóruns foram inicialmente estudados em ambientes estáticos, onde não é permitida a entrada e saída de servidores durante a execução do sistema (GIFFORD, 1979; BAZZI; DING, 2004; MALKHI; REITER, 1998a, 1998b). Esta abordagem não é adequada para sistemas que permanecerão em execução por um longo tempo (*long lived systems*), uma vez que dispondo de um quantidade suficiente de tempo, um adversário pode comprometer um número maior de servidores do que o tolerado por determinado sistema e então quebrar as propriedades do mesmo. Outra limitação destes protocolos é que os mesmos não permitem que um administrador, em tempo de execução, adicione novas máquinas no sistema (para suportar um aumento na carga de processamento) ou troque máquinas antigas. Além disso, estes protocolos não são adequados para muitos sistemas desenvolvidos para ambientes onde, pela sua própria natureza, o conjunto de processos que compõem o sistema sofre modificações durante a execução (ex.: MANETs e P2P).

Para superar estas limitações, alguns trabalhos adaptaram protocolos

de sistemas de quóruns para ambientes dinâmicos, onde servidores podem entrar e sair do sistema durante a execução do mesmo (AGUILERA et al., 2009, 2011; LYNCH; SHVARTSMAN, 2002; MARTIN; ALVISI, 2004; RODRIGUES; LISKOV, 2004). Esta dinamicidade do sistema ocasiona o aparecimento de novos desafios no projeto destes protocolos. Além da necessidade de gerenciamento de visões (*membership*), também é necessário ficar atento ao número de processos que estão deixando o sistema, uma vez que caso este número seja grande, o sistema pode perder vivacidade (AGUILERA et al., 2009, 2011). Outro desafio é a manutenção das propriedades de segurança do sistema enquanto o mesmo estiver sendo reconfigurado, i.e., enquanto processos estão entrando e saindo do sistema. Lidar com estes problemas é ainda mais complexo quando consideramos a possibilidade de componentes maliciosos estarem presentes em determinada computação.

Protocolos que implementam sistemas de quóruns em ambientes estáticos (ex.: (BAZZI; DING, 2004; MALKHI; REITER, 1998a, 1998b)) não precisam de primitivas fortes de sincronização, como consenso. No entanto, a maioria dos trabalhos sobre armazenamento dinâmico utiliza consenso para reconfigurações (LYNCH; SHVARTSMAN, 2002; MARTIN; ALVISI, 2004; RODRIGUES; LISKOV, 2004), fazendo os processos acordarem sobre uma sequência de visões, como em um protocolo de sincronismo de visões (*view-synchronous membership protocol* (CHOCKLER et al., 2001)). A única exceção é o recente algoritmo *DynaStore* (AGUILERA et al., 2009, 2011), que utiliza *weak snapshot objects* para implementar um registrador dinâmico tolerante a faltas por parada, onde as reconfigurações são executadas sem o auxilio de consenso. Neste protocolo as reconfigurações ocorrem em qualquer momento, gerando um grafo de visões a partir do qual é possível identificar uma sequência de visões onde os clientes executam operações.

Neste capítulo, propomos protocolos para sistemas de quóruns bizantinos dinâmicos do tipo f-disseminação (*dynamic f-dissemination Byzantine quorum system*) (MALKHI; REITER, 1998b), os quais toleram faltas maliciosas nos servidores e nos clientes leitores, além de um número ilimitado de faltas por parada de clientes escritores. No final deste capítulo discutimos um protocolo para tolerar também escritores maliciosos.

Inicialmente, propomos uma abstração, chamada *geradores de visões*, que é responsável pela geração de novas visões para reconfiguração do sistema. Estes geradores são classificados de acordo com certas propriedades definidas para os mesmos, onde a classe mais forte necessita de consenso para gerar uma nova visão (Seção 5.6.1.1) mas garante tanto as propriedades de segurança (*safety* – cada gerador gera uma única visão) quanto as de terminação (*liveness* – um gerador sempre gera alguma visão) dos geradores. Em contrapartida, por utilizar consenso, este gerador não pode ser usado em

sistemas completamente assíncronos.

Deste modo, também propomos implementações de outras classes de geradores de visões que não utilizam primitivas de consenso e portanto podem ser empregados em modelos assíncronos. Em contrapartida, tais protocolos não conseguem garantir tanto as propriedades de segurança quanto as de terminação na geração de uma nova visão. Primeiramente, apresentamos um gerador que garante as propriedades de segurança (Seção 5.6.1.2) e posteriormente um que garante as propriedades de terminação (Seção 5.6.1.3). Neste sentido, propomos protocolos para reconfiguração do sistema através de cada um dos geradores de visões, sendo que *estes protocolos sempre atendem as propriedades de segurança e de terminação do sistema* (Seção 5.3).

Os protocolos propostos neste trabalho fazem parte de um sistema chamado QUINCUNX[1], que implementa um registrador atômico livre de espera (*wait-free*) (HERLIHY, 1991) e, ao mesmo tempo, permite reconfigurações no conjunto de servidores. Os protocolos de leitura e escrita do QUINCUNX são uma variante do PHALANX (MALKHI; REITER, 1998b), que é uma das primeiras implementações de registrador atômico tolerante a faltas bizantinas.

O QUINCUNX é composto por dois conjuntos de protocolos: *(1)* protocolos de leitura e escrita; e *(2)* protocolos para atualização de visões. Protocolos de leitura e escrita estão relacionados com a comunicação entre clientes e servidores e são muito similares as suas versões estáticas (MALKHI; REITER, 1998b). Já os protocolos para atualização de visões implementam os procedimentos de reconfiguração do sistema em tempo de execução, um requisito indispensável para sistemas dinâmicos. Por tolerar faltas bizantinas, estes protocolos são interessantes para sistemas dinâmicos que, devido às suas características de sistemas abertos, são mais susceptíveis à presença de componentes com comportamento faltoso, devido tanto a atividades maliciosas quanto a faltas não omissas causadas por *hardware/software*.

5.2 MODELO DE SISTEMA

Consideramos um sistema distribuído completamente conectado composto pelo conjunto universo de processos U, que é dividido em três subconjuntos distintos: um conjunto infinito de servidores $\Pi = \{s_1, s_2, ...\}$; um conjunto infinito de clientes leitores $R = \{r_1, r_2, ...\}$; e um conjunto infinito de clientes escritores $W = \{w_1, w_2, ...\}$. Cada processo do sistema (cliente ou servidor) possui um identificador único. Leitores e escritores acessam o sistema implementado por algum subconjunto de servidores através da execução de operações de leitura e escrita, respectivamente. A chegada dos processos

[1] Uma formação militar romana que permite reconfigurações nas linhas de batalha.

no sistema segue o modelo de chegadas infinita com concorrência desconhecida mas finita (modelo M_2^{finito} da Tabela 1).

Os servidores e os leitores estão sujeitos a faltas Bizantinas (LAMPORT et al., 1982), i.e., podem exibir qualquer comportamento, podendo parar, omitir envio ou entrega de mensagens, ou desviar de suas especificações arbitrariamente e trabalhar em conjunto com o objetivo de corromper o sistema. Por outro lado, os escritores apenas podem exibir comportamento de falha por parada. Um processo que apresenta comportamento de falha é dito falho (ou faltoso), de outra forma é dito correto.

A solução que emprega consenso para reconfigurações (Seção 5.6.1.1) necessita de um sistema parcialmente síncrono (DWORK et al., 1988). A ideia por trás destes modelos é de que o sistema trabalha de forma assíncrona (não respeitando nenhum limite de tempo) a maior parte do tempo. Porém, durante períodos de estabilidade, o tempo para transmissão de mensagens é limitado. Já para as soluções que não utilizam consenso (seções 5.6.1.2 e 5.6.1.3), consideramos um sistema distribuído assíncrono, onde não existem limites para o tempo de transmissão de mensagens ou processamentos locais nos processos. Além disso, cada servidor do sistema tem acesso a um relógio local usado para iniciar reconfigurações. Estes relógios não são sincronizados e não existe qualquer limite (*bounds*) para seus desvios (*drifts*), sendo portanto apenas contadores.

Por simplicidade, consideramos que as comunicações entre os processos são realizadas através de canais ponto-a-ponto confiáveis e autenticados. Estes canais são implementáveis em um sistema assíncrono através de SSL/TLS (DIERKS; ALLEN, 1999). No entanto, os processos poderiam conectar-se por uma rede não confiável, que poderia: não enviar, atrasar o envio, duplicar ou corromper mensagens. Assim, para terminação seria necessário que (1) mensagens corrompidas fossem descartadas; (2) um processo faltoso não conseguisse se passar por outros processos; e (3) se um processo enviasse uma mensagem infinitas vezes para outro processo correto, então tal mensagem acabaria por ser entregue no receptor. Os requisitos (1) e (2) são garantidos pelo uso de códigos de autenticação de mensagens enquanto que o requisito (3) poderia ser assegurado por *fair links* (LYNCH, 1996).

Finalmente, consideramos que cada servidor possui um par distinto de chaves (chave pública e privada) para usar um sistema de criptografia assimétrica. Todos os escritores também compartilham um par de chaves: as *chaves pública e privada de escrita*. Cada chave privada é conhecida apenas pelo seu próprio dono (ou donos no caso dos escritores), por outro lado todos os processos conhecem todas as chaves públicas. Denotamos uma mensagem m assinada pelo processo i como m_i e consideramos que apenas mensagens corretamente assinadas são processadas pelos processos corretos.

5.3 DINAMICIDADE E PROPRIEDADES DO SISTEMA

Nesta seção definimos as terminologias básicas, bem como algumas considerações e propriedades relevantes para os protocolos do QUINCUNX. Definimos um $update = \{+,-\} \times \Pi$, onde a tupla $\langle +,i\rangle_i$ ou $\langle -,i\rangle_i$ (assinada por i) indica que o servidor i requisitou um *join* ou *leave* do sistema, respectivamente. Dizemos que um *update* foi proposto por i na reconfiguração r caso i execute a operação de *join* ou *leave* antes de r iniciar e nenhuma reconfiguração anterior processou este *update*. Para um conjunto de *updates* u, o conjunto de remoções de u, denotado $u.remove$, é o conjunto $\{i \in \Pi : \langle -,i\rangle_i \in u\}$. Similarmente, o conjunto de adições de u, denotado $u.join$, é o conjunto $\{i \in \Pi : \langle +,i\rangle_i \in u\}$. O *membership* de u, denotado $u.members$, é o conjunto $u.join \setminus u.remove$. O *membership* de uma visão v é definido pela união de todos os *updates* computados em v. Um servidor i pertence a v se e somente se $i \in v.members$ (também usamos a nomenclatura $i \in v$).

Definição 10 (Visão atual no tempo *t*) *Em qualquer tempo t da execução, definimos $V(t)$ como sendo a visão mais atual que algum servidor correto $i \in V(t)$ instalou desde o início de sua execução até t.*

Seguindo a Definição 10, para qualquer tempo t, os protocolos de reconfiguração do QUINCUNX garantem que pelo menos um quórum de servidores da(s) visão(ões) anterior(es) conhece(m) $V(t)$, de forma que $V(t)$ é a única visão onde clientes podem executar operações de leitura e escrita no tempo t. $V(t)$ permanece *ativa* desde o momento em que foi instalada por i até que uma reconfiguração faça com que todos os servidores corretos de uma outra visão mais atualizada instalem esta última visão.

Consideramos uma visão inicial $V(0)$ não vazia, que é inicialmente conhecida por todos os processos do sistema no tempo t_0. Também definimos $U(t)$ como sendo o conjunto de *updates pendentes* no tempo t, onde para cada $i \in \Pi$ tal que $\langle +,i\rangle_i \in U(t).join$ temos que $i \notin V(t).join$ e para cada $i \in \Pi$ tal que $\langle -,i\rangle_i \in U(t).remove$ temos que $i \notin V(t).remove$ (i.e., o *update* não foi processado até $V(t)$). Dadas estas definições, as seguintes hipóteses devem ser atendidas para o correto funcionamento do QUINCUNX.

Suposição 3 (Gentle leaves) *Um servidor correto i que requisita um leave no tempo t ($i \in V(t).members \wedge i \in U(t).remove$) permanece no sistema até terminar a instalação de uma visão atualizada na qual i não pertença.*

Esta suposição garante a presença de pelo menos um quórum de servidores corretos em uma dada visão ativa do sistema, garantindo o término das operações e a vivacidade do mesmo.

Suposição 4 (Finite reconfigurations) *O número de updates (join ou leave) requisitados em uma execução é finito.*

Esta suposição permite que as operações de leitura e escrita de clientes, bem como as operações de *join* e *leave* dos servidores, possam terminar (propriedade de *wait-free*). Para não acessar dados obsoletos, estas operações devem ser executadas na visão mais atual do sistema. Assim, caso um processo muito lento não consiga completar sua requisição antes que a visão do sistema seja atualizada, tal processo deve reiniciar esta requisição utilizando a visão já atualizada. De fato, somente o número de *updates* requisitados concorrentemente com cada operação precisa ser finito. Outros protocolos, como o *DynaStore* (AGUILERA et al., 2009, 2011), também consideram que requisições de *updates* são finitas. Na verdade, todas as propostas para sistemas de quóruns dinâmicos (LYNCH; SHVARTSMAN, 2002; MARTIN; ALVISI, 2004; RODRIGUES; LISKOV, 2004) necessitam reiniciar operações devido à reconfigurações concorrentes e, então, devem fazer uso desta suposição para fornecer operações livre de espera (*wait-free*).

Suposição 5 (Fault threshold) *No máximo* $\lfloor \frac{|V(t).members|-1}{3} \rfloor$ *servidores em* $V(t)$ *podem falhar.*

Esta suposição restringe o número máximo de falhas de servidores que podem ocorrer em cada visão do sistema, representando a resiliência ótima para sistemas de quóruns bizantinos de disseminação (MALKHI; REITER, 1998b): este limite é uma generalização da usual equação $n \geq 3f+1$.

Antes de descrever as propriedades do QUINCUNX, algumas terminologias adicionais precisam ser introduzidas. Em qualquer tempo t, somente servidores em $V(t)$ podem enviar respostas para as operações de leitura e escrita executadas por clientes. Quando um servidor i requisita um *join* no sistema, estas operações permanecem desabilitadas em i até a ocorrência de um evento *enable operations*. Após isso, i permanecerá apto para responder estas operações até que i solicite o *leave* do sistema, o que acontece com a ocorrência do evento *disable operations*.

Definição 11 (Propriedades do QUINCUNX) *Dado que, em qualquer tempo de execução t, as suposições 3, 4 e 5 sejam satisfeitas, as seguintes propriedades devem ser garantidas:*

- ***Segurança 1 (Armazenamento)**: Os protocolos de leitura e escrita satisfazem as propriedades de segurança de um registrador de leitura e escrita atômico (LAMPORT, 1986).*

- ***Segurança 2 (Reconfiguração – Join)****: Se um servidor correto $i \notin V(t)$ executa a operação de join no tempo t, então existe um tempo $t' > t$ tal que $i \in V(t')$.*

- ***Segurança 3 (Reconfiguração – Leave)****: Se um servidor correto $i \in V(t)$ executa a operação de leave no tempo t, então existe um tempo $t' > t$ tal que $i \notin V(t')$.*

- ***Terminação 1 (Armazenamento)****: Todas operações de leitura e escrita executadas por clientes corretos terminam.*

- ***Terminação 2 (Reconfiguração – Join)****: O evento enable operations terminará por ocorrer em todo servidor correto que executa a operação de join.*

- ***Terminação 3 (Reconfiguração – Leave)****: O evento disable operations terminará por ocorrer em todo servidor correto que executa a operação de leave.*

5.4 VISÕES: ESTRUTURAS E OPERAÇÕES

Antes de descrever os protocolos para armazenamento de dados e reconfiguração do QUINCUNX, apresentamos a estrutura de dados e as operações realizadas com as visões, uma vez que as mesmas são vastamente utilizadas por estes protocolos.

Cada visão v é uma tupla $\langle ov, entries, \mathtt{P} \rangle$, onde: ov é a visão que gerou v, i.e., na reconfiguração de ov a visão v é gerada pelos servidores em ov; $entries$ é um conjunto de $updates$ (Seção 5.3) e define o *membership* de v, como veremos a seguir; e P é um certificado contendo as provas que garantem tanto a integridade de v como também que v foi instalada no sistema.

O campo $entries$ é incremental: para cada visão v, temos que $v.entries = v.ov.entries \cup updates$, onde $updates$ é um conjunto de atualizações válidas usadas para reconfigurar o sistema (Seção 5.3). Como mencionado, $v.entries$ define o *membership* de v: definimos $v.members = \{i \in \Pi : \langle +, i \rangle_i \in v.entries \wedge \langle -, i \rangle_i \notin v.entries\}$ (também usamos a notação $i \in v.members$ ou $i \in v$). Note que um servidor pode entrar e sair do sistema apenas uma única vez, mas podemos relaxar esta condição na prática adicionando números de época em cada requisição de reconfiguração.

Bootstrapping. Na inicialização do sistema, cada servidor ativo $i \in V(0)$ recebe a visão inicial $v_0 = \langle \perp, u_0, \emptyset \rangle$, onde $u_0 = \{\langle +, j \rangle_j : j \in V(0)\}$. $V(0)$ é conhecida por todos os processos do sistema, não sendo necessárias provas para garantir sua integridade.

Limites de falhas e tamanho dos quóruns. Para cada visão v, denotamos $v.f$ como sendo o número de falhas toleradas em v. Seguindo a Suposição 5, $v.f = \lfloor \frac{|v.members|-1}{3} \rfloor$. Os protocolos do QUINCUNX utilizam quóruns de tamanho $v.q = \lceil \frac{|v.members|+v.f+1}{2} \rceil$ servidores (MALKHI; REITER, 1998b).
Comparando e validando visões. Comparamos duas visões v_1 e v_2 através de seus campos *entries*. Usamos a notação $v_1 \subset v_2$ e $v_1 = v_2$ como uma abreviação para $v_1.entries \subset v_2.entries$ e $v_1.entries = v_2.entries$, respectivamente. Caso $v_1 \subset v_2$, então v_2 é *mais atual* do que v_1. Cada solução para reconfiguração do QUINCUNX produz diferentes provas de validade para as visões. Deste modo, cada forma de validar visões é apresenta juntamente com a solução para reconfiguração do sistema relacionada.

5.5 PROTOCOLOS DE LEITURA E ESCRITA

O fato das reconfigurações do QUINCUNX serem desacopladas dos protocolos de leitura e escrita, usado pelos clientes para armazenamento de dados, faz com que estes protocolos sejam muito similares com os protocolos do PHALANX (MALKHI; REITER, 1998b), seu precursor para sistemas estáticos. A única diferença é que cada processo i (servidor ou cliente) possui uma variável cv (visão corrente) usada para armazenar a visão mais atual conhecida por i. Desta forma, o único processamento adicional nos protocolos de leitura e escrita do QUINCUNX (quando comparado com o PHALANX) está relacionado com o gerenciamento desta variável, atualizando-a de acordo com as reconfigurações do sistema. Cada processo anexa a sua visão corrente em todas as mensagens trocadas pelo protocolo e atualiza esta visão assim que descobre uma visão mais recente disponível no sistema.

Antes de acessar o sistema, um cliente c precisa obter a visão corrente do mesmo, o que pode ser realizado através do armazenamento das visões, por parte dos servidores, em algum lugar padrão. Para aumentar o grau de disponibilidade, visões podem ser registradas em vários lugares. Outra opção é fazer com que c realize um "*flooding*" (inundação) de uma mensagem no sistema requisitando a visão corrente. Além disso, caso c permaneça por um longo tempo sem enviar requisições para os servidores, c pode estar armazenando uma visão muito desatualizada, contendo apenas servidores que já deixaram o sistema e não estão aptos à enviar respostas. Deste modo, c precisa aguardar algum tempo pelas respostas de uma requisição e caso não complete esta requisição neste período, c deve atualizar sua visão corrente como acima descrito. Note que este é um problema intrínseco de qualquer sistema dinâmico (ex.:(LYNCH; SHVARTSMAN, 2002; AGUILERA et al., 2009, 2011; MARTIN; ALVISI, 2004; RODRIGUES; LISKOV, 2004)).

Os protocolos de leitura e escrita do QUINCUNX executam em fases, sendo que cada fase compreende o envio de uma mensagem para um quórum de servidores em *cv* e o recebimento das respostas deste quórum. As operações de leitura e escrita necessitam de 1 ou 2 e 2 fases, respectivamente, como no PHALANX (MALKHI; REITER, 1998b). Para evitar ataques de *replay*, deve-se anexar *nonces* nas mensagens trocadas entre clientes e servidores e os clientes não podem escolher *nonces* repetidos. Em geral, estes algoritmos funcionam da seguinte maneira:

Operações de Leitura. *1º Fase*: O leitor requisita um conjunto de tuplas $\langle val, ts \rangle_w$, assinadas com o chave privada de escrita, de um quórum de servidores em *cv* (*val* é o valor que o servidor armazena e *ts* é o *timestamp* associado a este valor) e seleciona a tupla $\langle val_h, ts_h \rangle_w$ com o maior *timestamp* ts_h. A operação termina e retorna val_h caso todos os pares retornados sejam iguais, o que acontece em execuções sem concorrência de escritas e sem faltas. *2º Fase*: Caso contrário, o leitor executa uma fase adicional de *write-back* no sistema e aguarda por confirmações até ter certeza que um quórum de servidores em *cv* recebeu $\langle val_h, ts_h \rangle_w$ e, então, retorna val_h.

Operações de Escrita. *1º Fase*: O escritor obtêm um conjunto de *timestamps* de um quórum de servidores em *cv* e seleciona o maior *timestamp* ts_h. O *timestamp* da escrita *ts* é definido através do incremento de ts_h e da concatenação do identificador do escritor nos *bits* de menor ordem (para resolver conflitos de escritas com o mesmo *timestamp*). *2º Fase*: O escritor envia a tupla $\langle val, ts \rangle_w$, assinada com a chave privada de escrita, para um quórum de servidores em *cv*, escrevendo *val* neste quórum, e aguarda pelas confirmações.

5.5.1 Lidando com Reconfigurações

Como já mencionado, todas as mensagens trocadas pelos protocolos contêm a visão mais atual instalada no emissor. Desta forma, quando um cliente *c* tenta executar alguma fase de algum dos protocolos, recebendo respostas de um quórum de servidores, *c* verifica se alguma visão válida recebida[2] é mais atual do que a visão que armazena. Se este for o caso, *c* atualiza sua variável *cv* e reinicia a fase que está tentando executar. Isto é necessário para garantir que *c* acessará a configuração mais atual do sistema, o que é imprescindível para a correta execução das operações de leitura e escrita. Caso contrário, *c* poderia acessar dados obsoletos.

[2]Um servidor malicioso não é capaz de gerar uma visão válida - Seção 5.6.

5.6 PROTOCOLOS DE ATUALIZAÇÃO DE VISÕES

Esta seção apresenta os protocolos utilizados na reconfiguração do sistema. Este procedimento pode ser dividido em duas partes: uma nova visão mais atual primeiramente deve ser gerada para então ser efetivamente instalada no sistema. Deste modo, apresentamos o conceito de *geradores de visões*, discutimos suas propriedades e definimos diversas classes de geradores, de acordo com suas características e sua forma de funcionamento. Por fim, apresentamos protocolos para reconfigurar o sistema através de cada um dos geradores de visões especificados.

5.6.1 Geradores de Visões

Geradores de visões são abstrações utilizadas pelos servidores do sistema na obtenção de novas visões. Para cada visão instalada no sistema v, existe um gerador de visões associado. Por simplicidade, consideramos que este gerador é implementado pelos próprios servidores presentes em v, mas o mesmo pode ser executado de qualquer forma, desde que suas propriedades sejam atendidas.

Esta abstração fornece duas operações básicas:

- $generate\_view(v, updates)$: função usada pelos servidores de v, para iniciar o gerador de visões associado a esta visão, o qual atuará na obtenção de uma nova visão atualizada pela computação de $updates$ (Seção 5.3).
- $new\_view(v, w)$: função usada pelo gerador de visões, para informar os servidores de v sobre a obtenção de uma visão mais atual w.

Os geradores de visões podem implementar estas operações de diversas formas. Estas diferenças geralmente são motivadas pelos vários ambientes nos quais os geradores são projetados para serem executados (ex.: síncrono ou assíncrono), mas podem aparecer por qualquer outro motivo (ex.: pela escolha de um algoritmo mais eficiente). Em vista disso, as seguintes propriedades são definidas para os geradores de visões:

- Exatidão forte: o gerador de visões gera uma única visão w em todos os processos.
- Exatidão fraca: o gerador de visões pode gerar diferentes visões em diferentes processos. No entanto, para qualquer duas visões w e w' geradas, temos que $w \subset w' \vee w' \subset w$.

- Terminação forte: o gerador de visões acabará por gerar uma visão.
- Terminação fraca: o gerador de visões pode não gerar uma visão.
- Não trivialidade: para qualquer visão w, gerada pelo gerador de visões associado a um visão v, temos que $v \subset w$.

As propriedades de exatidão estão relacionadas com a quantidade de visões geradas (segurança na geração de visões), enquanto que as propriedades de terminação abstraem a possibilidade de um gerador de visões não terminar. A combinação destas propriedades define quatro classes de geradores de visões, apresentadas na Tabela 7. Por outro lado, a propriedade de não trivialidade deve ser satisfeita por qualquer gerador de visões, garantindo que as visões geradas são sempre mais atuais.

Tabela 7 – Classes de geradores de visões.

Terminação	**Exatidão**	
	Forte	Fraca
Forte	*Perfeito* $\mathscr{P}$	*Vivo* $\mathscr{L}$
Fraca	*Seguro* $\mathscr{S}$	*Fraco* $\mathscr{W}$

Geradores das classes $\mathscr{P}$ e $\mathscr{S}$ são considerados seguros, pois os mesmos geram uma única visão (propriedade de exatidão forte). Já os geradores das classes $\mathscr{P}$ e $\mathscr{L}$ são considerados vivos, pois os mesmos sempre geram alguma visão (propriedade de terminação forte).

Por outro lado, os geradores da classe $\mathscr{W}$ são considerados fracos pois podem não terminar (gerar uma visão) ou ainda gerar várias visões. Os geradores fracos possuem apenas interesse teórico uma vez que até mesmo em sistemas assíncronos é possível garantir ou exatidão forte ou terminação forte. Deste modo, a seguir apresentamos protocolos que implementam cada uma das outras três classes de geradores de visões que consideramos realistas (geradores $\mathscr{P}$, $\mathscr{S}$ e $\mathscr{L}$) e também protocolos para reconfiguração do QUINCUNX a partir de cada um destes geradores.

5.6.1.1 Gerador de Visões Perfeito

Este gerador de visões possui as propriedades de exatidão e terminação fortes (Seção 5.6.1). Deste modo, o mesmo garante que *uma única* (exatidão forte) visão é *sempre* gerada (terminação forte). Para garantir estas propriedades, é necessário o auxílio de um protocolo de consenso (Seção 3.4.2), o qual

requer premissas de tempo para garantir a propriedade de terminação, como por exemplo sincronia parcial (DWORK et al., 1988).

A implementação deste gerador, apresentada no Algoritmo 5, é bastante simples. A ideia é que quando o mesmo for iniciado em um servidor $i \in v$, para a obtenção de uma nova visão para atualização da visão v, um protocolo de consenso seja executado onde i propõe uma nova visão w' que computa os *updates* recebidos na inicialização (linha 2). Quando i terminar o protocolo de consenso, decidindo por uma nova visão mais atual w, o servidor que iniciou o gerador é notificado através da função *new_view* (linha 4).

Algoritmo 5 Gerador de visões perfeito (servidor $i \in v$)

```
procedure generate_view(v, updates)
1: w' ← ⟨v, v.entries ∪ updates, ∅⟩
2: Consensus.propose(w');          {underlying Byzantine consensus with all i ∈ v}
3: upon Consensus.decide(w)
4: new_view(v, w)
```

5.6.1.1.1 Corretude

As propriedades do protocolo de consenso (Seção 3.4.2) garantem que todos os servidores que implementam este gerador acabam decidindo pela mesma visão mais atual w, garantindo as propriedades deste gerador.

Teorema 3 *O gerador de visões perfeito implementado pelo Algoritmo 5 satisfaz as propriedades de exatidão forte, terminação forte e não trivialidade.*

Prova. Esta prova é derivada diretamente a partir das propriedades garantidas pelos protocolos de consenso (Seção 3.4.2): *exatidão forte* é garantida pela propriedade de acordo do consenso; *terminação forte* é garantida pela propriedade de terminação do consenso; e *não trivialidade* é garantida pelas propostas dos servidores, que são para visões mais atuais (linha 1), e pela propriedade de validade do consenso. $\square_{Teorema\ 3}$

5.6.1.2 Gerador de Visões Seguro

Uma maneira usual de gerenciar as visões de um sistema dinâmico é empregar um protocolo de *group membership* (CHOCKLER et al., 2001), onde dado uma visão v todos os processos geram uma única visão seguinte w (como os geradores perfeitos - Seção 5.6.1.1). No entanto, estes protocolos não são implementáveis em sistemas assíncronos e novas abordagens devem ser consideradas para estes ambientes.

O gerador de visões seguro possui as propriedades de exatidão forte e terminação fraca (Seção 5.6.1), garantindo que *uma única* (exatidão forte) visão *poderá* ser gerada (terminação fraca). Como veremos a seguir, é possível garantir estas propriedades em um sistema assíncrono (Seção 5.2).

Por satisfazer a propriedade de exatidão forte, garantindo segurança na geração das visões (a maioria dos protocolos tolerantes a faltas e intrusões opta pela preservação das propriedades de segurança ao invés das propriedades de terminação, são os chamados algoritmos indulgentes (RAYNAL, 2005; GUERRAOUI, 2000)), um gerador de visões seguro pode não terminar seu protocolo e não gerar uma visão mais atual (terminação fraca). Em nossa proposta de implementação para este gerador, isto somente ocorre caso a entrega das mensagens siga um padrão muito improvável de acontecer na prática (ver adiante). De qualquer modo, no final desta seção discutimos as condições que fazem com que nossa implementação para este gerador não termine e explicamos como *fair schedulers* (BRACHA; TOUEG, 1985) asseguram, com probabilidade 1, o término do mesmo.

Apesar de não utilizar consenso, a nossa implementação do gerador de visões seguro (Algoritmo 6) garante que os servidores apenas possam gerar uma sequência de visões auxiliares (ver adiante) fortemente relacionadas (Lema 12), a partir das quais qualquer servidor é capaz de escolher uma mesma visão w para ser reportada aos processos do sistema (Lema 13). Como veremos adiante, o gerador não terminará caso esta sequência seja vazia.

A ideia principal deste protocolo é permitir que os servidores façam propostas para a nova visão de tal forma que eventualmente todos os servidores corretos convirjam para a mesma visão mais atual, sem utilizar consenso. O protocolo permite que dois processos proponham diferentes composições para a nova visão, mas cada um destes deverá convergir para a mesma visão mais atual formada pela união destas composições, sem ferir nenhuma propriedade do sistema. Seja $next_i$ a composição da nova visão esperada pelo processo i. Quando i receber uma mensagem de outro processo j com sua proposta para nova visão $next_j$, i adiciona $next_j \setminus next_i$ em sua composição esperada para nova visão. O resultado é que a composição para nova visão esperada por i torna-se $next_i \cup next_j$. Este mesmo processamento acontece em j quando o mesmo receber $next_i$. Deste modo, todos os processos corretos convergem para a mesma composição de nova visão. Um aspecto fundamental deste protocolo é que servidores maliciosos não conseguem propor infinitos *updates* diferentes com o objetivo de evitar que esta convergência ocorra.

A Figura 9 ilustra os passos na obtenção de uma visão por este gerador, o que geralmente demanda 4 passos de comunicação. O passo crucial NEW-VIEW pode ser executado mais de uma vez para garantir convergência. Como os protocolos de leitura e escrita e de reconfiguração do QUINCUNX

executam em paralelo, o custo dos protocolos envolvidos na reconfiguração do QUINCUNX não afetam o desempenho das operações executadas pelos clientes. O Algoritmo 6 apresenta este protocolo.

O algoritmo utiliza a variável PROP_PR para armazenar as provas de que sua proposta é válida. Como veremos a seguir, um servidor i obtém um conjunto de assinaturas para validar sua proposta de nova visão. Deste modo, esta proposta somente é válida caso contenha todos os $updates$ para os quais as assinaturas foram obtidas, além de poder englobar outros $updates$ propostos por um outro servidor j (neste caso esta variável também contém as provas de que a proposta de j é válida).

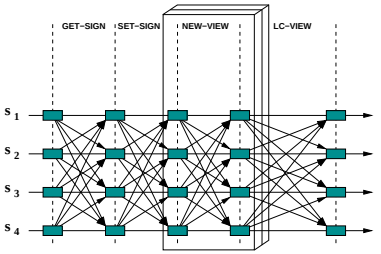


Figura 9 – Gerador de Visões Seguro.

A composição esperada para a nova visão é armazenada em NVENTR e a variável NVENTR_PR armazena as provas de que servidores propuseram esta composição para a nova visão. O algoritmo também utiliza uma variável para armazenar a última visão convergida LCVIEW por algum processo. Esta visão é chamada de *visão auxiliar*, pois apenas auxiliará na geração de uma nova visão do sistema. Uma visão auxiliar v_{aux} possui a seguinte estrutura $\langle\langle\langle ov, entries, \emptyset\rangle, lcv\rangle, \texttt{PLCV}\rangle$, onde: $\langle ov, entries, \emptyset\rangle$ representa uma visão normal (Seção 5.4), lcv é a última visão convergida conhecida quando da obtenção de v_{aux} (portanto também é uma visão auxiliar) e PLCV armazena as provas de que v_{aux} foi proposta por um quórum de servidores em $v_{aux}.ov$. Note que os campos lcv formam uma "corrente" entre as visões auxiliares (uma visão é armazenada "dentro" da visão seguinte, e assim por diante). Estas visões não são instaladas, apenas utilizadas como auxiliares na geração da visão atualizada, como veremos a seguir na apresentação dos passos para gerar uma visão atualizada w pelo gerador associado a uma visão v.

Certificando a proposta de nova visão. Para acelerar a convergência na geração da visão, mesmo com a presença de servidores maliciosos em v, cada servidor deve ser capaz de realizar uma única proposta de atualização de v. Cada servidor i deve obter um quórum ($v.q$) de assinaturas para sua proposta

de *updates* a serem computados na nova visão (linha 1), onde cada servidor assina um conjunto de *updates* propostos por i, para serem aplicados em v, uma única vez (passos GET-SIGN e SET-SIGN – Figura 9).

Algoritmo 6 Gerador de visões seguro (servidor $i \in v$)

variables: {Sets and arrays used in the protocol}
PROP_PR - proofs for the new view entries proposal
NVENTR, NVENTR_PR - expected next view entries and its corresponding proofs
LCVIEW - last converged view known

procedure *generate_view*$(v, updates)$
1: obtain signed tuples $\langle updates, v, i\rangle_*$ from $v.q$ different servers in v and store it on $\texttt{PROP\_PR}^v$
2: $\texttt{NVENTR}^v \leftarrow v.entries \cup updates$
3: $\texttt{LCVIEW}^v \leftarrow \perp$
4: $\forall j \in v, send\langle$NEW-VIEW$, \langle\langle v, \texttt{NVENTR}^v, \emptyset\rangle, \texttt{LCVIEW}^v\rangle_i, \texttt{PROP_PR}^v\rangle$ to j

Upon receipt of $\langle$NEW-VIEW$, \langle\langle v, entries, \emptyset\rangle, lcv\rangle_j, proof\rangle$ from j
5: **if** $isValid(entries, v, proof, j) \wedge verifyLCV(v, lcv) \wedge check(j, v, lcv, entries)$ **then**
6: **wait until** $\texttt{NVENTR}^v \neq \perp$
7: **if** $((\texttt{NVENTR}^v \neq entries) \wedge (entries \not\subset \texttt{NVENTR}^v)) \vee ((maxUpdated(\texttt{LCVIEW}^v, lcv) \neq \texttt{LCVIEW}^v) \vee$ not sent NEW-VIEW to $\texttt{LCVIEW}^v$ yet) **then**
8: $\texttt{NVENTR}^v \leftarrow \texttt{NVENTR}^v \cup entries$
9: $\texttt{PROP\_PR}^v \leftarrow \texttt{PROP\_PR}^v \cup \{proofs\}$
10: $\texttt{LCVIEW}^v \leftarrow maxUpdated(\texttt{LCVIEW}^v, lcv)$
11: $\texttt{NVENTR\_PR}^v \leftarrow \emptyset$
12: $\forall k \in v$, $send\langle$NEW-VIEW$, \langle\langle v, \texttt{NVENTR}^v, \emptyset\rangle, \texttt{LCVIEW}^v\rangle_i, \texttt{PROP_PR}^v\rangle$ to k
13: **end if**
14: **if** $(\texttt{NVENTR}^v = entries) \wedge equals(\texttt{LCVIEW}^v, lcv)$ **then**
15: $\texttt{NVENTR\_PR}^v \leftarrow \texttt{NVENTR\_PR}^v \cup \{\langle\langle v, entries, \emptyset\rangle, lcv\rangle_j\}$
16: **if** $|\texttt{NVENTR\_PR}^v| \geq v.q$ **then**
17: $\texttt{LCVIEW}^v \leftarrow \langle\langle\langle v, \texttt{NVENTR}^v, \emptyset\rangle, \texttt{LCVIEW}^v\rangle, \texttt{NVENTR\_PR}^v\rangle$
18: $\forall k \in v, send\langle$LC-VIEW$, \texttt{LCVIEW}^v\rangle$ to k
19: **end if**
20: **end if**
21: **end if**

Upon receipt of $\langle$LC-VIEW$, lcv\rangle$ from $lcv.ov.q$ different servers in $lcv.ov \wedge i \in lcv.ov$
22: $w \leftarrow extractView(lcv)$ {w is the generated view}
23: $new\_view(lcv.ov, w)$ {Notice that $lcv.ov = v$ (line 17)}

1. Function $isValid(entries, v, proof, j)$ verifies if: (1) $(\forall e \in entries) \rightarrow (\exists z: \#_{\langle u_z, v, z\rangle_*} proof \geq v.q \wedge e \in u_z) \vee (e \in v.entries)$; and (2) $\#_{\langle u_j, v, j\rangle_*} proof \geq v.q \wedge \forall e \in u_j \rightarrow e \in entries$.

2. $check(j, v, lcv, entries) \equiv (i, j \in v) \wedge (lcv = \perp \vee ((v.entries \subseteq lcv.entries) \wedge (lcv.entries \subseteq entries))$

3. Function $verifyLCV(v, lcv)$ verifies if: (1) $(\#_{\langle\langle lcv.ov, lcv.entries, \emptyset\rangle, lcv.lcv\rangle_*} lcv.PLCV \geq v.q) \wedge (lcv.ov = v)$ (line 17); or (2) $lcv = \perp$ (line 3).

4. Function $maxUpdated(lcv_1, lcv_2)$ returns the most updated last converged view by using the operator "$\subset$" (as a view). If $lcv_1 = lcv_2$, then it compares its last converged views (lcv) by calling recursively $maxUpdated(lcv_1.lcv, lcv_2.lcv)$ until reach $\perp$, i.e., if a recursive call contains two "views" equal to $\perp$, then lcv_1 and lcv_2 are equal.

5. Function $equals(lcv_1, lcv_2)$ follows the same principles of function $maxUpdated(lcv_1, lcv_2)$.

6. $extractView(lcv) \equiv$ if $(lcv.lcv = \perp)$ then return $\langle lcv.ov, lcv.entries, \emptyset\rangle$; else return $extractView(lcv.lcv)$

Estas assinaturas formam a prova de que a proposta é válida. Através das propriedades de intersecção dos quóruns, cada servidor apenas é capaz de fazer uma única proposta para nova visão para todos os servidores do sistema. Quando i obtém este conjunto de assinaturas, i determina a composição esperada para a nova visão (`NVENTR`), que será formada pela união dos *updates* propostos e das *entries* em v (linha 2). Além disso, i define sua última visão convergida (visão auxiliar) conhecida como sendo $\perp$ (linha 3), uma vez que a geração de uma nova visão está apenas iniciando.

Convergindo para a mesma visão. O protocolo descrito até agora garante que cada servidor apenas é capaz de gerar uma proposta válida para nova visão. No entanto, servidores podem gerar diferentes propostas e divergir na escolha da nova visão. As trocas de mensagens NEW-VIEW resolvem este problema, fazendo com que os servidores em v convirjam para uma sequência de visões auxiliares fortemente relacionadas, a partir das quais cada servidor é capaz de escolher e gerar a mesma visão w.

Após um servidor $j \in v$ preparar sua proposta para nova visão, j envia uma mensagem de NEW-VIEW para todos os servidores em v com esta proposta assinada (linha 4). Quando esta mensagem é recebida em um servidor i, i verifica se as *entries* (composição) propostas, bem como a última visão convergida, são válidas. Então, após i determinar sua proposta para nova visão (linha 6), dois cenários (não mutualmente exclusivos) são possíveis:

1. i fez uma proposta diferente para nova visão, onde a composição de nova visão esperada por i (`NVENTR`) não contém algum *update* da proposta recebida (algum *update* em *entries* não pertence à `NVENTR`) ou a última visão convergida conhecida por i (`LCVIEW`) está desatualizada (linhas 7-13): Então, i reinicia este passo considerando a composição esperada para a nova visão como sendo a união das duas propostas e atualiza sua última visão convergida, se for o caso.

2. i fez a mesma proposta (linhas 14-20): este é o caso normal, onde i coleta a proposta assinada recebida. Quando i obtém a prova de que um quórum fez esta mesma proposta, i determina uma nova última visão convergida lcv. Neste caso, dizemos que i *convergiu* para lcv.

Um servidor malicioso não é capaz de evitar que os outros servidores convirjam para a mesma visão auxiliar, pois, devido à certificação de propostas, o mesmo não é capaz de realizar infinitas propostas para novas visões.

Informando os processos sobre uma nova visão auxiliar convergida. Após um servidor correto $i \in v$ convergir para uma nova visão auxiliar lcv, i envia uma mensagem LC-VIEW para os servidores em v (note que $lcv.ov = v$) informando-os sobre lcv. Os servidores corretos aguardam por um quórum

destas mensagens, com a mesma lcv, para avançar no protocolo. Isto garante que pelo menos um quórum de servidores em v armazenaram lcv em suas variáveis LCVIEW, então uma possível visão auxiliar convergida a seguir conterá lcv como última visão convergida (Lema 12). Quando um quórum de servidores em v converge para lcv, dizemos que lcv foi *gerada* pelo gerador.

Os servidores corretos avançam no protocolo extraindo uma visão mais atual w da visão auxiliar gerada lcv. Esta visão w é a visão do sistema gerada e reportada para os servidores em v (linhas 22-23). Qualquer visão futuramente extraída de uma outra visão auxiliar gerada será igual a w. Basicamente, a função *extractView* navega pelos campos lcv das visões auxiliares até atingir a primeira visão auxiliar convergida v_{aux} (esta visão é determinada quando $v_{aux}.lcv = \perp$ e representa a primeira visão convergida por algum servidor) e então cria a nova visão w com sendo $\langle v_{aux}.ov, v_{aux}.entries, \emptyset \rangle$. Como qualquer visão auxiliar v'_{aux} gerada a seguir possui $v'_{aux}.lcv = v_{aux}$, uma navegação iniciando em v'_{aux} também terminará pela obtenção de w. Na verdade, poderemos ter $v'_{aux}.lcv = w_{aux}$ tal que $w_{aux} \neq v_{aux}$ (sendo v'_{aux} mais atual do que w_{aux}, que é mais atual do que v_{aux}), mas uma navegação a partir de w_{aux} atingirá v_{aux} e então também terminará com a obtenção de w. Assim, após uma visão w ser obtida de v_{aux} e reportada para os servidores em v, w é a única visão que pode ser resultado da execução do gerador de visões seguro associado à v (a função *extractView* é melhor explicada a seguir).

Sequência de visões auxiliares geradas. Dado um gerador relacionado a uma visão v, uma visão auxiliar v_{aux} é gerada por este gerador quando um quórum de servidores convergem para v_{aux}. Na prática isso significa que a visão w extraída de v_{aux} será a visão gerada por este gerador. Deste modo, este protocolo deve garantir que somente w é extraída de qualquer outra visão auxiliar da sequência de visões auxiliares geradas.

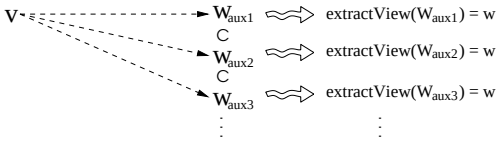


Figura 10 – Sequência de visões auxiliares geradas pelo gerador associado a uma visão v. A partir de qualquer uma destas visões auxiliares é possível extrair uma mesma visão w.

A Figura 10 mostra uma sequência de visões auxiliares geradas. Como é necessário um quórum de propostas para convergir para uma visão auxiliar, existe uma relação entre as mesmas: uma visão auxiliar gerada sempre

é mais atual do que as visões auxiliares anteriormente geradas. Note que é muito improvável desta sequência ser grande na prática, pois para isso as mensagens NEW-VIEW devem ser entregues seguindo um padrão muito específico. Além disso, após a visão do sistema ser atualizada (Seção 5.6.2.2), este gerador é finalizado.

A principal diferença entre este algoritmo e um protocolo de *membership* é que, apesar de existir uma sequência de visões auxiliares, não é possível saber, a priori, quais destas visões serão geradas e qual será extraída. Além disso, é possível que nenhuma visão auxiliar seja gerada, caso os processos não consigam convergir para uma mesma visão, como descrito adiante.

A Tabela 8 apresenta sequências de visões auxiliares geradas, pelo gerador associado a uma visão v, permitidas e não permitidas de ocorrerem no sistema. A tabela apresenta cenários para quando a visão auxiliar $v_{aux1} = \langle\langle\langle v, entries, \emptyset\rangle, \perp\rangle, lcvp\rangle$ (usaremos a notação $v_{aux1} = \langle\langle *, \perp\rangle, *\rangle$) não é gerada (i.e., não existe um quórum de servidores em v que convergiu para v_{aux1}) mas pelo menos um servidor de v convergiu para v_{aux1} e também cenários para quando v_{aux1} é gerada. É importante notar que após uma visão auxiliar v_{aux} ser gerada, esta visão é a única que pode ser usada como última visão convergida (campo lcv) na próxima visão gerada. Caso v_{aux} não seja gerada, v_{aux} pode ser usada ou não, de acordo com o quórum de mensagens NEW-VIEW usadas para gerar a visão auxiliar seguinte.

$1 < NP(v_{aux1}) < v.q$		$NP(v_{aux1}) \geq v.q$	
Permitida	Não Permitida	Permitida	Não Permitida
$v_{aux2} : \langle\langle *, v_{aux1}\rangle, *\rangle$ $v_{aux3} : \langle\langle *, v_{aux2}\rangle, *\rangle$ $v_{aux4} : \langle\langle *, v_{aux3}\rangle, *\rangle$	$v_{aux2} : \langle\langle *, \perp\rangle, *\rangle$ $v_{aux3} : \langle\langle *, \perp\rangle, *\rangle$ $v_{aux4} : \langle\langle *, v_{aux3}\rangle, *\rangle$	$v_{aux1} : \langle\langle *, \perp\rangle, *\rangle$ $v_{aux2} : \langle\langle *, v_{aux1}\rangle, *\rangle$ $v_{aux3} : \langle\langle *, v_{aux2}\rangle, *\rangle$	$v_{aux1} : \langle\langle *, \perp\rangle, *\rangle$ $v_{aux2} : \langle\langle *, \perp\rangle, *\rangle$ $v_{aux3} : \langle\langle *, v_{aux2}\rangle, *\rangle$
$v_{aux2} : \langle\langle *, \perp\rangle, *\rangle$ $v_{aux3} : \langle\langle *, v_{aux2}\rangle, *\rangle$ $v_{aux4} : \langle\langle *, v_{aux3}\rangle, *\rangle$	$v_{aux2} : \langle\langle *, \perp\rangle, *\rangle$ $v_{aux3} : \langle\langle *, v_{aux1}\rangle, *\rangle$ $v_{aux4} : \langle\langle *, v_{aux3}\rangle, *\rangle$	$v_{aux1} : \langle\langle *, \perp\rangle, *\rangle$ $v_{aux2} : \langle\langle *, v_{aux1}\rangle, *\rangle$ $v_{aux4} : \langle\langle *, v_{aux2}\rangle, *\rangle$	$v_{aux1} : \langle\langle *, \perp\rangle, *\rangle$ $v_{aux2} : \langle\langle *, v_{aux1}\rangle, *\rangle$ $v_{aux4} : \langle\langle *, v_{aux1}\rangle, *\rangle$

Tabela 8 – Sequências de visões auxiliares, geradas pelo gerador associado a uma visão v, permitidas e não permitidas de ocorrerem no sistema. $NP(v_{aux1})$ representa o número de servidores de v que convergiram para v_{aux1}.

Dado um gerador de visões associado a uma visão v, esta função navega pelos campos lcv das visões auxiliares geradas, até atingir a primeira visão auxiliar v_{aux} convergida por algum servidor em v, o que ocorre quando $v_{aux}.lcv = \perp$ (linha 3), i.e., v_{aux} é a primeira visão convergida por algum servidor de v. Após isso, esta função define uma nova visão $w = \langle v, v_{aux}.entries, \emptyset\rangle$. Como $v.entries \subset v_{aux}.entries$ (linha 2), temos que $v \subset w$.

Por exemplo, na primeira sequência permitida de visões auxiliares geradas $v_{aux2} \rightarrow v_{aux3} \rightarrow v_{aux4}$ (ver Tabela 8), uma navegação começando a partir de qualquer uma destas visões retorna uma visão $w = \langle v, v_{aux1}.entries, \emptyset\rangle$. Na

segunda sequência, uma navegação começando tanto em v_{aux2}, v_{aux3} ou v_{aux4} retorna uma visão $w = \langle v, v_{aux2}.entries, \emptyset \rangle$. Por outro lado, aplicando esta função na primeira sequência não permitida da segunda coluna, temos que $extractView(v_{aux4}) = extractView(v_{aux3}) = w$, tal que $w = \langle v, v_{aux3}.entries, \emptyset \rangle$. No entanto, $extractView(v_{aux2}) = w'$, tal que $w' = \langle v, v_{aux2}.entries, \emptyset \rangle$. Como podemos ter que $v_{aux2}.entries \neq v_{aux3}.entries$, teremos que $w \neq w'$ e a propriedade de exatidão forte do gerador não é satisfeita. Veja que isto não acontece no sistema, uma vez que após um quórum de servidores em v convergirem para v_{aux2}, é impossível que outro quórum de servidores em v convirja para v_{aux3} tal que $v_{aux3}.lcv \neq v_{aux2}$ (Lema 12).

5.6.1.2.1 Terminação do gerador

Um gerador de visões seguro pode não terminar caso a entrega das mensagens NEW-VIEW (através destas mensagens os servidores convergem para uma visão auxiliar) siga um determinado padrão, onde nenhum quórum de servidores corretos associados a este gerador consegue convergir para a mesma visão auxiliar, i.e., uma visão auxiliar não é gerada. Embora pouco provável, é possível que este padrão seja observado no sistema.

Esta característica não quebra as propriedades definidas para esta classe de geradores de visões, que deve prover apenas terminação fraca (Seção 5.6.1). Como em determinadas situações este gerador pode não gerar uma visão atualizada, as propriedades de terminação definidas para as reconfigurações (Seção 5.3) não são garantidas pelo simples emprego desta classe de geradores. No entanto, como veremos a seguir, escalonadores justos (*fair schedulers*) (BRACHA; TOUEG, 1985) garantem a terminação deste gerador e, por consequência, da reconfiguração que o emprega na obtenção de visões atualizadas.

Suposição 6 (Fair Schedulers) *Mensagens NEW-VIEW são entregues por fair schedulers.*

Em um sistema assíncrono onde as trocas de mensagens podem ser organizadas em passos que são executados (ou re-executados) até que alguma condição seja alcançada no sistema, os *fair schedulers* (BRACHA; TOUEG, 1985) garantem que em alguma destas execuções todos os processos corretos do sistema recebem o mesmo conjunto de mensagens, assegurando a evolução da computação. Este conceito apenas pode ser aplicado em sistemas onde a probabilidade de um processo correto p receber uma mensagem de outro processo correto q é uma constante positiva e é independente da probabilidade de p receber outra mensagem de outro processo z. Formalmente, este conceito

é definido da seguinte maneira. Seja $R(q,p,r)$ o evento do processo p receber uma mensagem do processo q no passo r, então um escalonador (*scheduler*) é considerado justo (*fair*) caso as seguintes condições forem verificadas:

1. Para qualquer dois processos p e q, e passo r, existe uma constante positiva c tal que $Pr[R(q,p,r)] > c$, i.e., a probabilidade de $R(q,p,r)$ ocorrer é maior do que c.

2. Para quaisquer três processos distintos p, q e z, e passo r, os eventos $R(q,p,r)$ e $R(q,z,r)$ são independentes.

Estas condições garantem que, para qualquer passo r, exite uma probabilidade constante $pr > 0$ de todos os processos receberem o mesmo quórum de mensagens (neste caso, de mensagens NEW-VIEW). A seguir é apresentado um cenário onde uma visão auxiliar não é gerada e discutido como *fair schedulers* resolvem este problema.

Considere o cenário apresentado na Tabela 9, onde quatro servidores ($\{s_1,s_2,s_3,s_4\} \in v$) estão executando o gerador de reconfigurações associado a uma visão v e os servidores s_1,s_2 e s_3 fazem um proposta de composição de nova visão e_1 (campo *entries*) enquanto s_4 propõe e_2, tal que $e_1 \subset e_2$ (note que $v.entries \subset e_1 \subset e_2$).

tempo →				
s_1	$\langle\langle v,e_1,\emptyset\rangle,\perp\rangle,*\rangle^p$	$w_{aux1} : \langle\langle v,e_1,\emptyset\rangle,\perp\rangle,*\rangle$	$\langle\langle v,e_2,\emptyset\rangle,w_{aux1}\rangle,*\rangle^p$	
s_2	$\langle\langle v,e_1,\emptyset\rangle,\perp\rangle,*\rangle^p$	$\langle\langle v,e_2,\emptyset\rangle,\perp\rangle,*\rangle^p$		$\langle\langle v,e_2,\emptyset\rangle,w_{aux1}\rangle,*\rangle^p$
s_3	$\langle\langle v,e_1,\emptyset\rangle,\perp\rangle,*\rangle^p$	$\langle\langle v,e_2,\emptyset\rangle,\perp\rangle,*\rangle^p$		$\langle\langle v,e_2,\emptyset\rangle,w_{aux1}\rangle,*\rangle^p$
s_4	$\langle\langle v,e_2,\emptyset\rangle,\perp\rangle,*\rangle^p$		$w_{aux2} : \langle\langle v,e_2,\emptyset\rangle,\perp\rangle,*\rangle$	
tempo (continuação) →				
s_1	$w_{aux21} : \langle\langle v,e_2,\emptyset\rangle,w_{aux1}\rangle,*\rangle$		$\langle\langle v,e_2,\emptyset\rangle,v_{aux21}\rangle,*\rangle^p$	
s_2		$\langle\langle v,e_2,\emptyset\rangle,v_{aux2}\rangle,*\rangle^p$		$\langle\langle v,e_2,\emptyset\rangle,v_{aux21}\rangle,*\rangle^p$
s_3		$\langle\langle v,e_2,\emptyset\rangle,v_{aux2}\rangle,*\rangle^p$		$\langle\langle v,e_2,\emptyset\rangle,v_{aux21}\rangle,*\rangle^p$
s_4	$\langle\langle v,e_2,\emptyset\rangle,v_{aux2}\rangle,*\rangle^p$		$w_{aux22} : \langle\langle v,e_2,\emptyset\rangle,v_{aux2}\rangle,*\rangle$	

Tabela 9 – Gerador de visões sem terminação. $\langle ...\rangle^p$ representa uma proposta e $w_{aux*} : \langle ...\rangle$ representa que determinado servidor convergiu para w_{aux*}.

Neste cenário, é possível que s_1 convirja para uma visão auxiliar $w_{aux1} : \langle\langle\langle v,e_1,\emptyset\rangle,\perp\rangle,*\rangle$ (pois s_1,s_2 e s_3 assinaram tal proposta) e que s_2 e s_3 mudem suas propostas para $\langle\langle\langle v,e_2,\emptyset\rangle,\perp\rangle,*\rangle^p$ antes de convergirem para w_{aux1} (pois receberam a proposta de s_4 antes disto). Por outro lado, ao receber esta proposta diferente, s_1 faz uma nova proposta $\langle\langle\langle v,e_2,\emptyset\rangle,w_{aux1}\rangle,*\rangle^p$ uma vez que w_{aux1} está armazenado em sua variável `LCVIEW` e é mais atual do que $\perp$. Caso

s_4 receba as propostas de s_2 e s_3 antes da proposta de s_1, s_4 converge para uma visão auxiliar $w_{aux2} : \langle\langle\langle v, e_2, \emptyset\rangle, \perp\rangle, *\rangle$. Novamente, caso s_2 e s_3 receberem a proposta de s_1 antes de convergirem para w_{aux2}, estes servidores mudam suas propostas para $\langle\langle\langle v, e_2, \emptyset\rangle, w_{aux1}\rangle, *\rangle^p$, e este ciclo pode ser repetido indefinidamente de tal forma que nenhuma visão auxiliar é gerada (um quórum de servidores não converge para a mesma visão auxiliar).

Isto acontece devido a dois fatores: (1) os servidores s_2 e s_3 sempre mudam suas propostas antes de convergirem para alguma visão auxiliar, i.e., não conseguem convergir para nenhuma visão auxiliar; e (2) cada um dos servidores s_1 e s_4 converge para sua visão auxiliar proposta antes de receber a proposta do outro com uma última visão convergida mais atual. *Fair schedulers* (BRACHA; TOUEG, 1985) garantem, com probabilidade 1, que estes dois casos não se repetirão para sempre no sistema, de forma que existe um tempo após o qual os servidores recebem o mesmo conjunto de mensagens (propostas) e uma visão auxiliar é gerada.

5.6.1.2.2 Corretude

A seguir apresentamos as provas de corretude do gerador de visões seguro implementado pelo Algoritmo 6.

Lema 12 *Seja um gerador de visões seguro gv_s implementado pelo Algoritmo 6 e associado a uma visão v. Após uma visão auxiliar w_{aux} ser gerada por gv_s, os servidores em v apenas podem convergir para w_{aux} ou para uma visão auxiliar mais atual w'_{aux}, tal que $w'_{aux}.lcv = w_{aux}$.*

Prova. Considere que uma visão auxiliar w_{aux} é gerada por gv_s. Então, pelo menos um quórum de servidores em v convergiram para w_{aux}, armazenando esta visão em `LCVIEW` (linha 17). Além disso, nenhum servidor possui armazenado em `LCVIEW` uma visão mais atualizada do que w_{aux}, caso contrário os servidores não convergiriam para w_{aux} uma vez que é necessário um quórum de mensagens NEW-VIEW para convergir para alguma visão auxiliar e, então, pelas propriedades de intersecção dos quóruns, seria impossível algum servidor convergir para w_{aux}. Agora, considerando que um servidor $i \in v$ converge para uma visão auxiliar w'_{aux}, temos dois casos:

1. $w'_{aux} = w_{aux}$: se i convergir para w_{aux}, então i também propôs w_{aux} (ou uma visão auxiliar menos atual do que w_{aux} e mudou sua proposta para w_{aux} quando recebeu alguma proposta para w_{aux} – linhas 7-13), e recebeu as mensagens usadas para gerar w_{aux} (linhas 14-20).

2. $w'_{aux} \neq w_{aux}$: Como pelo menos $v.q$ servidores já propuseram w_{aux}, pela

propriedade de intersecção de quóruns, e como é necessário $v.q$ propostas para convergir para alguma visão auxiliar, w'_{aux} é mais atual do que w_{aux}. Além disso, por estas mesmas propriedades, pelo menos um servidor correto em v enviou a mensagem NEW-VIEW com w_{aux} como proposta para lcv de w'_{aux} (linha 12) antes de i convergir para w'_{aux} e, como w_{aux} era a visão auxiliar mais atual, i convergiu para w'_{aux}, tal que $w'_{aux}.lcv = w_{aux}$.

Deste modo, após w_{aux} ser gerada por gv_s, os servidores em v apenas podem convergir para w_{aux} ou para uma visão auxiliar mais atual w'_{aux}, tal que $w'_{aux}.lcv = w_{aux}$. $\square_{Lema\ 12}$

Lema 13 *Seja um gerador de visões seguro gv_s implementado pelo Algoritmo 6 e associado a uma visão v. Caso um servidor correto de v extrair uma visão w, então w é a única visão que pode ser extraída pelos servidores em v.*

Prova. Considere que na execução de gv_s um servidor correto $i \in v$ extrai uma visão w. Então, os servidores em v geraram uma visão v_{aux}, tal que $extractView(v_{aux}) = w$. Pelo Lema 12, os servidores em v apenas podem convergir para v_{aux} ou para uma visão auxiliar mais atual v'_{aux}, tal que $v'_{aux}.lcv = v_{aux}$. Então, por construção, $extractView(v'_{aux}) = extractView(v_{aux}) = w$. Por indução, servidores em v apenas podem convergir para uma visão auxiliar mais atual v''_{aux}, tal que $extractView(v''_{aux}) = w$. $\square_{Lema\ 13}$

Teorema 4 *O gerador de visões seguro implementado pelo Algoritmo 6 satisfaz as propriedades de exatidão forte, terminação fraca e não trivialidade.*

Prova. A prova da propriedade de *exatidão forte* é derivada diretamente a partir do Lema 13. Por outro lado, a propriedade de *terminação fraca*, a qual estabelece que uma visão pode não ser gerada (mas não impede que isso aconteça), é garantida tanto para os casos onde este gerador de visões termina quanto para os cenários onde tal gerador não termina. Já a propriedade de *não trivialidade* é garantida pela forma como os servidores definem sua proposta inicial para nova visão (linha 2) e por ser necessário um quórum de propostas iguais para uma visão auxiliar ser gerada (linha 16). $\square_{Teorema\ 4}$

Lema 14 *Seja um gerador de visões seguro gv_s implementado pelo Algoritmo 6 e associado a uma visão v. Então, fair schedulers garantem, com probabilidade 1, a propriedade de terminação forte para gv_s.*

Prova. Considere a execução de gv_s onde um quórum de servidores em v não consegue convergir para uma mesma visão auxiliar v_{aux}. Isto apenas é

possível caso estes servidores recebam diferentes conjuntos de mensagens NEW-VIEW antes de convergirem para alguma visão auxiliar (linha 17) ou mudarem suas propostas (linhas 7-13), caso contrário gerariam v_{aux}. Os *fair schedulers* garantem, com probabilidade 1, que as trocas destas mensagens NEW-VIEW atingem um ponto na execução onde todos os servidores recebem o mesmo conjunto destas mensagens. Desta forma, todos os servidores corretos convergem para a mesma v_{aux} e esta visão auxiliar é gerada. Com isso, também é gerada uma visão para atualizar o sistema w, tal que $extractView(v_{aux}) = w$. $\square_{Lema\ 14}$

5.6.1.3 Gerador de Visões Vivo

O gerador de visões vivo recebe este nome por sempre terminar, sendo outro gerador implementável em ambientes completamente assíncronos. Este gerador possui as propriedades de exatidão fraca e terminação forte (Seção 5.6.1), garantindo que *alguma visão sempre é gerada* (terminação forte), mesmo sendo possível que *mais de uma visão* seja gerada (exatidão fraca).

Apesar de não empregar consenso, a nossa implementação deste gerador (Algoritmo 7) garante que os servidores apenas geram visões fortemente relacionadas (Lema 15): dado um gerador vivo gv associado a uma visão v, para qualquer duas visões w e w' geradas por gv, temos que $w \subset w'$ ou $w' \subset w$.

A forma de funcionamento, bem como as váriaveis utilizadas neste algoritmo, são praticamente iguais ao algoritmo que implementa o gerador de visões seguro (Seção 5.6.1.2). No entanto, como não é necessário garantir que apenas uma única visão seja gerada, este protocolo é mais simples e sempre termina gerando alguma visão em um dado servidor.

Este protocolo, apresentado pelo Algoritmo 7, geralmente requer 3 passos de comunicação para gerar uma visão. No entanto, como no gerador de visões seguro, pode ser necessário executar mais de uma vez o passo crucial NEW-VIEW. A seguir é descrito como um gerador de visões vivo, associado a uma visão v, gera novas visões para atualização do sistema.

Como no protocolo descrito anteriormente (Seção 5.6.1.2), cada servidor correto $i \in v$ certifica sua proposta e envia uma mensagem NEW-VIEW para todos os servidores de v. Note que aqui a proposta dos servidores contém uma visão do sistema e não uma visão auxiliar (Seção 5.6.1.2), que não é utilizada por este protocolo.

De forma semelhante ao protocolo anterior, os servidores em v atualizam suas propostas caso recebam propostas diferentes (linhas 6-11) e convergem para uma nova visão w assim que os mesmos receberem um quórum de propostas para w (linhas 12-17). Como sempre temos pelo menos um quórum

de servidores corretos em v, é garantido que cada servidor $i \in v$ sempre acabará gerando pelo menos uma visão. Vale ressaltar que servidores poderão gerar visões diferentes. No entanto, através das propriedades de intersecção dos quóruns, é garantido que existe uma relação entre qualquer duas visões w e w' geradas por este gerador: $w \subset w'$ ou $w' \subset w$. Esta relação faz com que estas visões possam ser organizadas em uma sequência, de acordo com o operador "$\subset$".

Algoritmo 7 Gerador de visões vivo (servidor $i \in v$)

variables: {Sets and arrays used in the protocol}
PROP_PR - proofs for the new view entries proposal
NVENTR, NVENTR_PR - expected next view entries and its corresponding proofs

procedure *generate_view*$(v, updates)$
1: obtain signed tuples $\langle updates, v, i\rangle_*$ from $v.q$ different servers in v and store it on $\texttt{PROP\_PR}^v$
2: $\texttt{NVENTR}^v \leftarrow v.entries \cup updates$
3: $\forall j \in v, send\langle \text{NEW-VIEW}, \langle v, \texttt{NVENTR}^v, \emptyset\rangle_i, \texttt{PROP\_PR}^v\rangle$ to j

Upon receipt of $\langle \text{NEW-VIEW}, \langle v, entries, \emptyset\rangle_j, proof\rangle$ from j
4: **if** $isValid(\langle v, entries, \emptyset\rangle_j, proof) \wedge (i, j \in v)$ **then**
5: **wait until** $\texttt{NVENTR}^v \neq \perp$
6: **if** $(\texttt{NVENTR}^v \neq entries) \wedge (entries \not\subset \texttt{NVENTR}^v)$ **then**
7: $\texttt{NVENTR}^v \leftarrow \texttt{NVENTR}^v \cup entries$
8: $\texttt{PROP\_PR}^v \leftarrow \texttt{PROP\_PR}^v \cup \{proofs\}$
9: $\texttt{NVENTR\_PR}^v \leftarrow \emptyset$
10: $\forall k \in v$, $send\langle \text{NEW-VIEW}, \langle v, \texttt{NVENTR}^v, \emptyset\rangle_i, \texttt{PROP\_PR}^v\rangle$ to k
11: **end if**
12: **if** $\texttt{NVENTR}^v = entries$ **then**
13: $\texttt{NVENTR\_PR}^v \leftarrow \texttt{NVENTR\_PR}^v \cup \{\langle v, entries, \emptyset\rangle_j\}$
14: **if** $(|\texttt{NVENTR\_PR}^v| \geq v.q)$ **then**
15: $new\_view(v, \langle v, entries, \texttt{NVENTR\_PR}^v\rangle)$
16: **end if**
17: **end if**
18: **end if**

Function $isValid(\langle v, entries, \emptyset\rangle_j, proof)$ verifies if:
a) $(\forall e \in entries) \rightarrow (\exists z: \#_{\langle u_z, v, z\rangle_*} proof \geq v.q \wedge e \in u_z) \vee (e \in v.entries)$; and
b) $\#_{\langle u_j, v, j\rangle_*} proof \geq v.q \wedge \forall e \in u_j \rightarrow e \in v.entries$;

A Figura 11 apresenta a sequência de visões que pode ser gerada. Como é necessário um quórum de propostas para convergir para uma visão, esta sequência é limitada e conterá no máximo $|v.members| - v.q + 1$ visões (Lema 15): a primeira visão obtida pela computação dos *updates* de um quórum de servidores ($v.q$) e $|v.members| - v.q$ outras visões obtidas pela adição dos *updates* de cada um dos outros servidores restantes, que devem possuir diferentes propostas para nova visão. Novamente, é muito improvável desta sequência ser gerada na prática, pois para isso é necessário que a entrega das mensagens NEW-VIEW siga um padrão muito específico.

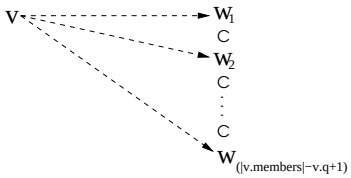


Figura 11 – Sequência de visões geradas pelo gerador associado a visão v.

5.6.1.3.1 Corretude

A seguir apresentamos as provas de corretude do gerador de visões vivo implementado pelo Algoritmo 7.

Lema 15 *Seja um gerador de visões vivo gv_v implementado pelo Algoritmo 7 e associado a uma visão v. Então, os servidores em v geram no máximo $|v.members| - v.q + 1$ visões, que apresentam a seguinte relação: $w_1 \subset w_2 \subset \ldots \subset w_{(|v.members|-v.q)} \subset w_{(|v.members|-v.q+1)}$.*

Prova. Considere que na execução de gv_v uma primeira visão w_1 é gerada por algum servidor de v. Então, é necessário que pelo menos um quórum ($v.q$) de servidores de v tenham proposto w_1 (linha 14). Além disso, pelo passos de GET-SIGN e SET-SIGN, cada servidor em v é capaz de propor apenas a mesma composição para nova visão, garantindo que composições de pelo menos um quórum de servidores de v são computadas em w_1. Considere que estes servidores formam o conjunto `COMP`: servidores cujas propostas já foram computadas por um quórum de servidores. Considere também que cada um dos $|v.members| - v.q$ servidores restantes de v armazena diferentes composições para nova visão, ainda não computadas em w_1. Estes servidores foram o conjunto `NCOMP`: servidores cujas propostas ainda não foram computadas. Agora, considere que um servidor $i \in$ `NCOMP` envia uma mensagem NEW-VIEW com sua proposta de composição u_i, que é recebida pelos servidores em `COMP`, os quais juntamente com i geram uma nova visão mais atual w_2. Pelas propriedades de intersecção dos quóruns, temos que $w_2.entries = w_1.entries \cup u_i$, uma vez que pelo menos um servidor correto possuía $\texttt{NVENTR} = w_1.entries$ antes de receber a mensagem de i. Então, temos que: $\texttt{COMP} = \texttt{COMP} \cup \{i\}$ e $\texttt{NCOMP} = \texttt{NCOMP} \setminus \{i\}$. Por indução, o último servidor $z \in$ `NCOMP` faz com que os servidores gerem a visão $w_{(|v.members|-v.q+1)}$, onde $w_{(|v.members|-v.q+1)}.entries = w_{(|v.members|-v.q)}.entries \cup u_z$. Então, temos que $w_1 \subset w_2 \subset \ldots \subset w_{(|v.members|-v.q)} \subset w_{(|v.members|-v.q+1)}$. $\square_{Lema\ 15}$

Lema 16 *Seja um gerador de visões vivo gv_v implementado pelo Algoritmo 7 e associado a uma visão v. Cada servidor correto $i \in v$ gera pelo menos uma nova visão.*

Prova. Para *i* gerar uma visão atualizada *w* é necessário que o mesmo colete *v.q* propostas para *w* (linha 14). Como sempre existem pelo menos *v.q* servidores corretos em *v*, e as propostas apenas são alteradas pela inclusão de *updates* onde cada servidor pode propor apenas o mesmo conjunto de *updates* (passos GET-SIGN e SET-SIGN), é garantido que *i* acabará gerando *w*. $\square_{Lema\ 16}$

Teorema 5 *O gerador de visões vivo implementado pelo Algoritmo 7 satisfaz as propriedades de exatidão fraca, terminação forte e não trivialidade.*

Prova. A prova da propriedade de *terminação forte* é derivada diretamente a partir do Lema 16. Por outro lado, os servidores podem coletar diferentes quóruns de mensagens NEW-VIEW e gerar diferentes visões. No entanto, pelo Lema 15, este protocolo garante a propriedade de *exatidão fraca*. Já a propriedade de *não trivialidade* é garantida pela forma como os servidores definem sua proposta inicial para nova visão (linha 2) e por ser necessário um quórum de propostas iguais para uma visão ser gerada (linha 14). $\square_{Teorema\ 5}$

5.6.2 Reconfiguração do Sistema

Esta seção apresenta protocolos para atualizar o conjunto de servidores que implementa o sistema de quóruns, através da utilização dos geradores de visões discutidos anteriormente. Estes protocolos não são executados cada vez que um processo solicita um *join* e/ou *leave* do sistema, mas periodicamente quando o tempo de duração de determinada visão se esgotar e existirem requisições pendentes para reconfiguração do sistema. Esta abordagem diminui as necessidades de processamento ao executar as requisições de reconfigurações em lotes dentro de uma única atualização de visão. Além disso, esta solução desacopla os protocolos de reconfiguração dos protocolos para acesso (leitura e escrita) ao sistema: as reconfigurações do QUINCUNX podem ser executadas em paralelo com os protocolos de leitura e escrita no registrador, melhorando o desempenho do sistema quando comparado com soluções onde estes protocolos são fortemente acoplados (ex.: (AGUILERA et al., 2009, 2011)).

5.6.2.1 Iniciando o Gerador de Visões

Esta seção discute os procedimentos que fazem com que todos os servidores da visão corrente do sistema *cv* iniciem o gerador de visões associado a esta visão, a fim de reconfigurar o sistema. Este protocolo, apresentado no Algoritmo 8, é utilizado para iniciar qualquer gerador de visões, sendo que apenas os procedimentos após as visões serem geradas são diferentes e dependem do gerador utilizado (seções 5.6.2.2 e 5.6.2.3). Primeiramente, processos devem requisitar sua entrada ou saída do sistema, enquanto *cv* estiver ativa. Após o tempo de duração de *cv* se esgotar, o seu gerador de visões associado é iniciado para que novas visões sejam geradas e o sistema reconfigurado.

Algoritmo 8 Iniciando o gerador de visões (servidor i)

variables: {Sets and view generator object used in the protocol}
`RECV` - set of received updates
`VIEW_GEN` - the view generator

procedure *join*(v)
1: $\forall j \in v, send\langle$RECONFIG$, \langle +, i\rangle_i, v\rangle$ to j
2: **wait** for $v.q$ $\langle$REC-CONFIRM$\rangle$ replies from different servers in v

procedure *leave*()
3: $\forall j \in cv, send\langle$RECONFIG$, \langle -, i\rangle_i, cv\rangle$ to j
4: **wait** for $cv.q$ $\langle$REC-CONFIRM$\rangle$ replies from different servers in cv

Upon receipt of $\langle$RECONFIG$, \langle *, j\rangle_j, cv\rangle$ from $j \wedge \langle *, j\rangle_j \notin cv$
5: `RECV` $\leftarrow$ `RECV` $\cup \{\langle *, j\rangle_j\}$
6: *send*$\langle$REC-CONFIRM$\rangle$ to j

Upon ((timeout for cv and `RECV` $\neq \emptyset$) $\vee$ (start)) $\wedge$ (not started reconfig. for cv)
7: $\forall j \in cv, send\langle$START-RECONF$,$ `RECV`$, cv\rangle$ to j

Upon receipt of $\langle$START-RECONF$, recv, cv\rangle$ from $j \in cv \wedge (\forall u \in recv \rightarrow u \notin cv)$
8: `RECV` $\leftarrow$ `RECV` $\cup$ *recv*
9: **if** (received START-RECONF from $cv.f + 1$ servers) **then** start $\leftarrow$ *true* **end if**

Upon receipt of $\langle$START-RECONF$, recv, cv\rangle$ from $cv.q$ different servers in cv
10: `VIEW_GEN`.*generate_view*(cv, `RECV`)

Requisitando a entrada e/ou saída do sistema. Caso um servidor j desejar entrar no sistema, o mesmo deve encontrar uma visão com algum(s) membro(s) em comum com a visão corrente do sistema *cv* e executar a operação de *join* (linas 1-2), onde j envia a tupla assinada $\langle +, j\rangle_j$ indicando que deseja entrar no sistema. Por outro lado, para sair do sistema, j executa a operação de *leave* (linhas 3-4) e envia a tupla assinada $\langle -, j\rangle_j$. Quando um servidor i receber estas mensagens, i verifica se ambos os servidores (j e i) estão usando a mesma visão mais atual. Caso a visão de j esteja desatualizada, i envia a visão corrente para j, o qual executa novamente sua operação utilizando a visão atualizada (por simplicidade, este processamento não aparece no algoritmo). Do contrário, e caso j ainda não executou a operação de *join* ou *leave*, i adiciona esta requisição de atualização em seu conjunto de *updates*

(RECV), para ser processada na próxima reconfiguração do sistema, e envia uma mensagem para j confirmando que a requisição foi recebida (linhas 5-6).

Iniciando o gerador de visões. Cada servidor $i \in cv$ inicia uma reconfiguração para cv através do envio de uma mensagem START-RECONF (linha 7), o que acontece quando o tempo de duração de cv se esgotar e i possuir *updates* para propor (caso contrário o *timeout* para cv é reiniciado) ou quando i receber mais do que $cv.f$ destas mensagens de outros servidores em cv (linha 9), garantindo que o tempo de duração de cv se esgotou em pelo menos um servidor correto. Assim, servidores maliciosos não são capazes de iniciar uma reconfiguração prematuramente.

Cada mensagem START-RECONF contêm a visão corrente do emissor e o conjunto de *updates* proposto para ser aplicado nesta visão. Quando um servidor i recebe estas mensagens, i adiciona os *updates* recebidos em seu conjunto RECV (linha 8), a fim de realizar uma proposta com todos os *updates* conhecidos e, principalmente, para garantir que i iniciará a reconfiguração quando o próximo *timeout* para cv se esgotar, pois $\texttt{RECV} \neq \emptyset$. Um servidor i espera por $cv.q$ destas mensagens para então iniciar seu gerador de visões (linha 10), propondo todos os *updates* conhecidos até o momento para atualização de cv. Desta forma, todos os servidores corretos de cv iniciarão o gerador de visões associado a esta visão.

5.6.2.1.1 Corretude

A seguir apresentamos lemas que provam algumas propriedades do Algoritmo 8, que inicializa os geradores de visões.

Lema 17 *Dada uma visão v que foi instalada no sistema, os servidores $i \in v$ eventualmente iniciam o gerador de visões gv, associado a v, caso existam updates pendentes originados pela execução das operações de join ou leave por processos corretos.*

Prova. Considere uma visão v que foi instala no sistema e que existem *updates* pendentes originados pela execução das operações de *join* ou *leave* por processos corretos. Neste caso, os servidores em v acabarão por receber as requisições de *updates* (linhas 5 e 8) de forma que acabará por ocorrer um *timeout* para v onde $\texttt{RECV} \neq \emptyset$. Desta forma, cada servidor correto $i \in v$ acabará por enviar uma mensagem START-RECONF. Como v possui pelo menos um quórum de servidores corretos, é garantido que todos os servidores corretos eventualmente iniciam gv (linha 10). $\square_{Lema\ 17}$

Lema 18 *Dada uma visão v que foi instalada no sistema, um servidor malicioso $i \in v$ não é capaz de fazer um servidor correto $j \in v$ iniciar o gerador de visões gv, associado a v, prematuramente.*

Prova. Considere que um servidor malicioso $i \in v$ envia uma mensagem de START-RECONF antes que o *timeout* para v se esgote. Como um servidor correto $j \in v$ apenas envia esta mensagem quanto o seu *timeout* para v se esgotar (linha 7) ou receber $v.f+1$ destas mensagens (linha 9), nunca teremos um quórum $v.q$ destas mensagens antes que o *timeout* para v se esgote em pelo menos um servidor correto de v. Deste modo, gv não pode ser iniciado prematuramente (linha 10). $\square_{Lema\ 18}$

Lema 19 *Caso existam updates pendentes no sistema, reconfigurações são iniciadas periodicamente.*

Prova. Esta prova é obtida diretamente a partir dos Lemas 17 e 18. $\square_{Lema\ 19}$

5.6.2.2 Reconfiguração com Geradores de Visões Seguros

Esta seção apresenta o protocolo para reconfiguração do sistema através de um gerador de visões com a propriedade de exatidão forte, a qual estabelece que apenas uma única visão pode ser gerada. Deste modo, tanto um gerador de visões perfeito $\mathscr{P}$ (Seção 5.6.1.1) quanto um gerador de visões seguro $\mathscr{S}$ (Seção 5.6.1.2) podem ser empregados nesta solução.

Neste protocolo, apresentado no Algoritmo 9, para uma visão do sistema v ser válida (função $verify(v)$ usada pelos processos para verificar a validade das visões) é necessário que $v.\mathrm{P}$ contenha pelo menos $v.ov.q$ mensagens INSTALL-VIEW (ver adiante) devidamente assinadas por diferentes servidores de $v.ov$, indicando que o sistema reconfigurou de $v.ov$ para v. Além disso, a validade de $v.ov$ também deve ser verificada.

Instalando uma visão atualizada w. Após o gerador de visões gerar uma visão mais atualizada w para ser instalada no sistema, cada servidor $i \in cv$ envia uma mensagem de INSTALL-VIEW para todos os servidores em $cv \cup w$ (ou $w.ov \cup w$, pois ainda temos que $w.ov = cv$), informando estes processos sobre a nova visão a ser instalada no sistema. Esta mensagem é assinada, uma vez que a mesma é utilizada como prova que de w é válida e foi instalada no sistema (linha 3). Os servidores aguardam por um quórum destas mensagens para então reconfigurar o sistema (linha 4). Adicionalmente, um servidor $i \in cv$ também envia esta mensagem quando receber mais de $w.ov.f$ mensagens de INSTALL-VIEW, garantindo que w foi gerada em pelo menos um servidor correto. Esta abordagem garante que todos os processos corretos instalam w.

Algoritmo 9 Reconfigurando o sistema com geradores seguros (servidor i)

variables: {Sets and arrays used in the protocol}
RECV - set of received updates (from Algorithm 8)
VIEW_GEN - the view generator (from Algorithm 8)

Upon VIEW_GEN.*new_view*(cv, w)
1: $\forall j \in w \cup w.ov$, *send*$\langle$INSTALL-VIEW$, w\rangle_i$ to j {$w.ov = cv$ (line 1 of Alg. 5 and line 22 of Alg. 6}

Upon receipt of $\langle$INSTALL-VIEW$, w\rangle_*$ from $w.ov.f + 1$ different servers in $w.ov \wedge i \in w.ov$
2: $\forall j \in w \cup w.ov$, *send*$\langle$INSTALL-VIEW$, w\rangle_i$ to j

Upon receipt of $\langle$INSTALL-VIEW$, w\rangle_*$ from $w.ov.q$ different servers in $w.ov \wedge (i \in w \cup w.ov)$
3: w.P $\leftarrow$ {the $w.ov.q$ signed $\langle$INSTALL-VIEW$, w\rangle_*$ messages received}
4: **reconfigure**(w)

procedure *reconfigure*(w)
5: **if** $cv \subset w$ **then**
6: **if** $(i \notin w)$ **then** {i is leaving the system}
7: *disable operations* {stops accepting R/W client requests}
8: **end if**
9: $cv \leftarrow w$
10: RECV $\leftarrow$ RECV $\setminus$ $w.entries$
11: **if** $i \in w.ow$ **then** {i is member of the old view}
12: $\forall j \in w$, *send*$\langle$STATE-UPDATE$, data, ts, w,$ RECV$\rangle$ to j
13: **end if**
14: **if** $i \in w$ **then** {i is member of the updated view}
15: **wait** for $w.ov.q$ STATE-UPDATE from different servers in $w.ov$
16: $\langle data, ts\rangle \leftarrow$ read from the STATE-UPDATE messages received
17: RECV $\leftarrow$ RECV $\cup$ {all valid update requests collected from STATE-UPDATE messages $\setminus w.entries$}
18: **if** $i \notin w.ov$ **then** {i is entering on the system}
19: *enable operations* {starts to accept R/W client requests}
20: **end if**
21: start timer for cv {Notice that at this point $cv = w$ (line 19)}
22: **else**
23: discard keys and halt
24: **end if**
25: **end if**

Procedimento de reconfiguração. Todos os servidores corretos da nova visão w, juntamente com todos os servidores corretos da visão que está sendo atualizada $w.ov$ (neste ponto ainda temos que $w.ov = cv$), executam o procedimento de reconfiguração (linhas 5-25), onde a visão corrente do sistema cv é atualizada de $w.ov$ para w (linha 9). Inicialmente, servidores que estão deixando o sistema desabilitam suas operações (evento de *disable operations*). Então, todos os servidores atualizam suas variáveis cv e seus conjuntos RECV, eliminando todas as requisições de atualização já executadas. Após isso, os servidores em w lêem o valor armazenado em $w.ov$ (mensagens STATE-UPDATE), assegurando que os mesmos passam a armazenar o valor atualizado do registro ($data$) e do *timestamp* (ts). Os servidores em w também atualizam seus conjuntos RECV, de forma a manterem alguma requisição de atualização ainda não computada (linhas 15-17). No fim, os servidores que estão deixando o sistema descartam suas chaves, enquanto que os servidores que entraram no sistema habilitam suas operações (evento *enable operations*).

5.6.2.2.1 Corretude

A seguir apresentamos as provas de corretude do Algoritmo 9, que reconfigura o sistema utilizando tanto um gerador de visões perfeito quanto um gerador de visões seguro.

Lema 20 *Considere uma visão v instalada no sistema, com um gerador de visões perfeito ou seguro associado gv, e o Algoritmo 9. Se um servidor correto $i \in v$ iniciar uma reconfiguração para v, então todos os servidores corretos em $v \cup w$ instalam uma única visão atualizada w.*

Prova. Pelo Lema 17, caso *i* inicie a reconfiguração para *v*, então *gv* é inicializado. Pelo Teorema 3 e pelo Lema 14, *gv* acabará por gerar uma única visão atualizada *w*. Desta forma, através do processamento das mensagens INSTALL-VIEW, todos os servidores em $v \cup w$ executam a operação *reconfigure* (linha 4) e, como $v \subset w$, instalam *w*. $\square_{Lema\ 20}$

5.6.2.3 Reconfiguração com Geradores de Visões Vivos

Esta seção apresenta o protocolo para reconfiguração do sistema através de um gerador de visões com a propriedade de terminação forte, a qual define que alguma visão sempre é gerada, mesmo que seja diferente nos diversos servidores (exatidão fraca). Deste modo, tanto um gerador de visões perfeito $\mathscr{P}$ quanto um gerador de visões vivo $\mathscr{L}$ podem ser empregados neste protocolo. Rigorosamente falando, até mesmo um gerador de visões seguro $\mathscr{S}$ pode ser utilizado neste protocolo. No entanto, não faz muito sentido usar um gerador perfeito ou seguro com este protocolo que é mais custoso e complicado do que aquele apresentado na seção anterior.

Neste protocolo, apesar da possibilidade de várias visões serem geradas pelo gerador, apenas algumas delas são instaladas no sistema, formando uma sequência de visões instaladas. Os clientes apenas poderão executar operações nas visões que formam esta sequência, o que garante as propriedades tanto de terminação quanto de segurança do QUINCUNX.

A principal ideia deste protocolo é fazer com que os servidores organizem as várias visões geradas em sequências. O ponto fundamental para entender o funcionamento deste protocolo é a separação do algoritmo que gera visões atualizadas (encapsulado nos geradores de visões) do algoritmo que organiza estas visões em sequências (apresentado nesta seção). Deste modo, cada visão *v* estabelece uma sequência de visões para sua atualização e encaminha estes dados para que a visão seguinte na sequência *w* seja instalada. Após isto, os servidores de *w* utilizam os dados recebidos de *v* e definem

uma nova sequência de visões para atualização de w, e assim por diante até que existam visões mais atuais para serem instaladas no sistema.

Como não utiliza consenso, mais de uma sequência seq_1 e seq_2 de atualização podem ser definidas por uma visão v. No entanto, através da propriedade de intersecção dos quóruns é possível garantir que uma sequência sempre estará contida em outra definida posteriormente, i.e., $seq_1 \subset seq_2$. Por exemplo, seq_1 pode ser $w_1 \rightarrow w_2$ e então seq_2 é composta por $w_1 \rightarrow w_2 \rightarrow w_3$ (ou $w_0 \rightarrow w_1 \rightarrow w_2 \rightarrow w_3$, ou $w_1 \rightarrow w_2 \rightarrow w_3 \rightarrow w_4$, etc...). Note que todas estas sequências podem ser agrupadas em uma única sequência que engloba todas as visões geradas por determinado gerador, uma vez que as mesmas possuem uma relação que permite esta estruturação (Lema 15).

Dada uma sequência seq definida para atualização de uma visão v, apenas a última visão w de seq é efetivamente instalada no sistema (em nosso exemplo: seq_1 instalará w_2 e seq_2 instalará w_3). As visões restantes apenas são utilizadas no processo de instalação de w, pois como é possível que uma sequência anterior definida para atualização de v tenha instalado uma visão $w' \in seq$, o valor do registrador dos servidores em w deve ser atualizado através de uma leitura "realizada em cadeia" nesta sequência (cada visão lê o valor armazenado na visão imediatamente anterior na sequência). Em nosso exemplo, quando da instalação de w_3 (instalada pela seq_2) uma leitura deve ser realizada em w_2, que lê de w_1, que lê de v. Note que w_2 pode ter sido instalada anteriormente através de seq_1, neste caso qualquer valor mais atual escrito em w_2 será enviado para w_3, não sendo perdido durante a atualização da visão do sistema.

As visões que não são instaladas no sistema, chamadas de *visões auxiliares*, não possuem um gerador de visões associado, pois não é necessário atualizá-las (estas visões já se encontram no meio de alguma sequência de visões, i.e., já existe uma visão mais atual para ser instalada no sistema). Desta forma, apenas visões instaladas possuem geradores de visões para geração de novas visões. Neste sentido, um ponto fundamental para o funcionamento correto deste algoritmo é que cada visão efetivamente instalada no sistema pode mudar uma sequência previamente definida para sua atualização, definindo uma nova sequência que "mistura" visões de sequências previamente definidas com visões geradas por seu gerador de visões associado. Também é possível que novas visões sejam criadas através da união de visões previamente definidas com visões geradas pelo seu gerador.

Com isto, é possível resolver impasses e organizar todas as visões em uma única sequência. Em nosso exemplo, a visão w_2 poderia iniciar uma reconfiguração antes de receber seq_2 (que faria o sistema atualizar para w_3). Neste caso, o gerador associado à w_2 poderia gerar uma visão w_{2_1} e não seria possível definir se w_2 atualizaria para w_{2_1} (por meio de seu gerador)

ou w_3 (por meio de seq_2). Pior ainda, alguns servidores de w_2 poderiam receber primeiro w_{2_1} e tentar atualizar o sistema para esta visão, enquanto que outros servidores poderiam receber primeiro w_3 (seq_2) e tentar instalar esta visão, causando uma divisão de modo que nenhuma das visões pudesse ser instalada. Nossa abordagem resolve este problema fazendo com que w_2 defina uma nova sequência de atualização do sistema, a qual poderá conter tanto a visão w_3 quando a nova visão w_{2_1} ou ainda uma visão representando a união de ambas $w_3 \cup w_{2_1}$, mantendo uma sequência única de visões, como veremos adiante.

Neste protocolo, apresentado nos algoritmos 10 e 11, para validação de uma visão do sistema v (função $verify(v)$ usada pelos processos para verificar a validade das visões) é necessário que:

- v seja válida (ver Algoritmo 10): $v.P$ contém um quórum de assinaturas seq_pr^w geradas por servidores de w, para uma sequência seq^w, e $v \in seq^w$ (sequência contendo v gerada pela visão w). Então, deve-se também verificar se w é válida seguindo estes mesmos critérios.

- v foi instalada (ver Algoritmo 11): $v.P$ contém um quórum de tuplas $\langle \text{INSTALLED}, w, v \rangle_*$ assinadas por servidores de w.

Os algoritmos utilizam as seguintes variáveis: `SEQ` – sequência de visões propostas; `SEQ_PR` – contém as provas de que servidores propuseram `SEQ`; `LCSEQ` – última sequência de visões convergida por algum servidor; e `LCSEQ_PR` – contém as provas de que `LCSEQ` foi proposta por pelo menos um quórum de servidores.

Idealmente, uma sequência para atualização da visão corrente do sistema cv deveria conter uma única visão w de tal modo que o sistema seria atualizado para w. Infelizmente isto não é possível em um sistema assíncrono. No entanto, nosso protocolo tenta restringir ao máximo o número de visões incluidas em uma sequência: primeiramente, para o gerador de visões gerar mais de uma visão é necessário que as trocas de mensagens siga um padrão muito específico (Seção 5.6.1.3); posteriormente, o protocolo apresentado nesta seção para organização destas visões em sequências tenta adicionar o menor número possível de visões em determinada sequência.

Neste sentido, quando o gerador de visões reporta a geração de uma nova visão w para um servidor $i \in cv$, o mesmo verifica se já fez qualquer proposta para sequência de atualização de cv (linha 1), para então propor uma sequência contendo apenas w (linhas 2-3). Esta visão w será a única visão proposta por i, mas diferentes servidores podem propor diferentes visões (propriedade de exatidão fraca do gerador de visões vivo) e as mesmas devem ser organizadas em uma sequência.

Algoritmo 10 Reconfigurando o sistema com geradores vivos (servidor i)

variables: {Sets and arrays used in the protocol}
SEQ, SEQ_PR - proposed sequence of views and its corresponding proofs
LCSEQ, LCSEQ_PR - last converged sequence of views known and its corresponding proofs
VIEW_GEN - the view generator (from Algorithm 8)

Upon VIEW_GEN.*new_view*(cv,w)
1: **if** $\texttt{SEQ}^{cv} = \emptyset$ **then** {$w.ov = cv$ (line 15 of Alg. 7}
2: $\quad\texttt{SEQ}^{cv} \leftarrow \{w\}$
3: $\quad\forall j \in cv, send\langle$SEQ-VIEW$, \texttt{SEQ}_i^{cv}, \emptyset\rangle$ to j
4: **end if**

Upon receipt of $\langle$SEQ-VIEW$, seq_j^{cv}, \langle lcseq, proof\rangle\rangle$ from j {i only executes this message if it remains in cv }
5: **if** $isValid(seq_j^{cv}, \langle lcseq, proof\rangle) \wedge (i,j \in cv)$ **then**
6: $\quad$**if** $\exists v \in seq_j^{cv} : v \notin \texttt{SEQ}^{cv}$ **then**
7: $\quad\quad$**if** $\exists v,w : v \in seq_j^{cv} \wedge w \in \texttt{SEQ}^{cv} \wedge v \not\subset w \wedge w \not\subset v$ **then** {v and w was generated by different reconfig.}
8: $\quad\quad\quad v \leftarrow w : (w \in \texttt{LCSEQ}^{cv}) \wedge (\nexists w' \in \texttt{LCSEQ}^{cv} : w \subset w')$ {the most updated view in $\texttt{LCSEQ}^{cv}$}
9: $\quad\quad\quad v' \leftarrow w : (w \in lcseq) \wedge (\nexists w' \in lcseq : w \subset w')$ {the most updated view in $lcseq$}
10: $\quad\quad\quad$**if** $v \subset v'$ **then** {$lcseq$ is up to date than $\texttt{LCSEQ}^{cv}$}
11: $\quad\quad\quad\quad\langle \texttt{LCSEQ}^{cv}, \texttt{LCSEQ\_PR}^{cv}\rangle \leftarrow \langle lcseq, proof\rangle$
12: $\quad\quad\quad$**end if**
13: $\quad\quad\quad v \leftarrow w : (w \in \texttt{SEQ}^{cv}) \wedge (\nexists w' \in \texttt{SEQ}^{cv} : w \subset w')$ {the most updated view in $\texttt{SEQ}^{cv}$}
14: $\quad\quad\quad v' \leftarrow w : (w \in seq_j^{cv}) \wedge (\nexists w' \in seq_j^{cv} : w \subset w')$ {the most updated view in seq_j^{cv}}
15: $\quad\quad\quad \texttt{SEQ}^{cv} \leftarrow \texttt{LCSEQ}^{cv} \cup \{\langle cv, v.entries \cup v'.entries, \langle v, v'\rangle\rangle\}$
16: $\quad\quad$**else**
17: $\quad\quad\quad \texttt{SEQ}^{cv} \leftarrow \texttt{SEQ}^{cv} \cup seq^{cv}$
18: $\quad\quad$**end if**
19: $\quad\quad \texttt{SEQ\_PR}^{cv} \leftarrow \emptyset$
20: $\quad\quad \forall k \in cv, send\langle$SEQ-VIEW$, \texttt{SEQ}_i^{cv}, \langle \texttt{LCSEQ}^{cv}, \texttt{LCSEQ_PR}^{cv}\rangle\rangle$ to k
21: $\quad$**end if**
22: $\quad$**if** $\forall v \in seq_j^{cv} \leftrightarrow v \in \texttt{SEQ}^{cv}$ **then**
23: $\quad\quad \texttt{SEQ\_PR}^{cv} \leftarrow \texttt{SEQ\_PR}^{cv} \cup seq_j^{cv}$
24: $\quad\quad$**if** $(|\texttt{SEQ\_PR}^{cv}| \geq cv.q)$ **then**
25: $\quad\quad\quad \langle \texttt{LCSEQ}^{cv}, \texttt{LCSEQ\_PR}^{cv}\rangle \leftarrow \langle \texttt{SEQ}^{cv}, \texttt{SEQ\_PR}^{cv}\rangle$
26: $\quad\quad\quad v \leftarrow w : (w \in \texttt{SEQ}^{cv}) \wedge (\nexists w' \in \texttt{SEQ}^{cv} : w' \subset w)$ {the next view in the sequence $\texttt{SEQ}^{cv}$}
27: $\quad\quad\quad \forall k \in cv \cup v, send\langle$INSTALL-SEQ$, v, \texttt{SEQ}^{cv}, \texttt{SEQ_PR}^{cv}\rangle$ to k
28: $\quad\quad$**end if**
29: $\quad$**end if**
30: **end if**
Upon receipt of $\langle$INSTALL-SEQ$, v, seq^{ov}, seq_pr^{ov}\rangle$ from $ov.q$ different servers in ov and $i \in \{ov \cup v\}$
31: **reconfigure**$(v, seq^{ov}, seq\_pr^{ov})$

Function $isValid(seq_j^{cv}, \langle lcseq, proof\rangle)$ verifies if:

a) $\forall w \in seq_j^{cv}$ we have that $cv \subset w$ and all views in seq_j^{cv} form a sequence according with the $\subset$ operator;

b) $\forall w \in seq_j^{cv}$ we have one of the following:

1 $w.P$ is a set of $\langle w.ov, w.entries, \emptyset\rangle_*$ tuples and: $\#_{\langle w.ov, w.entries, \emptyset\rangle_*} w.P \geq w.ov.q$

2 $w.P$ is a tuple $\langle v, v'\rangle$ and: *(i)* v and v' satisfy the previous condition (case 1) and *(ii)* $((w.entries = v.entries \cup v'.entries) \wedge (v.ov \neq v'.ov))$.

c) $\#_{lcseq_*} proof \geq cv.q$.

A forma como os servidores convergem para sequências é muito se-

melhante com a forma de como os mesmos convergem para visões no gerador de visões vivo (Seção 5.6.1.3). Deste modo, quando um servidor $i \in cv$ receber uma proposta para sequência de visões de outro servidor j, o mesmo verifica se todas as visões propostas por j são válidas, i.e., foram geradas pelo gerador de visões (linha 15 do algoritmo 7) ou durante a organização de outras visões válidas (linha 15 do algoritmo 10).

Algoritmo 11 Reconfigurando o sistema com geradores vivos (servidor i), continuação...

variables: {Sets and arrays used in the protocol}
RECV - set of received updates (from Algorithm 8)
SEQ - proposed sequence of views (from Algorithm 10)

procedure *reconfigure*$(v, seq^{ov}, seq\_pr^{ov})$
1: $v.P \leftarrow seq\_pr^{ov}$ {proofs that v is in the sequence}
2: **if** $i \notin v$ **then** {i is leaving the system}
3: *disable operations* {stops accepting R/W client requests}
4: **end if**
5: **if** $cv \subset v$ **then** {i updates its current view to v }
6: $cv \leftarrow v$
7: **end if**
8: RECV $\leftarrow$ RECV $\setminus\, v.entries$
9: **if** $i \in ov$ **then** {i is member of the previous view in the sequence}
10: **if** $|seq^{ov} \setminus v = \emptyset|$ **then** {v is the last view in the sequence}
11: $\forall j \in v, send\langle$STATE-UPDATE$, data, ts, \langle$INSTALLED$, ov, v\rangle_i,$ RECV$\rangle$ to j
12: **else**
13: $\forall j \in v, send\langle$STATE-UPDATE$, data, ts, \perp,$ RECV$\rangle$ to j
14: **end if**
15: **end if**
16: **if** $i \in v$ **then** {i is member of the updated view}
17: **wait** for $ov.q$ $\langle$STATE-UPDATE$, *, *, *, *\rangle$ for different servers in ov
18: **if** $(|\{seq^{ov} \setminus v\}| = \emptyset) \wedge cv = v$ **then** {v is the last view in the sequence}
19: $cv.P \leftarrow cv.P \cup$ the $ov.q$ $\langle$INSTALLED$, ov, v\rangle_*$ received {proofs that v was installed}
20: **end if**
21: $\langle data, ts\rangle \leftarrow$ read from the quorum of STATE-UPDATE messages received
22: RECV $\leftarrow$ RECV $\cup$ {all valid updates requests collected from STATE-UPDATE messages} $\setminus\, cv.entries$
23: **if** $i \notin ov$ **then** {i is entering on the system}
24: *enable operations* {starts to accept R/W client requests}
25: **end if**
26: $\forall j \in ov$: $j \notin v, send\langle$VIEW-INSTALLED$, v\rangle$ to j
27: **if** $(\mathtt{SEQ}^v = \emptyset) \wedge (\{seq^{ov} \setminus v\} \neq \emptyset)$ **then**
28: $\mathtt{SEQ}^v \leftarrow \{seq^{ov} \setminus v\}$
29: $\forall k \in v, send\langle$SEQ-VIEW$, \mathtt{SEQ}_i^v, \emptyset\rangle$ to k
30: **else if** $\mathtt{SEQ}^v = \emptyset$ **then** {v is the last view in the sequence seq^{ov}}
31: start timer for cv {if updated on line 6}
32: **end if**
33: **else** {i is leaving the system}
34: **wait** for $v.f + 1$ $\langle$VIEW-INSTALLED$, v\rangle$ from different servers in v
35: discard keys and halt
36: **end if**

Então, dois casos (não mutualmente exclusivos) são possíveis:

1. i ainda não fez proposta ou a mesma é diferente da proposta de j, onde alguma visão proposta por j ainda não foi proposta por i (linhas 6-21): Então, dois cenários são possíveis:

- As visões já propostas por i e as novas visões recebidas não formam uma sequência (linhas 8-15): isso pode acontecer quando visões geradas por diferentes geradores de visões devem ser organizadas em uma sequência. Então, i define a mais atual sequência de visões já convergida (linhas 8-12) e determina sua proposta de sequência adicionando uma visão que representa a união das duas sequências (a já proposta por i e a recebida de j - linhas 13-15). Isto resolve o conflito devido à necessidade de organizar sequência de visões geradas por diferentes geradores.
- As visões já propostas por i (caso existam) e as novas visões recebidas formam uma sequência (linha 17): então, a nova sequência a ser proposta por i é formada por todas estas visões conhecidas.

2. i fez a mesma proposta de j (linhas 22-29): neste caso i coleta a proposta assinada. Quando i obtém a prova de que um quórum fez a mesma proposta, i converge para uma nova sequência de visões e envia uma mensagem para que a visão seguinte na sequência v seja inicializada.

Os servidores aguardam por um quórum de INSTALL-SEQ para então "instalar" (lembrando que v apenas será instalada caso não exista uma visão seguinte em seq) a visão seguinte v de uma sequência seq (linha 31). Pela propriedade de intersecção dos quóruns, qualquer sequência posteriormente obtida seq' conterá seq, de modo que qualquer visão instalada através de seq estará contida em seq' e será acessada na atualização do estado da visão instalada através de seq'.

Para instalar v, o sistema é reconfigurado de forma semelhante ao protocolo da seção anterior. No entanto, algumas diferenças são apresentadas:

- As provas de que uma visão é válida (linha 1), bem como de que a mesma foi instalda (linhas 10-14,18-20), são coletadas.
- Caso exista alguma visão seguinte em seq, uma sequência para atualização de v é proposta (linhas 27-29).
- Um servidor deixa o sistema somente após garantir que a visão mais atual, para a qual enviou o estado, foi instalada (ou teve seu estado atualizado caso seja uma visão auxiliar – mensagens VIEW-INSTALLED). Por exemplo, caso o algoritmo para atualização de cv produza $seq_1 : v_1 \rightarrow v_2$ e depois $seq_2 : v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow v_3$, alguns servidores de cv podem receber primeiro seq_1 e outros seq_2 de forma que alguns enviam as mensagens de atualização de estado antes para v_1 e outros para v_0. Caso saíssem do sistema após o primeiro envio desta mensagem, seria

possível que nenhuma das visões v_0 ou v_1 recebesse um quórum destas mensagens, de modo que o sistema perderia vivacidade.

Exemplo de funcionamento do protocolo. Imagine que na reconfiguração de uma visão v_0, o gerador de visões associado a esta visão gera as visões atualizadas v_1, v_2 e v_3, sendo que $v_1 \subset v_2 \subset v_3$ (Lema 15). Após isso, os servidores em v_0 geram as sequências $seq_1^{v_0} : v_1 \rightarrow v_2$ e $seq_2^{v_0} : v_1 \rightarrow v_2 \rightarrow v_3$. Então, após v_1 ser inicializada (estado atualizado), a mesma pode gerar as sequências $seq_1^{v_1} : v_2$ e $seq_2^{v_1} : v_2 \rightarrow v_3$. Neste exemplo, v_1 é uma visão auxiliar enquanto que v_2 e v_3 poderão ser instaladas. Os servidores de v_2 que receberem $seq_2^{v_1}$ antes de $seq_1^{v_1}$ também não instalam v_2.

Por outro lado, caso os servidores em v_2 receberem primeiro $seq_1^{v_1}$, então v_2 é instalada no sistema. Como estamos em um sistema assíncrono, é possível que os servidores de v_2 iniciem o gerador de visões *gv* para atualização de v_2 e só então recebam $seq_2^{v_1}$ para atualização de v_2 através da instalação de v_3. Neste caso, é possível que concorrentemente *gv* gere uma nova visão w_1 para atualização de v_2.

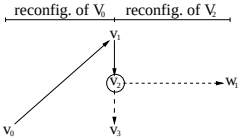
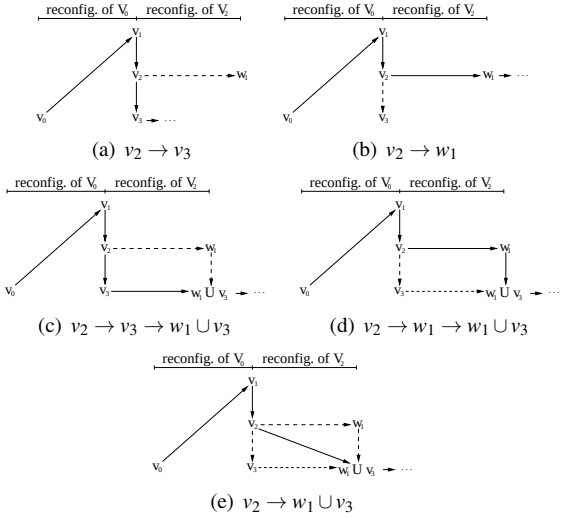


Figura 12 – Servidores de v_2 em conflito.

Neste cenário, os servidores de v_2 podem "entrar em conflito" onde alguns destes servidores recebem primeiro as mensagens para gerar a sequência com w_1 enquanto outros recebem primeiramente as mensagens para gerar a sequência com v_3, como mostra a Figura 12. Neste caso, tanto v_3 quanto w_1 devem ser organizadas em uma sequência a ser instalada a partir de v_2. Para isso, é possível que uma nova visão representando a união destas sequências $(w_1 \cup v_3)$ seja gerada.

No processamento das mensagens SEQ-VIEW, dependendo de quais mensagens compondo um quorum são entregues nos servidores, é possível que seja instalada apenas uma das duas visões (v_3 ou w_1), ou uma destas visões seguida pela união das mesmas $(w_1 \cup v_3)$, ou ainda apenas a união de ambas (Figura 13). Porém, como é necessário um quórum de mensagens para definir uma nova sequência, nunca teremos a instalação de ambas, w_1 e v_3, fazendo com que a sequência de visões instaladas não seja quebrada.



(a) $v_2 \rightarrow v_3$ (b) $v_2 \rightarrow w_1$

(c) $v_2 \rightarrow v_3 \rightarrow w_1 \cup v_3$ (d) $v_2 \rightarrow w_1 \rightarrow w_1 \cup v_3$

(e) $v_2 \rightarrow w_1 \cup v_3$

Figura 13 – Possíveis sequências de visões instaladas.

5.6.2.3.1 Corretude

A seguir apresentamos as provas de corretude dos algoritmos 10 e 11, que reconfiguram o sistema utilizando um gerador de visões vivo.

Lema 21 *Considere uma visão v instalada no sistema, com um gerador de visões vivo associado gv, e os algoritmos 10 e 11. Então, qualquer visão w, definida para atualização de v e instalada no sistema, estará contida em qualquer outra sequência de visões definida para atualização de v.*

Prova. Considere que uma visão w é instalada no sistema. Então, os servidores de alguma visão desatualizada v definiram uma sequência *seq* para sua atualização onde a visão mais atual é w (linha 1, 10, 18-20 do algoritmo 11). Além disso, nenhum servidor possui armazenado em `LCSEQ` uma sequência contendo uma visão mais atual do que w, caso contrário não convergiriam para *seq* uma vez que é necessário um quórum de mensagens SEQ-VIEW para convergir para alguma sequência e, então, pelas propriedades de intersecção dos quóruns, seria impossível convergirem para *seq*. Desta forma, *seq* não é substituída por qualquer outra sequência (linhas 8-12 do algoritmo 10). Agora, considere que uma outra sequência seq' é definida para atualização de v. Pela propriedade de intersecção dos quóruns e pelo fato de ser necessário

um quórum de servidores para gerar qualquer sequência (linha 24 do algoritmo 10), é garantido que $seq \subset seq'$ e, então, $w \in seq'$. $\square_{Lema\ 21}$

Lema 22 *Considere uma visão v instalada no sistema, com um gerador de visões vivo associado gv, e os algoritmos 10 e 11. Então, quando uma visão atualizada w é instalada no sistema, qualquer visão w' menos atual do que w ($w' \subset w$) não está mais ativa e nunca voltará a ser ativada.*

Prova. Considere que uma visão w é instalada no sistema. Então, os servidores de alguma visão desatualizada v definiram uma sequência seq para sua atualização onde a visão mais atual é w. Neste caso, para w ser instalada, todas as visões em seq definiram sequências para suas atualizações até chegar em uma visão w_{aux}, imediatamente anterior a w em seq, que definiu uma sequência com apenas w, de forma que w é instalada (linha 1, 10, 18-20 do algoritmo 11). Neste caso, todas as outras visões em seq foram desativadas (linhas 2-7,33-36 do algoritmo 11). Agora, considere que uma sequência anterior seq' tenha instalado uma outra visão w'. Pelo Lema 21, temos que $seq' \subset seq$ e, então, $w' \in seq$. Como $w' \in seq$, w' é desativada na instalação de w. Por indução, qualquer visão w' menos atual do que w está desativada quando w é instalada. Além disso, pelo Lema 21, qualquer sequência seguinte seq'' será tal que $seq \subset seq''$, então w' jamais será ativada novamente, pois apenas será uma visão auxiliar em seq''. $\square_{Lema\ 22}$

Lema 23 *Considere uma visão v instalada no sistema, com um gerador de visões vivo associado gv, e os algoritmos 10 e 11. Então, quando uma visão atualizada w é instalada, qualquer visão w' mais atual do que w ($w \subset w'$) ainda não foi instalada no sistema.*

Prova. Considere que uma visão w é instalada no sistema. Então, neste momento w está ativa. Agora, considere que uma visão mais atual w' ($w \subset w'$) é instalada no sistema. Neste caso, pelo Lema 22, w não está ativa, o que é uma contradição. Logo, w' ainda não foi instalada no sistema. $\square_{Lema\ 23}$

Lema 24 *Considere uma visão v instalada no sistema, com uma gerador de visões vivo associado gv, e os algoritmos 10 e 11. Então, visões auxiliares não modificam uma sequência previamente definida para atualização de v.*

Prova. Uma visão v_{aux} é auxiliar quando não é a visão mais atual de uma sequência seq definida para atualização de uma visão v. Neste caso, não são geradas as provas de que v_{aux} foi instalada (linhas 10-14), pois a mesma não é instalada. Então, uma sequência com as visões seguintes em seq é proposta para atualização de v_{aux} (linhas 27-29 do algoritmo 11). Como v_{aux} contém pelo menos um quórum de servidores corretos e é necessário um quórum de

servidores para a obtenção de uma sequência seq' para atualização de v_{aux}, nenhuma visão seguinte em seq é suprimida em seq' e assim $seq' = seq \setminus v_{aux}$. Além disso, o *timer* associado a v_{aux} não é ativado (linhas 27-32 do algoritmo 11). Desta forma, pelo Lema 18, uma reconfiguração para v_{aux} nunca é iniciada. Com isso, seu gerador de visões associado também nunca é iniciado e novas visões não são geradas para propor na atualização de v_{aux}. Desta forma, $seq' = seq \setminus v_{aux}$ e a sequência previamente definida seq não é alterada. $\square_{Lema\ 24}$

Lema 25 *Considere uma visão v instalada no sistema, com um gerador de visões vivo associado gv, e os algoritmos 10 e 11. Então, as visões instaladas a partir de v formam uma sequência única.*

Prova. Pelo Lema 24, apenas visões que são instaladas no sistema podem modificar uma sequência previamente definida para atualização do mesmo, a partir de uma visão desatualizada v. Pelo Lema 22, quando uma visão w é instalada todas as visões menos atuais do que w já foram desativadas. Além disso, pelo Lema 23, nenhuma visão mais atual do que w já foi instalada no sistema. Desta forma, w pode definir qualquer nova sequência para sua atualização, pois é garantido que nenhuma visão instalada ficará de fora desta sequência, visto que nenhuma visão mais atual foi instalada. Assim, as visões instaladas no sistema a partir de v formam uma sequência única. $\square_{Lema\ 25}$

5.6.2.4 Provas das Propriedades do Sistema

Esta seção apresenta a provas de que os protocolos do QUINCUNX satisfazem as propriedades do sistema definidas na Seção 5.3. Primeiramente, as propriedades de terminação são analisadas e, posteriormente, as propriedades de segurança também são provadas. Todas as definições da Seção 5.3 são consideradas. Em especial, $V(t)$ é a visão mais atual que algum servidor correto $i \in V(t)$ instalou desde o início da execução (t_0) até o tempo t.

5.6.2.4.1 Propriedades de Terminação

Teorema 6 ***(Terminação 1)*** *Todas operações de leitura e escrita executadas por clientes corretos terminam.*

Prova. Considere uma operação de leitura ou escrita o executada por um cliente correto c, iniciando no tempo t. Neste caso, para cada fase ph de o, c envia uma mensagem para todos os servidores em sua visão corrente cv_c

e aguarda por um quórum de respostas. Caso cv_c seja a visão mais atual ($cv_c = V(t)$) e como cv_c contêm pelo menos um quórum de servidores corretos (Suposição 5), c termina ph em cv_c e inicia a execução da fase seguinte de o, e assim por diante até terminar a última fase de o em cv_c, terminando a execução de o. Por outro lado, caso cv_c esteja desatualizada ($cv_c \subset V(t)$), os servidores enviam $V(t)$ para c, que reinicia ph em $V(t)$, terminando o em $V(t)$. No entanto, caso exista um tempo $t' > t$ tal que c não terminou ph até t' e $V(t') \neq V(t)$, então c recebe $V(t')$ e reinicia novamente ph em $V(t')$ e assim por diante. Pela Suposição 4, o número de *updates* requisitados concorrentemente com o é finito, então existe um tempo $t'' \geq t$ tal que o sistema não executa mais reconfigurações e c termina o em $V(t'')$. $\square_{Teorema\ 6}$

Lema 26 *Seja um servidor correto i que executa uma operação de join ou leave no tempo t, enviando o update u para os servidores em $V(t)$ enquanto o sistema não está reconfigurando. Então, para qualquer visão $V(t')$ tal que $V(t') \neq V(t) \wedge t' > t$, temos que $u \in V(t')$.*

Prova. Considere um servidor correto i que envia u para os servidores em $V(t)$ no tempo t. Caso o sistema não esteja reconfigurando, pelo menos um quórum de servidores de $V(t)$ armazenaram u (linha 5 do algoritmo 8). Agora, considere um tempo $t' > t$ onde uma reconfiguração gerou uma nova visão $V(t')$ (Lemas 14 e 19; teoremas 3 e 5). Neste caso, o gerador de visões gv associado a $V(t)$ foi iniciado em pelo menos $V(t).q - V(t).f$ servidores com *updates* up tal que $u \in up$ (linha 10 do algoritmo 8). Como foi necessário um quórum de propostas para gv gerar $V(t')$ (linhas 16 do algoritmo 6 e 14 do algoritmo 5.6.1.3) e pela propriedade de intersecção dos quóruns, é garantido que $u \in V(t')$. $\square_{Lema\ 26}$

Lema 27 *Seja um servidor correto i que executa uma operação de join ou leave no tempo t, enviando o update u para os servidores em $V(t)$ enquanto o sistema está reconfigurando. Então, existirá uma visão $V(t')$ tal que $V(t') \neq V(t) \wedge t' > t$, onde $u \in V(t')$.*

Prova. Considere um servidor correto i que envia u para os servidores em $V(t)$ no tempo t. Caso o sistema esteja reconfigurando, temos dois casos:

- i envia u para um quórum de servidores de $V(t)$ antes que os mesmos iniciem o gerador de visões gv associado à $V(t)$: caso do Lema 26;
- i não envia u para um quórum de servidores de $V(t)$ antes que os mesmos iniciem o gerador de visões gv associado à $V(t)$: neste caso, não é garantido que pelo menos $V(t).q - V(t).f$ servidores inicializem gv com *updates* up, tal que $u \in up$ (linha 10 do algoritmo 8). Então, no

pior caso, gv gera uma única visão w (Lemas 14 e 19; teoremas 3 e 5) tal que o quórum de propostas usadas na sua geração não contenha u. Mesmo assim, no final da reconfiguração os servidores de $V(t)$ enviam u nas mensagens STATE-UPDATE para os servidores em w, os quais iniciarão seu gerador de visões gv' associado com *updates* up', tal que $u \in up'$. Assim, para qualquer visão v gerada por gv' (Lemas 14 e 19; teoremas 3 e 5), $u \in v$. Então, existirá uma visão $V(t')$ tal que $V(t') \neq V(t) \wedge t' > t$, onde $u \in V(t')$.

Deste modo, em qualquer dos casos, existirá uma visão $V(t')$ tal que $V(t') \neq V(t) \wedge t' > t$, onde $u \in V(t')$. $\square_{Lema\ 27}$

Teorema 7 ***(Terminação 2)*** *O evento enable operations terminará por ocorrer em todo servidor correto que executa a operação de join.*

Prova. Considere que um servidor correto i executa a operação de *join* no tempo t, enviando o *update* $u = \langle +, i \rangle_i$ para os servidores de $V(t)$. Pelos Lemas 26 e 27, existirá uma visão $V(t')$ tal que $V(t') \neq V(t) \wedge t' > t$, onde $u \in V(t')$. Então, quando $V(t')$ é instalada em i (Lemas 20 e 25), o evento de *enable operations* ocorre neste servidor, pois as condições são verificadas para servidores que estão entrando no sistema (linhas 19 do algoritmo 9 ou 24 do algoritmo 11). $\square_{Teorema\ 7}$

Teorema 8 ***(Terminação 3)*** *O evento disable operations terminará por ocorrer em todo servidor correto que executa a operação de leave.*

Prova. Considere que um servidor correto i executa a operação de *leave* no tempo t, enviando o *update* $u = \langle -, i \rangle_i$ para os servidores de $V(t)$. Pelos Lemas 26 e 27, existirá uma visão $V(t')$ tal que $V(t') \neq V(t) \wedge t' > t$, onde $u \in V(t')$. Então, quando $V(t)$ é atualizada para $V(t')$ em i (Lemas 20 e 25), o evento de *disable operations* ocorre neste servidor, pois as condições são verificadas para servidores que estão saindo do sistema (linhas 7 do algoritmo 9 ou 3 do algoritmo 11). $\square_{Teorema\ 8}$

5.6.2.4.2 Propriedades de Segurança

Antes de provar que o QUINCUNX satisfaz a propriedade de segurança de um registrador atômico (LAMPORT, 1986), a definição desta propriedade é apresentada.

Definição 12 (Propriedade de Registrador Atômico) *Considere duas operações de leitura l_1 e l_2 executadas por clientes corretos, onde l_1 completa*

antes de l_2 iniciar. Caso l_1 leia um valor val do registro, então l_2 também lê val ou um valor mais atual escrito neste registrador.

Lema 28 *Seja val o valor relacionado com a última escrita completa em $V(t)$, terminada no tempo t. Então, na reconfiguração r para $V(t)$ que instala a visão seguinte $V(t')$, onde $t' > t$, temos dois casos:*

1. *Sem concorrência com r: neste caso, val é propagado para $V(t')$;*
2. *r concorrente com uma operação de escrita op_e: neste caso, val ou val_e (valor relacionado com op_e) é propagado para $V(t')$;*

Prova. Considere que *val* é o valor relacionado com a última escrita completa em $V(t)$ (com *timestamp* ts) e que a reconfiguração *r* instala uma visão sequinte $V(t')$ (Lemas 20 e 25). Então, vamos provar cada um dos casos:

1. Sem concorrência com *r*: neste caso, pelo menos $V(t).q$ servidores de $V(t)$ armazenam *val* e ts, que é o maior *timestamp* do sistema, pois está relacionado com a última escrita realizada no mesmo. Então, através das mensagens STATE-UPDATE e da propriedade de intersecção dos quóruns, os servidores de $V(t')$ lêem *val* (e ts) de $V(t)$.
2. *r* concorrente com uma operação de escrita op_e: neste caso, considere que op_e possui um *timestamp* associado ts_e. Então, temos duas possibilidades:
   - Caso $ts > ts_e$: este é o mesmo caso do item anterior;
   - Caso $ts < ts_e$: neste caso, é possível que alguns servidores de $V(t)$ estejam armazenando val_e e ts_e. Então, é possível que alguns servidores de $V(t')$ leiam *val* enquanto outros leiam val_e. De qualquer forma, op_e será reiniciada em $V(t')$ (Teorema 6) e todos os servidores de $V(t')$ acabarão por armazenar val_e e ts_e.

Como sempre temos uma sequência de visões instaladas (Lemas 20 e 25), estes casos também valem na atualização de $V(t')$ quando a visão seguinte $V(t'')$ (sendo, $t'' > t'$) é instalada, e assim por diante. $\square_{Lema\ 28}$

Lema 29 *Seja um cliente correto c executando uma operação de escrita op_e no tempo t, com valor e timestamp associados val_e e ts_e, respectivamente. Então, temos quatro casos mutualmente exclusivos:*

1. *Sem concorrência: neste caso, val_e é escrito em $V(t)$.*
2. *Concorrente com outra operação de escrita $op_{e'}$: neste caso, val_e ou $val_{e'}$ (valor relacionado com $op_{e'}$) é escrito em $V(t)$.*

3. *Concorrente com uma reconfiguração r que instala a visão seguinte $V(t')$, sendo $t' > t$: neste caso, temos duas situações: (1) op_e termina antes do tempo t', então val_e é escrito em $V(t)$ e propagado para $V(t')$; ou (2) op_e termina depois do tempo t', e val_e é escrito diretamente em $V(t')$.*

4. *Concorrente com uma reconfiguração r que instala a visão seguinte $V(t')$, sendo $t' > t$, e com outra operação de escrita $op_{e'}$: neste caso, temos quatro situações: (1) op_e e $op_{e'}$ terminam antes do tempo t', então val_e ou $val_{e'}$ (valor relacionado com $op_{e'}$) é escrito em $V(t)$ e propagado para $V(t')$; (2) op_e termina antes e $op_{e'}$ depois do tempo t', então val_e é escrito em $V(t)$ e propagado para $V(t')$ e/ou $val_{e'}$ é escrito diretamente em $V(t')$; (3) op_e termina depois e $op_{e'}$ antes do tempo t', então $val_{e'}$ é escrito em $V(t)$ e propagado para $V(t')$ e/ou val_e é escrito diretamente em $V(t')$; ou (4) op_e e $op_{e'}$ terminam depois do tempo t', então val_e ou $val_{e'}$ é escrito diretamente em $V(t')$.*

Prova. Considere um cliente correto c executando uma operação de escrita op_e no tempo t, com valor e *timestamp* associados val_e e ts_e, respectivamente. Então, vamos provar cada um dos casos:

- Casos 1 e 2: nos casos sem concorrência de reconfigurações o comportamento é o mesmo do protocolo de escrita base, neste caso o PHALANX. Então, no caso 1 ts_e é o maior *timestamp* do sistema e val_e é escrito em $V(t)$. No caso 2, o valor relacionado com o maior *timestamp* entre val_e ou $val_{e'}$ é escrito em $V(t)$.

- Caso 3: (1) se op_e termina antes de t', então temos o mesmo do caso 1. No entanto, pelo Lema 28, val_e é propagado para $V(t')$. (2) Já se op_e termina depois de t', então c passa a conhecer $V(t')$ antes de terminar op_e e completa esta operação em $V(t')$ (Teorema 6), escrevendo val_e em $V(t')$.

- Caso 4: (1) se op_e e $op_{e'}$ terminam antes de t', então temos o mesmo do Caso 2 e o valor associado com o maior *timestamp* é escrito em $V(t)$ e propagado para $V(t')$ (Lema 28). (2) Se op_e termina antes e $op_{e'}$ depois do tempo t', então val_e é escrito em $V(t)$ e propagado para $V(t')$ (Lema 28) e caso o *timestamp* de $op_{e'}$ é maior do que ts_e, $val_{e'}$ é escrito em $V(t')$. (3) Se op_e termina depois e $op_{e'}$ antes do tempo t', este caso é o inverso do anterior. (4) op_e e $op_{e'}$ terminam depois do tempo t', então como no Caso 3, op_e e $op_{e'}$ são completadas em $V(t')$ (Teorema 6) e o valor associado ao maior *timestamp* val_e ou $val_{e'}$ é escrito em $V(t')$.

$\square_{Lema\ 29}$

Lema 30 *Considere que val é o valor lido do sistema no tempo t por um cliente correto c, através de uma operação de leitura op_l. Então, temos cinco casos mutualmente exclusivos:*

1. *Sem concorrência: val é o último valor escrito em $V(t)$.*
2. *Concorrente com uma operação de escrita op_e, com o maior timestamp do sistema associado ts_e: val é o último valor escrito em $V(t)$ ou val_e (valor associado à op_e).*
3. *Concorrente com uma reconfiguração r que instala a visão seguinte $V(t')$, sendo $t' > t$: neste caso, temos duas situações: (1) op_l termina antes do tempo t', então val é o último valor escrito em $V(t)$; ou (2) op_l termina depois do tempo t', então val é o valor lido de $V(t')$, sendo o valor propagado de $V(t)$ para $V(t')$.*
4. *Concorrente com uma reconfiguração r que instala a visão seguinte $V(t')$, sendo $t' > t$, e com uma operação de escrita op_e com o maior timestamp do sistema associado ts_e: neste caso, temos duas situações: (1) op_l termina antes do tempo t', então val é o último valor escrito em $V(t)$ ou val_e; (2) op_l termina depois do tempo t', então val é o valor lido de $V(t')$, sendo o valor propagado de $V(t)$ para $V(t')$ ou val_e.*
5. *$val = \perp$: nenhuma operação de escrita foi completada no sistema.*

Prova. Considere que um cliente correto c lê o valor *val* do sistema no tempo t, através de uma operação de leitura op_l. Então, vamos provar cada caso:

- Casos 1, 2 e 5: nos casos sem concorrência de reconfigurações o comportamento é o mesmo do protocolo de leitura base, neste caso o PHALANX. Então, no caso 1 *val* é o último valor escrito em $V(t)$. No caso 2, *val* também pode ser o valor que está sendo escrito val_e, pois ts_e é o maior *timestamp* do sistema. Se nenhuma operação de escrita foi completada no sistema (caso 5), é possível que $val = \perp$, pois não é garantido que $V(t).q$ servidores armazenam algum valor.
- Caso 3: (1) se op_l termina antes de t', então temos o mesmo do Caso 1. (2) Já se op_l termina depois de t', então c passa a conhecer $V(t')$ antes de terminar op_l e completa esta operação em $V(t')$ (Teorema 6), sendo que *val* é o valor propagado de $V(t)$ para $V(t')$ (Lema 28).
- Caso 4: (1) se op_l termina antes de t', então temos o mesmo do Caso 2. (2) Se op_l termina depois do tempo t', então *val* é lido de $V(t')$ (como no Caso 3), sendo o valor propagado de $V(t)$ para $V(t')$ (Lema 28) ou

o valor que está sendo escrito val_e, pois ts_e é o maior *timestamp* do sistema.

$\square_{Lema\ 30}$

Teorema 9 ***(Segurança 1)*** *Os protocolos de leitura e escrita satisfazem as propriedades de segurança de um registrador de leitura e escrita atômico.*

Prova. Este teorema é provado a partir da prova da propriedade de registrador atômico. Considere duas operações de leitura l_1 e l_2 executadas por clientes corretos, onde l_1 completa antes de l_2 iniciar. Considere também que l_1 lê um valor *val*, então devemos provar que l_2 também lê *val* ou um valor mais atual. Considerando que l_1 inicia no tempo t, temos quatro cenários mutualmente excludentes, de acordo com a concorrência com uma reconfiguração r que instala a visão $V(t')$, sendo $t' > t$:

1. Sem concorrência: sem concorrência de reconfigurações o comportamento é o mesmo do protocolo base, neste caso o PHALANX. Assim, a fase de *write-back wb* de l_1 garante que pelo menos $V(t).q$ servidores de $V(t)$ armazenam *val* e seu *timestamp ts* (Lema 29 - Caso 1). Qualquer operação de escrita concorrente (Lema 29 - Caso 2) ou posterior à *wb* somente consegue escrever um valor associado com um *timestamp* maior do que ts. Então, pela propriedade de intersecção dos quóruns, pelo Lema 30 (Casos 1 e 2) e como l_2 inicia após *wb*, é garantido que l_2 lê *val* ou um valor mais atual escrito em $V(t)$.

2. r concorrente com l_1: pelo Lema 29 (Caso 3), a fase de *write-back wb* de l_1 escreve *val* (e seu *timestamp ts*) em $V(t)$ e este valor é propagado para $V(t')$ ou escreve *val* diretamente em $V(t')$. Qualquer operação de escrita concorrente (Lema 29 - Caso 4) ou posterior à *wb* somente consegue escrever em $V(t')$ um valor associado com um *timestamp* maior do que ts. Como r é concorrente com l_1 mas não com l_2, l_2 é executada em uma visão seguinte $V(t')$. Então, pela propriedade de intersecção dos quóruns e como l_2 inicia após *wb*, é garantido que l_2 lê *val* ou um valor mais atual escrito em $V(t')$.

3. r concorrente com l_2: neste caso, a fase de *write-back wb* de l_1 garante que pelo menos $V(t).q$ servidores de $V(t)$ armazenam *val* e seu *timestamp ts* (Lema 29 - Caso 1). Qualquer operação de escrita concorrente (Lema 29 - Caso 2) ou posterior à *wb* somente consegue escrever um valor associado com um *timestamp* maior do que ts. Pela propriedade de intersecção dos quóruns, pelo Lema 30 (Casos 3 e 4) e como l_2 inicia após *wb*, é garantido que l_2 lê *val* ou um valor mais atual escrito em $V(t)$ ou $V(t')$.

4. r concorrente com l_1 e l_2: pelo Lema 29 (Caso 3), a fase de *write-back wb* de l_1 escreve *val* (e seu *timestamp ts*) em $V(t)$ e este valor é propagado para $V(t')$ ou escreve *val* diretamente em $V(t')$. Qualquer operação de escrita concorrente (Lema 29 - Caso 4) ou posterior à *wb* somente consegue escrever um valor associado com um *timestamp* maior do que ts. Pela propriedade de intersecção dos quóruns, pelo Lema 30 (Casos 3 e 4) e como l_2 inicia após *wb*, é garantido que l_2 lê *val* ou um valor mais atual escrito em $V(t)$ ou $V(t')$.

Em qualquer caso, l_2 lê *val* ou um valor mais atual, então os protocolos de leitura e escrita satisfazem as propriedades de segurança de um registrador de leitura e escrita atômico. $\square_{Teorema\ 9}$

Lema 31 *Cada servidor é capaz de requisitar somente a sua própria entrada ou saída do sistema.*

Prova. Um servidor i deve executar a operação de *join* (linhas 1-2 do algoritmo 8) ou *leave* (linhas 3-4 do algoritmo 8) para solicitar sua entrada ou saída do sistema, respectivamente. Nestas operações, uma tupla assinada por i deve ser enviada ao servidores da visão corrente. Além disso, a assinatura de cada *update* proposto pelos servidores é verificada durante a reconfiguração (passos GET-SIGN e SET-SIGN). Então, como não é possível forjar assinaturas, somente i é capaz de requisitar sua entrada ou saída do sistema. $\square_{Lema\ 31}$

Teorema 10 ***(Segurança 2)*** *Se um servidor correto $i \notin V(t)$ executa a operação de join no tempo t, então existe um tempo $t' > t$ tal que $i \in V(t')$.*

Prova. Pelo Lema 31, se $i \notin V(t) \wedge i \in V(t')$ para algum $t' > t$, então i requisitou sua entrada no sistema. Pelos Lemas 26 e 27, se i requisitar sua entrada no sistema em $V(t)$ e $i \notin V(t)$, então $i \in V(t')$ para algum $t' > t$. $\square_{Teorema\ 10}$

Teorema 11 ***(Segurança 3)*** *Se um servidor correto $i \in V(t)$ executa a operação de leave no tempo t, então existe um tempo $t' > t$ tal que $i \notin V(t')$.*

Prova. Pelo Lema 31, se $i \notin V(t') \wedge i \in V(t)$ para algum $t' > t$, então i requisitou sua saída no sistema. Pelos Lemas 26 e 27, se i requisitar sua saída do sistema em $V(t)$ e $i \in V(t)$, então $i \notin V(t')$ para algum $t' > t$. $\square_{Teorema\ 11}$

5.7 TOLERANDO CLIENTES MALICIOSOS

Um dos grandes desafios em sistemas de quóruns é desenvolver protocolos eficientes para tolerar comportamento malicioso por parte dos clientes,

pois os mesmos podem executar uma série de ações maliciosas com o objetivo de ferir as propriedades do sistema, como por exemplo realizar uma escrita incompleta, atualizando apenas alguns servidores. Esta característica é importante na medida em que os sistemas de quóruns foram desenvolvidos para operar em ambientes de larga escala, como a Internet, onde sua natureza de sistema aberto aumenta significativamente a possibilidade de clientes maliciosos estarem presentes no sistema. Se considerarmos um ambiente dinâmico e todas suas incertezas, esta possibilidade aumenta ainda mais.

Outras ações maliciosas que clientes podem executar, com o intuito de corromper o sistema, são (LISKOV; RODRIGUES, 2006): *(i)* escrever valores diferentes associados com o mesmo *timestamp*; *(ii)* executar o protocolo parcialmente, atualizando os valores armazenados por apenas algumas réplicas; *(iii)* escolher um *timestamp* muito grande, causando a exaustão do espaço de *timestamps*; e *(iv)* preparar um grande número de escritas e trabalhar junto com um aliado que executa estas operações mesmo após o cliente faltoso ser removido do sistema. Apesar de ser possível projetar protocolos que evitem a execução destas ações, a utilidade dos mesmos ainda é uma questão discutível, pois um cliente malicioso pode se comportar corretamente e escrever um valor qualquer no registrador (valor incorreto ou que não faz sentido para a aplicação), sobrescrevendo as escritas de clientes corretos.

Os primeiros protocolos a suportar clientes maliciosos necessitam que o sistema seja replicado em $4f+1$ servidores para suportar até f falhas, não utilizando assinaturas (MALKHI; REITER, 1998a, 1998b). Porém, estes protocolos permitem que um cliente malicioso execute uma série de ações maliciosas, como preparar várias escritas para serem executadas por um aliado após sua exclusão do sistema, interferindo assim no funcionamento dos clientes corretos. Estes problemas foram tratados no protocolo BFT-BC (*Byzantine fault-tolerance for Byzantine clients*) (LISKOV; RODRIGUES, 2006), que requer apenas $3f+1$ servidores e permite que um cliente prepare uma escrita somente após terminar a escrita anterior. Para isso, os servidores utilizam assinaturas assimétricas (principal fonte de atrasos em protocolos tolerantes a faltas bizantinas (CASTRO; LISKOV, 2002)) no controle das ações dos clientes e na garantia da integridade dos dados armazenados.

Implementar um sistema de quóruns replicado em apenas $3f+1$ servidores requer que os dados armazenados sejam auto-verificáveis, pois a intersecção entre os quóruns poderá conter apenas um servidor correto. Assim, os clientes obtêm corretamente os dados armazenados, a partir deste servidor, apenas se tais dados forem auto-verificáveis, pois deste modo é possível verificar a integridade dos mesmos. Neste sentido, a grande diferença do BFT-BC em relação a outros sistemas que armazenam dados auto-verificáveis está justamente na forma de garantir a integridade destes dados, onde o BFT-BC

utiliza um conjunto de assinaturas de servidores e outros protocolos, como o *f-dissemination quorum system* (MALKHI; REITER, 1998a, 1998b), utilizam assinaturas de clientes e, portanto, não toleram clientes maliciosos[3].

Para manter suas semânticas de consistência, o BFT-BC utiliza um mecanismo de provas assinadas pelos servidores em todas as suas etapas de execução. Desta maneira, para um cliente ingressar em uma nova fase do algoritmo, é necessário que o mesmo apresente uma prova de que completou a fase anterior. Esta prova, chamada de certificado, nada mais é que o conjunto das mensagens (assinadas) de resposta coletadas de um quórum de servidores na fase anterior. Por exemplo, para o cliente escrever no quórum é preciso que ele tenha terminado uma escrita anterior. Através desta técnica, o BFT-BC necessita de 1 ou 2 fases para realizar leituras e 2 ou 3 fases para executar escritas (LISKOV; RODRIGUES, 2006).

Recentemente, propomos um sistema de quóruns que estende o BFT-BC, modificando a forma de garantir a integridade dos dados e de controlar as ações dos clientes, que passam a ser realizadas através de criptografia de limiar (DESMEDT; FRANKEL, 1990). Nossa solução, chamada PBFT-BC (*Proactive Byzantine fault-tolerance for Byzantine clients*) (ALCHIERI et al., 2009b, 2009a, 2009c) reduz drasticamente a quantidade de dados trocados entre cliente e servidores, aumentando o desempenho do sistema. Nosso protocolo funciona de forma semelhante ao BFT-BC, modificando apenas a geração de certificados (provas), onde o cliente deve obter um conjunto de assinaturas parciais e combiná-las para então obter uma assinatura do serviço (Seção 5.7.1) e provar que executou determinada fase do protocolo. A utilização de criptografia de limiar fornece a flexibilidade necessária para adaptação do protocolo às mudanças que ocorrem na composição do grupo de servidores em um ambiente dinâmico. Por exemplo, os clientes apenas precisam conhecer uma única chave pública do serviço (Seção 5.7.1), ao invés de todas as chaves públicas dos servidores.

5.7.1 Criptografia de Limiar

A principal ferramenta utilizada no PBFT-BC é um esquema de assinatura de limiar (*threshold signature scheme* - TSS) (SHOUP, 2000) através do qual é possível controlar as ações dos clientes e garantir a integridade dos dados armazenados pelos servidores. Em um esquema (n,k)-TSS, um distribuidor confiável inicialmente gera n chaves parciais $(SK_1,...,SK_n)$, n chaves

[3] O protocolo *f-masking quorum system* (MALKHI; REITER, 1998a, 1998b) também tolera algumas ações de clientes maliciosos, mas requer replicação em $4f+1$ servidores, pois não armazena dados auto-verificáveis.

de verificação ($VK_1, ..., VK_n$), a chave de verificação de grupo VK e a chave pública PK usada para validar assinaturas. Além disso, o distribuidor envia estas chaves para n partes diferentes, chamados portadores. Assim, cada portador i recebe sua chave parcial SK_i e sua chave de verificação VK_i. A chave pública e as chaves de verificação são disponibilizadas para qualquer parte que compõe o sistema.

Após esta fase de configuração, o sistema está apto à gerar assinaturas. Deste modo, na obtenção de uma assinatura do serviço A para o dado $data$, primeiramente cada portador i gera a sua assinatura parcial a_i para $data$. Posteriormente, um combinador obtém pelo menos k assinaturas parciais válidas $(a_1, ..., a_k)$ e constrói a assinatura A através da combinação destas k assinaturas parciais. Uma propriedade fundamental deste esquema é a impossibilidade de gerar assinaturas válidas com menos de k assinaturas parciais. Este esquema é baseado nas seguintes primitivas:

- $Thresh\_Sign(SK_i, VK_i, VK, data)$: função usada pelo portador i para gerar sua assinatura parcial a_i sobre $data$ e as provas v_i de validade desta assinatura, i.e., $\langle a_i, v_i \rangle$.

- $Thresh\_VerifyS(data, \langle a_i, v_i \rangle, PK, VK, VK_i)$: função usada para verificar se a assinatura parcial a_i, apresentada pelo portador i, é válida.

- $Thresh\_CombineS(a_1, ..., a_k, PK)$: função usada para combinar k assinaturas parciais válidas e obter a assinatura A.

- $verify(data, A, PK)$: função usada para verificar a validade da assinatura A sobre o dado $data$ (verificação normal de assinatura).

O PBFT-BC utiliza o protocolo proposto em (SHOUP, 2000), onde é provado que tal esquema é seguro no modelo dos oráculos aleatórios (BELLARE; ROGAWAY, 1993), não sendo possível forjar assinaturas. Este protocolo representa um esquema de assinaturas de limiar baseado no algoritmo RSA (RIVEST et al., 1978), i.e., a combinação das assinaturas parciais gera uma assinatura RSA. Neste modelo a geração e verificação de assinaturas parciais é completamente não interativa, não sendo necessárias trocas de mensagens para executar estas operações.

Para redistribuição de chaves parciais, seguimos as definições apresentadas em (WONG et al., 2002) onde a transição de um esquema (n,k)-TSS para um esquema (n',k')-TSS funciona da seguinte forma: primeiro, cada um dos n portadores de (n,k)-TSS gera n' *shares* de sua chave parcial (um *share* para cada portador do novo esquema) e envia estes *shares* para os n' portadores do novo esquema (n',k')-TSS, os quais obtém pelo menos k *shares*

válidos e combinam suas novas chaves parciais. Um requisito fundamental para este protocolo funcionar é que, para as novas chaves parciais serem compatíveis, é necessário que os k *shares* combinados por cada portador do esquema (n',k')-TSS sejam gerados pelo mesmo conjunto de k portadores de (n,k)-TSS (WONG et al., 2002). Para redistribuição de chaves parciais entre estes dois esquemas, as seguintes funções são necessárias:

- $Th\_share(SK_i, j, n')$: função usada pelo portador i para gerar o *share* $\widehat{k}_{ij}$ de sua chave parcial SK_i. Este *share* é usado por j na obtenção de sua nova chave parcial.

- $Th\_generateV(i,r)$: função usada pelo portador i para gerar o verificador e_{ir}, o qual atesta que o *share* de sua chave parcial é válido. O índice do verificador é representado por r e varia de 0 até $n'-1$.

- $Th\_verifySK(\widehat{k}_{ji}, \{e_{j0}, ..., e_{j(n'-1)}\})$: função usada pelo portador i para verificar a validade de *share* $\widehat{k}_{ji}$, gerado por j. Cada e_{jr} representa o verificador gerado por j com índice r.

- $Th\_combineSK(\widehat{k}_{1i}, ..., \widehat{k}_{ki}, red\_set)$: função usada pelo portador i para obter sua nova chave parcial SK_i (e verificador VK_i) a partir da combinação de k *shares* válidos obtidos das chaves parciais da configuração antiga. Os identificadores dos portadores, a partir dos quais os *shares* devem ser usados nesta combinação, estão contidos em red_set.

5.7.2 QUINCUNX-BC

Esta seção discute como podemos integrar os protocolos do PBFT-BC aos do QUINCUNX, fazendo com que o sistema passe a tolerar também clientes maliciosos (QUINCUNX-BC). Como o PBFT-BC utiliza o esquema de criptografia de limiar anteriormente descrito, consideramos que na inicialização do sistema dinâmico cada servidor correto da visão inicial (Seção 5.3) recebe sua chave parcial, que é secreta e nunca pode ser revelada. Estas chaves são geradas e distribuídas por um administrador correto que apenas é necessário para a inicialização do sistema. Após isso, um protocolo de redistribuição de claves (Seção 5.7.3) deve ser empregado.

O PBFT-BC necessita de $n \geq 3f+1$ servidores e utiliza quóruns de $2f+1$ réplicas para tolerar até f faltas, empregando um esquema $(n, 2f+1)$-TSS, i.e., para gerar uma assinatura válida (certificado) é necessário um quórum de servidores. Assim, todas as ações realizadas pelos clientes precisam ser autorizadas por um quórum de servidores. Além disso, o PBFT-BC

emprega 1 ou 2 fases para realizar leituras e 2 ou 3 fases para executar escritas (ALCHIERI et al., 2009b, 2009a, 2009c).

Neste sentido, para cada visão v do QUINCUNX é definido um esquema de criptografia de limiar $(|v.members|, v.q)$-TSS, utilizado por servidores e clientes que executam operações na visão v, na geração de assinaturas (certificados) válidas, a fim de provar a execução de determinada fase do protocolo e garantir a integridade dos dados armazenados no sistema.

Os protocolos de leitura e escrita do PBFT-BC são facilmente integrados aos protocolos do QUINCUNX, bastando para isso que cada cliente verifique a visão corrente do sistema na execução de cada fase do protocolo, como descrito na Seção 5.5. No entanto, também é necessário que ocorra uma redistribuição de chaves entre as visões instaladas por qualquer protocolo do QUINCUNX. Esta redistribuição deve ser executada de modo que os servidores de uma visão anterior na sequência de visões instaladas redistribuam suas chaves parciais para os servidores da visão seguinte e assim por diante. Por exemplo, dada uma sequência de visões instaladas $... \rightarrow v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow ...$, o esquema $(|v_0.members|, v_0.q)$-TSS deve sofrer uma transição para o esquema $(|v_1.members|, v_1.q)$-TSS, que deve ser redistribuído para o esquema $(|v_2.members|, v_2.q)$-TSS, e assim por diante. A seção seguinte apresenta este protocolo de redistribuição de chaves, que deve ser executado sempre que uma visão mais atual é instalada no sistema.

5.7.3 Redistribuição de Chaves Parciais

Esta seção apresenta um protocolo para redistribuição de chaves parciais, onde os servidores de uma visão v ($(|v.members|, v.q)$-TSS) redistribuem suas chaves parciais para servidores de uma visão w ($(|w.members|, w.q)$-TSS). Este protocolo pode ser utilizado por qualquer sistema que emprega criptografia de limiar e suporta mudanças na composição do sistema.

Este protocolo garante que os servidores de w obtém suas chaves parciais a partir de *shares* gerados pelo mesmo conjunto de servidores presentes em v, o que é um requisito para as novas chaves parciais serem compatíveis (WONG et al., 2002). A ideia principal é fazer com que os servidores de w troquem mensagens até convergirem para a mesma informação a respeito da redistribuição `RD_INFO`, que é composta por dois conjuntos: `RD_INFO`.*good_set* - conjunto de $v.q$ servidores de v a partir dos quais os *shares* devem ser usados na geração das novas chaves parciais; `RD_INFO`.*bad_set* - conjunto de servidores maliciosos de v que devem ser desconsiderados.

O Algoritmo 12 representa este protocolo e deve ficar sempre ativo para atualizar a `RD_INFO` de uma visão ativa assim que novas informações

forem recebidas, causando a obtenção de uma nova chave parcial. Portanto, qualquer alteração em RD_INFO de um servidor é refletida em todos os outros servidores de uma determinada visão (invariante mantido pelo algoritmo).

Algoritmo 12 Redistribuindo chaves parciais da visão v para a w (servidor i)

variables: {Sets and arrays used in the protocol}
RDK, RDV - set of encrypted shares and set of verifiers generated by i
GOOD, BAD - set of encrypted good shares and set of invalid shares collected by i
SHARES - set of good shares collected by i
RD_INFO - information about the redistribution

procedure *Partial_Keys_Redist*(v,w)
1: **if** $i \in v$ **then**
2: **for all** $j \in w$ **do**
3: $\widehat{k}_{ij} \leftarrow Th\_shareK(SK_i, j, |w.members|)$; $\texttt{RDK}^w \leftarrow \texttt{RDK}^w \cup \langle j, encrypt(\widehat{k}_{ij})\rangle$
4: **end for**
5: **for** $r = 0$ *to* $|w.members| - 1$ **do**
6: $e_{ir} \leftarrow Th\_generateV(i,r)$; $\texttt{RDV}^w \leftarrow \texttt{RDV}^w \cup e_{ir}$
7: **end for**
8: $\forall j \in w, send\langle \text{REDIST}, \langle \texttt{RDK}^w, \texttt{RDV}^w, v, w\rangle_i\rangle$ to j
9: **end if**

Upon receipt of $\langle \text{REDIST}, \langle rdk, rdv, v, w\rangle_j\rangle$ from $j \in v \wedge i \in w$
10: **if** $(\texttt{GOOD}_j^w = \perp) \wedge (\texttt{BAD}_j^w = \perp)$ **then**
11: *extractSK*$(\langle rdk, rdv, v, w\rangle_j)$
12: *updateSK*(v,w)
13: **end if**

procedure *extractSK*$(\langle rdk, rdv, v, w\rangle_j)$
14: **if** $(\forall z \in w \rightarrow \exists \langle z, encrypted_z\rangle \in rdk) \wedge (\forall r \in \{0,...,|w.members|-1\} \rightarrow \exists e_{jr} \in rdv)$ **then**
15: $\widehat{k}_{ji} \leftarrow decrypt(encrypted_i)$
16: **if** $Th\_verifySK(\widehat{k}_{ji}, rdv)$ **then**
17: $\texttt{GOOD}_j^w \leftarrow \langle rdk, rdv, v, w\rangle_j$; $\texttt{SHARES}_j^w \leftarrow \widehat{k}_{ji}$
18: **else**
19: $\texttt{BAD}_j^w \leftarrow \langle j, \widehat{k}_{ji}, \langle rdk, rdv, v, w\rangle_j\rangle$
20: **end if**
21: **else**
22: $\texttt{BAD}_j^w \leftarrow \langle j, \perp, \langle rdk, rdv, v, w\rangle_j\rangle$
23: **end if**

procedure *updateSK*(v,w)
24: $\texttt{RD\_INFO}^w.good\_set \leftarrow$ *select v.q members in* $\texttt{GOOD}^w$ *with lower ids*
25: **if** $|\texttt{RD\_INFO}^w.good\_set| = v.q$ **then**
26: $\texttt{RD\_INFO}^w.bad\_set \leftarrow$ *select all members in* $\texttt{BAD}^w$
27: **if** $\texttt{RD\_INFO}^w$ updated on line 24 or 26 **then**
28: $\langle SK_i, VK_i\rangle \leftarrow Th\_combineSK(\texttt{SHARES}^w, \texttt{RD\_INFO}^w.good\_set)$
29: $\forall j \in w, send\langle \text{REDIST-INFO}, \texttt{GOOD}^w, \texttt{BAD}^w, v, w\rangle$ to j
30: $\forall j \in w$, *update* VK_j *by fetching it from* j *for* $\texttt{RD\_INFO}^w$
31: **end if**
32: **end if**

Upon receipt of $\langle \text{REDIST-INFO}, good, bad, v, w\rangle$ from j
33: **for all** $\langle z, \widehat{k}_{zx}, \langle rdk, rdv, v, w\rangle_z\rangle \in bad$: $(z \in v) \wedge (\texttt{BAD}_z^w = \perp) \wedge isBad(\widehat{k}_{zx}, \langle rdk, rdv, v, w\rangle_z)$ **do**
34: $\texttt{BAD}_z^w \leftarrow \langle z, \widehat{k}_{zx}, \langle rdk, rdv, v, w\rangle_z\rangle$; $\texttt{GOOD}_z^w \leftarrow \perp$; $\texttt{SHARES}_z^w \leftarrow \perp$
35: **end for**
36: **for all** $\langle rdk, rdv, v, w\rangle_z \in good$: $(z \in v) \wedge (\texttt{GOOD}_z^w = \perp) \wedge (\texttt{BAD}_z^w = \perp)$ **do**
37: *extractSK*$(\langle rdk, rdv, v, w\rangle_z)$
38: **end for**
39: *updateSK*(v,w)

Gerando e redistribuindo chaves parciais. Para iniciar o processo de redistribuição de chaves parciais, todos os servidores de v (servidores que redistribuirão suas chaves) geram os *shares* de suas chaves parciais (linha 3), juntamente com os verificadores que atestam a validade destes *shares* (linha 6), e enviam estes dados para os servidores de w (servidores que obterão uma nova chave parcial). Para cada portador $j \in w$, um servidor $i \in v$ gera o *share* $\widehat{k}_{ij}$ e cifra este dado com a chave pública de j, deste modo somente j consegue acessar este dado. Todos os verificadores são públicos.

As chaves parciais dos servidores de v, utilizadas no procedimento de redistribuição, devem ter sido obtidas a partir do mesmo conjunto de *shares* ($\texttt{RD\_INFO}^v.good\_set$). Então, caso um servidor de v atualizar sua chave parcial (atualizar $\texttt{RD\_INFO}^v$), o mesmo deve executar novamente este procedimento e os servidores de w devem obter suas chaves parciais a partir de *shares* obtidos de chaves parciais geradas através dos dados mais atuais em $\texttt{RD\_INFO}^v.good\_set$ (por simplicidade, isso não aparece no algoritmo).

Convergindo para o mesmo `RD_INFO`**.** Quando um servidor $i \in w$ recebe os *shares* de um servidor $j \in v$, i verifica se ainda não recebeu qualquer informação de j (linha 10), utilizando dois conjuntos: `GOOD` - conjunto de *shares* cifrados válidos; e `BAD` - conjunto de *shares* inválidos. Caso estas informações sejam novas, i decifra seu *share* e caso o mesmo seja válido armazena estes dados em `SHARES` e `GOOD` (linha 17). Do contrário, j é malicioso e estes dados são armazenados em `BAD` (linhas 19 e 22).

A função *updateSK* é a parte principal do protocolo, onde sempre que alguma informação mais atual é recebida por algum servidor $i \in w$, i atualiza sua chave parcial (caso necessário). Além disso, sempre que alguma nova informação for recebida, i envia uma mensagem REDIST-INFO para os servidores de w contendo todas as informações coletadas, de forma que todos estes servidores possam coletar as mesmas informações sobre o processo de redistribuição das chaves parciais. Para isso, sempre que um servidor receber uma mensagem REDIST-INFO (linhas 33-39), o mesmo atualiza seus conjuntos `GOOD`, `BAD` e `SHARES` de acordo com as informações recebidas e executa a função *updateSK* para verificar se uma chave parcial nova deve ser obtida.

Um servidor $i \in w$ verifica a informação de que um servidor $z \in v$ gerou um *share* inválido (função $isBad(\widehat{k}_{zx}, \langle rdk, rdv, v, w\rangle_z)$) da seguinte forma: (1) i verifica se z gerou *shares* para todos os servidores de w, bem como todos os verificadores (como na linha 14); (2) i cifra o *share* $\widehat{k}_{zx}$ (*share* enviado para um servidor $x \in w$) com a chave pública de x e verifica se o resultado é igual ao *share* cifrado enviado de z para x ($encrypted_x$ contido em rdk), e caso seja igual, i verifica a validade de $\widehat{k}_{zx}$ através da função $Th\_verifySK(\widehat{k}_{zx}, rdv)$. Uma vez que i descobre que z é malicioso, i adiciona z no conjunto `BAD`, de onde nunca é removido.

Comparando RD_INFO**.** Para que os processos possam obter assinaturas válidas, através do mecanismo de criptografia de limiar, os mesmos devem usar assinaturas parciais geradas por chaves compatíveis. Para isso, as mensagens trocadas entre os processos (tanto entre servidores para redistribuição de chaves parciais quanto entre clientes e servidores para execução de operações de leitura e escrita) devem conter a informação RD_INFO usada na obtenção da chave parcial mais atual do emissor, de modo que os processos apenas considerem as mensagens acompanhadas de informações mais atuais. Caso algum servidor tenha enviado uma mensagem com informação antiga, o mesmo deve obter uma nova chave parcial (considerando o RD_INFO atualizado) e enviar tal mensagem novamente. Note que deve ser obtido um certificado para provar a autenticidade do conjunto RD_INFO, uma vez que estes dados são trocados entre os processos. Isto pode ser realizado através da obtenção de uma assinatura de serviço usando o próprio mecanismo de criptografia de limiar.

Neste sentido, dadas duas informações sobre a redistribuição rd_info_1 e rd_info_2, rd_info_2 é mais atual que rd_info_1 se $max(rd\_info_1.good\_set) > max(rd\_info_2.good\_set) \wedge (\forall j \in rd\_info_2.good\_set \rightarrow j \notin rd\_info_1.bad\_set)$: o maior identificador em $rd\_info_2.good\_set$ é menor do que o maior identificador em $rd\_info_1.good\_set$, o que implica que rd_info_2 é formado por servidores com identificadores menores. A segunda parte da condição verifica se o conjunto $rd\_info_2.good\_set$ é formado apenas por servidores corretos.

5.7.3.1 *Corretude*

O seguinte teorema prova a propriedade fundamental garantida pelo Algoritmo 12, usado para redistribuição de chaves entre visões.

Teorema 12 *Considere uma redistribuição de chaves parciais de uma visão v para uma visão w através do Algoritmo 12. Então, cada servidor $i \in w$ obtêm uma nova chave parcial e este algoritmo mantém o invariante de que as chaves parciais definidas para os servidores de w sempre acabam por serem compatíveis.*

Prova. Cada servidor correto de v define os *shares* de sua chave parcial e envia estes dados para os servidores de w (linhas 1-9). Como v sempre possui pelo menos $v.q$ servidores corretos, o predicado da linha 25 será verdadeiro pelo menos uma vez em cada servidor de w, que obterá uma nova chave parcial. Agora temos que provar que os servidores de w sempre acabam definindo o mesmo RD_INFO, pois assim estes servidores obtém suas chaves parciais do mesmo conjunto de servidores de v fazendo com que as mesmas sejam compatíveis. Sempre que um servidor $i \in w$ alterar seu RD_INFO (linhas

24 e 26), *i* envia todas as suas informações coletadas sobre a redistribuição para todos os outros servidores de *w* (linha 29), os quais também atualizam seus conjuntos RD_INFO (linhas 33-39). Desta forma, todos os servidores de *w* acabam definindo um mesmo RD_INFO. Como os servidores de *w* obtém suas chaves parciais a partir dos *shares* gerados pelo quórum de servidores corretos de *v* com menores identificadores (linha 24) e acabam definindo o mesmo RD_INFO, todos estes servidores acabam obtendo novas chaves parciais compatíveis. Como sempre que algum servidor de *w* atualiza seu RD_INFO o mesmo envia estas informações para os outros servidores de *w*, este invariante sempre é mantido no sistema. $\square_{Teorema\ 12}$

5.8 DISCUSSÃO

Esta seção realiza algumas discussões sobre os protocolos propostos.

Registrador livre de espera (*Wait-freedom*). Como já mencionado, quando existir uma reconfiguração concorrente com uma operação de leitura ou escrita (ou uma operação de *join* ou *leave*), pode ser necessário reiniciar alguma fase desta operação para que o cliente acesse a configuração mais atual do sistema. Pela Suposição 4, o número de solicitações de atualizações realizadas em uma execução é finito, então existe um tempo depois do qual não ocorrem mais reconfigurações e as operações pendentes de leitura e/ou escrita terminam. De fato, apenas o número de reconfigurações concorrentes com cada operação de leitura ou escrita (e de *join* ou *leave*) deve ser finito, possibilitando que estas operações sempre terminem.

Desempenho. Como no protocolo base, leituras e escritas são executados em 1 ou 2 e 2 fases, respectivamente (MALKHI; REITER, 1998b). Uma reconfiguração do QUINCUNX exige um número variado de passos de comunicação, de acordo com o conjunto de protocolos utilizados. Nestes protocolos, é possível eliminar a necessidade dos servidores certificarem suas propostas (removendo os passos SET-SIGN e GET-SIGN) aumentando o desempenho dos protocolos de reconfiguração em execuções livres de falhas. No entanto, servidores maliciosos poderiam realizar diversas propostas diferentes para uma nova visão, fazendo com que a convergência para uma nova visão fosse mais demorada no gerador de visões seguro e infinitas visões pudessem ser geradas pelo gerador de visões vivo.

Vale destacar que as reconfigurações do QUINCUNX são completamente desacopladas dos protocolos de leitura e escrita, possibilitando que operações de clientes e reconfigurações executem em paralelo. Desta forma, o custo de uma reconfiguração não interfere no custo das operações de leitura e escrita. Esta abordagem aumenta o desempenho do sistema quando

comparada com outras soluções onde estes protocolos estão acoplados (ex.: *DynaStore* (AGUILERA et al., 2009, 2011)).

Considerações práticas sobre reconfigurações. Como servidores podem entrar e sair do sistema arbitrariamente, é possível que em alguma reconfiguração muitos servidores corretos deixem o sistema enquanto muitos servidores maliciosos entrem no mesmo. Esta reconfiguração poderá instalar no sistema uma visão não segura v, onde o limite de falhas toleradas $v.f$ não seja respeitado. Uma vez que este cenário viola a Suposição 5, os algoritmos do QUINCUNX não garantem nenhuma propriedade do sistema (Seção 5.3) nesta configuração.

Este cenário representa uma limitação fundamental de sistemas dinâmicos (ex.: (LYNCH; SHVARTSMAN, 2002; AGUILERA et al., 2009, 2011; MARTIN; ALVISI, 2004; RODRIGUES; LISKOV, 2004)) e apenas pode ser solucionado através do uso de controle de acesso e de autorização das requisições de atualizações por um terceira parte confiável. Os detalhes destas soluções estão fora do escopo deste trabalho. No entanto, esta parte confiável também pode ser usada para remover servidores maliciosos do sistema, agindo como estes servidores em relação ao protocolo de reconfiguração, i.e., executando a operação de *leave*.

Forma alternativa de receber visões atualizadas. Todas as mensagens do QUINCUNX (nos protocolos de leitura, escrita e reconfiguração) contém a visão corrente do emissor. Consequentemente, quando um servidor receber uma mensagem, o mesmo verifica se esta mensagem contêm uma visão válida mais atual do que sua visão corrente. Se este for o caso, o servidor deve aguardar pela instalação da visão mais atual. De fato, a reconfiguração que instalará esta visão deve estar em progresso, mas ainda não terminada.

Verificação da validade das visões. Uma visão v do QUINCUNX possui um campo que indica sua visão geradora $v.ov$, onde a validação de v, para qualquer uma das formas de reconfiguração, é realizada recursivamente (Seção 5.6.2), i.e., para validar v deve-se validar também $v.ov$ e assim por diante. Neste sentido, algumas otimizações podem ser incorporadas nos protocolos:

- Um processo i não necessita validar a sequência completa de visões $v.ov.ov...$, mas apenas até encontrar alguma visão já validada no passado.
- Não é necessário enviar toda a sequência de visões nas mensagens, bastando enviar apenas a parte final da sequência, a qual o processo receptor ainda não conhece. Além disso, quando esta sequência já for conhecida pelo receptor, basta enviar o *hash* da mesma para verificação.
- Pode-se utilizar um administrador correto que periodicamente assina

uma determinada visão, eliminando a necessidade de validação de sua visão geradora.

Por fim, uma solução mais apropriada para este problema é a utilização de criptografia de limiar, onde apenas a assinatura do serviço deveria ser verificada para validação de uma determinada visão v, pois tal assinatura apenas é obtida através da participação de um quórum de servidores da visão anterior $v.ov$ (Seção 5.7), não sendo necessário verificar assinaturas individuais de cada servidor de $v.ov$ e portanto não precisando saber suas identidades.

Limite de faltas toleradas em cada visão. Dada uma visão v do QUINCUNX, o limite de faltas toleradas no sistema $v.f$ sempre deve ser mantido (Suposição 5), mesmo após o sistema ser reconfigurado para outra visão. Isto é necessário para evitar vários ataques que poderão ser executados pelos servidores de v após a reconfiguração do sistema para uma visão mais atual v', i.e., após v deixar de ser ativa.

Por exemplo, se em algum momento pelo menos $v.q$ servidores de v se tornarem maliciosos, é possível que tais servidores gerem e instalem no sistema uma nova visão $v'' \neq v'$ ($v \rightarrow v''$). Neste caso, é possível que alguns clientes acessem v' e outros v'', causando uma divergência e ferindo as propriedades de segurança do sistema. Além disso, estes servidores maliciosos até mesmo podem responder a algum cliente atrasado como se v fosse a visão mais atual do sistema, mesmo já tendo reconfigurado para v'.

Estes problemas se devem ao fato de que na reconfiguração de v para v', apenas os servidores de v que deixaram o sistema descartam suas chaves e servidores corretos de v poderão se tornar maliciosos em v' de forma que, mesmo respeitado $v'.f$, o limite $v.f$ é ferido. Deste modo, como estes servidores ainda possuem suas chaves, os mesmos estão aptos à executar os ataques anteriormente citados.

Basicamente, este problema surge devido ao fato dos servidores manterem suas chaves e com isso poderem continuar executando os protocolos como se estivessem em v. Existem duas formas de solucionar este problema, possibilitando que o limite $v.f$ seja respeitado apenas enquanto v estiver ativa no sistema:

- Criptografia de limiar: nesta abordagem, as chaves parciais utilizadas na obtenção da assinatura do serviço são redistribuídas de v para v', onde os servidores de v que permanecerem no sistema atualizam tais chaves (no máximo $v.f$ servidores maliciosos mantém suas chaves parciais). Desta forma, substituindo todos os certificados do protocolo (conjunto de um quórum de assinaturas) por uma assinatura do serviço, este problema é solucionado uma vez que, após a redistribuição, não é mais possível obter uma assinatura do serviço com o esquema de crip-

tografia de limiar relacionado com a visão v.

- Usando dois pares de chaves (fixo e temporário): nesta abordagem, cada servidor utiliza dois pares de chaves, um fixo que é utilizado na obtenção de um temporário. Desta forma, cada servidor possui um par de chaves temporário para cada visão v do sistema, o qual é descartado após a reconfiguração de v (no máximo $v.f$ servidores maliciosos mantém suas chaves temporárias), impedindo que novos certificados possam ser obtidos pelos servidores de v.

Ataque *sybil*. Como os pedidos para entrar ou sair do sistema devem ser assinados, é impossível que servidores maliciosos executem um ataque de *sybil* (DOUCEUR, 2002) contra o sistema visto que os mesmos não conseguem gerar múltiplas identidades.

5.9 CONSIDERAÇÕES FINAIS

Este capítulo apresentou os protocolos do QUINCUNX, o qual representam o primeiro sistema de quóruns bizantino dinâmico que não utiliza consenso para reconfigurações. O QUINCUNX implementa um registrador de leitura e escrita com as seguintes propriedades:

1. Tolera servidores e leitores maliciosos, além de faltas por parada de escritores. Na seção 5.7 é apresentado um protocolo para tolerar também escritores maliciosos.

2. Operações com semântica atômica (LAMPORT, 1986).

3. Operações de leitura e escrita livre de espera (*wait-freedom*).

4. Resiliência ótima: para cada visão v, $|v.members| \geq 3v.f + 1$.

5. Capaz de modificar o conjunto de servidores ativos, que implementam o registrador, durante o uso do mesmo. Para isso, vários protocolos são apresentados onde é possível combiná-los de acordo com as características desejadas para o sistema (ex.: preferir reconfigurações seguras ao invés de garantir que sempre terminem).

Os protocolos de reconfiguração do QUINCUNX foram projetados de forma a serem o mais desacoplados possíveis dos protocolos de leitura e escrita. Assim, é possível termos concorrência entre estes protocolos, aumentando o desempenho do sistema durante reconfigurações.

Além disso, dado este desacoplamento, é razoavelmente fácil adaptar os protocolos de reconfiguração do QUINCUNX para qualquer outro sistema baseado em quóruns, disponibilizando reconfiguração livre de consenso para sistemas tolerantes a faltas bizantinas. Prova disso é que neste capítulo discutimos como este protocolos podem ser facilmente incorporados tanto no PHALANX quanto no PBFT-BC.

6 REPLICAÇÃO MÁQUINA DE ESTADOS DINÂMICA

Este capítulo apresenta alguns conceitos envolvidos na adição do suporte à reconfigurações em uma replicação Máquina de Estados, seguindo uma abordagem mais prática. Neste sentido, é discutido como prover tal funcionalidade no sistema BFT-SMART, o qual é uma concretização de uma replicação Máquina de Estados.

6.1 INTRODUÇÃO

A replicação Máquina de Estados (SCHNEIDER, 1990), também chamada de RME, é a abordagem mais abrangente e também a mais usada na implementação de sistemas tolerantes a faltas, tanto por parada (SCHNEIDER, 1990) quanto bizantinas (CASTRO; LISKOV, 2002). Esta abordagem consiste em replicar os servidores e coordenar as interações entre os clientes e as réplicas dos servidores, com o intuito de que as várias réplicas apresentem a mesma evolução em seus estados.

O ponto fundamental de uma RME é a necessidade de ordenação das requisições dos clientes executadas pelas réplicas (servidores) que implementam o sistema, a fim de manter o determinismo de réplicas (Seção 3.4.5). Para isso, é necessário o emprego de um protocolo de difusão atômica (Seção 3.4.3), que geralmente é implementado através de um algoritmo de consenso, onde todas as réplicas entram em acordo sobre a ordem de execução das requisições, as quais são então organizadas em uma sequência que é executada no sistema.

As concretizações existentes para replicação Máquina de Estados (ex.: (CASTRO; LISKOV, 2002)), bem como os protocolos propostos para difusão atômica (ex.: (CORREIA et al., 2006)), consideram que o conjunto de réplicas que implementam o sistema, além dos parâmetros de configuração do mesmo, nunca sofre alterações. Uma grata exceção é o sistema SMART (LORCH et al., 2006) que implementa uma RME onde apenas o conjunto de servidores pode ser alterado, sendo que os mesmos podem falhar apenas por parada.

A abordagem estática para replicação Máquina de Estados impossibilita, por exemplo, que durante a execução do sistema novas réplicas sejam adicionadas ou réplicas antigas sejam removidas e/ou que qualquer outro parâmetro de configuração (como o limite de falhas suportadas) seja alterado. Para possibilitar a execução destas ações, surge a necessidade de *reconfiguração* do sistema.

Um sistema reconfigurável fornece as interfaces necessárias para sua reconfiguração, não se preocupando com aspectos relacionados com a escolha tanto do momento da reconfiguração quanto da nova configuração a ser adotada pelo sistema. De qualquer forma, os algoritmos que implementam estes sistemas devem prever a possibilidade de entradas e saídas de réplicas do mesmo, sendo que estas ações de reconfiguração devem ser sincronizadas com as execuções dos protocolos de difusão atômica do sistema, a fim de permitir que todas as réplicas executem a mesma sequência de operações (ou armazenam estados que refletem a mesma sequência de operações), preservando assim a consistência de seus estados.

Recentemente, Lamport *et al.* (LAMPORT et al., 2010) apresentam uma discussão, numa visão bastante ampla e pouco aprofundada (sem apresentar protocolos e nem mesmo se aprofundar nas questões envolvidas em uma reconfiguração), sobre como reconfigurar uma replicação Máquina de Estados. Nesta abordagem, a reconfiguração da RME é realizada através da própria execução da RME, i.e., a própria RME define a nova configuração do sistema. Apesar de bastante intuitiva, existem vários problemas não triviais que devem ser tratados em uma concretização desta reconfiguração. Também em vista disto, ainda não existe nenhuma implementação de uma RME que tolera comportamento malicioso de servidores e possui capacidade de reconfiguração.

Este capítulo apresenta nossos esforços para adicionar suporte à reconfigurações no sistema BFT-SMART, que é uma concretização de uma replicação Máquina de Estados tolerante a faltas bizantinas. De acordo com nossos conhecimentos, a inclusão desta funcionalidade no BFT-SMART o torna a primeira implementação de uma RME capaz de tolerar servidores maliciosos e de alterar o conjunto de réplicas do sistema em tempo de execução. Além disso, os mecanismos incluídos no BFT-SMART também fornecerão suporte para que outros parâmetros de configuração do sistema possam ser alterados durante a execução do mesmo.

Seguindo a abordagem de Lamport *et al.* (LAMPORT et al., 2010), as reconfigurações no BFT-SMART ocorrem entre execuções do protocolo de ordenação (consenso), de modo que a própria RME é usada para definir a nova configuração do sistema. De fato, não faz muito sentido reconfigurar o sistema em meio a uma execução do protocolo de consenso, visto que estes protocolos geralmente terminam rapidamente (em alguns milissegundos – (CASTRO; LISKOV, 2002)) e a RME em execução já fornece um suporte bastante forte que pode ser utilizado nas reconfigurações. Reconfigurar o sistema em meio a uma execução do consenso implicaria em adicionar muita complexidade neste protocolo e atrelar o mesmo a um protocolo de consenso específico, i.e., com capacidade de reconfiguração. Além

disso, estas limitações seriam introduzidas no sistema sem ganhos práticos: *(1)* a execução do consenso deveria ser paralizada; *(2)* o sistema reconfigurado, onde a nova configuração possivelmente seria escolhida através de uma execução do consenso; *(3)* e então a execução do consenso finalizada. Como veremos neste capítulo, a reconfiguração através da própria RME demanda no máximo uma execução do consenso (ou uma execução do protocolo de ordenação da RME) para definir a nova configuração do sistema.

6.2 MODELO DE SISTEMA

Consideramos um sistema distribuído completamente conectado composto pelo conjunto universo de processos U, que é dividido em dois conjuntos: um conjunto infinito de servidores $\Pi = \{s_1, s_2, ...\}$ e um conjunto infinito de clientes $C = \{c_1, c_2, ...\}$. A chegada dos processos segue o modelo de chegadas infinita com concorrência desconhecida mas finita (modelo M_2^{finito} da Tabela 1).

Os processos do sistema estão sujeitos a *faltas bizantinas* (LAMPORT et al., 1982), i.e., processos faltosos podem exibir qualquer comportamento, podendo parar, omitir envio ou entrega de mensagens, ou desviar de suas especificações arbitrariamente e trabalhar em conjunto com o objetivo de corromper o sistema. Um processo que apresenta comportamento de falha é dito falho (ou faltoso), de outra forma é dito correto.

Como veremos a seguir, o número máximo de faltas toleradas pelo sistema, denotado por f, pode ser reconfigurado durante a execução do mesmo. No entanto, sempre é necessário pelo menos $3f + 1$ réplicas para tolerar até f réplicas faltosas no sistema em um dado momento.

Com relação ao modelo de síncronia, consideramos um sistema parcialmente síncrono (DWORK et al., 1988). A ideia por trás destes modelos é de que o sistema trabalha de forma assíncrona (não respeitando nenhum limite de tempo) a maior parte do tempo. Porém, durante períodos de estabilidade, o tempo para transmissão de mensagens é limitado. Além disso, as comunicações entre os processos são realizadas através de canais ponto-a-ponto confiáveis e autenticados.

Finalmente, cada processo possui um par distinto de chaves (chave pública e privada) para usar um sistema de criptografia assimétrica. Cada chave privada é conhecida apenas pelo seu próprio dono, por outro lado todos os processos conhecem todas as chaves públicas. Cada processo do sistema (cliente ou servidor) possui um identificador único, representado por este par de chaves obtidas junto a uma autoridade certificadora, sendo inviável a obtenção de identificadores adicionais por processos faltosos com o objetivo

de lançar um ataque *Sybil* (Seção 3.2.1) contra o sistema.

6.3 BFT-SMART

O BFT-SMART (BESSANI et al., 2011) representa a concretização de uma replicação Máquina de Estados (SCHNEIDER, 1990) tolerante a faltas bizantinas (LAMPORT et al., 1982). Esta biblioteca de replicação foi desenvolvida na linguagem de programação Java e implementa um protocolo similar aos outros protocolos para tolerância a faltas bizantinas (ex.: (CASTRO; LISKOV, 2002)), mas que se preocupa tanto com o desempenho do sistema quanto com a corretude do mesmo nos mais diversos cenários originados pelo comportamento malicioso de réplicas.

O BFT-SMART surgiu da camada de replicação do sistema DEPSPACE (BESSANI et al., 2008), o qual representa a implementação de um espaço de tuplas tolerante a faltas bizantinas que utiliza replicação Máquina de Estados para garantir a consistência dos estados das réplicas do sistema. Atualmente, o BFT-SMART conta com protocolos para *checkpoints* e transferência de estados, tornando-se assim uma biblioteca completa para RME, a qual foi desenvolvida seguindo os seguintes princípios:

- Java: a escolha desta linguagem de programação visa a obtenção de uma implementação que apresenta características de portabilidade, segurança, facilidade de programação e manutenção.

- Modularidade: o BFT-SMART foi projetado de forma modular, apresentando uma notável separação entre os protocolos de consenso (o algoritmo de consenso utilizado é o *Paxos at War* (ZIELINSKI, 2004)), de difusão atômica, de *checkpoints* e de transferência de estado.

- Desprovido de otimizações que aumentam a complexidade dos algoritmos: além de adicionar complexidade, a utilização de otimizações "frágeis" torna os protocolos mais susceptíveis à ataques de degradação de performance (CLEMENT et al., 2009). Desta forma, o BFT-SMART não implementa algumas otimizações como a execução do acordo sobre *hashes* e especulação.

Estes princípios de projeto fazem do BFT-SMART uma concretização de uma biblioteca de replicação completa que pode ser usada no "mundo real", bem como ser evoluída pela comunidade acadêmica.

6.3.1 Arquitetura Básica do BFT-SMART

Esta seção discute os aspectos básicos da arquitetura do BFT-SMART, os quais são importantes para entender a forma de como as reconfigurações são executadas no sistema.

Como já comentado, o protocolo de ordenação das requisições utiliza o algoritmo de consenso *Paxos at war* (ZIELINSKI, 2004). Basicamente, este protocolo funciona da seguinte forma: o valor de decisão da instância *i* do *Paxos* é a *i*-ésima requisição a ser entregue para a aplicação. Desta forma, a entrega das requisições segue a mesma ordem nas diversas réplicas. Apesar da aparente simplicidade, alguns outros problemas devem ser tratados: *(1)* no protocolo de consenso, a réplica líder pode propor qualquer valor de tal forma que a decisão não seja uma requisição válida e autêntica (requisições forjadas); e *(2)* um líder malicioso pode executar o protocolo de consenso corretamente, mas nunca propor uma ordem para requisições de determinado(s) cliente(s) (negação de serviço para clientes). No BFT-SMART, os seguintes mecanismos foram incorporados ao sistema para evitar que réplicas maliciosas sejam capazes de executar qualquer uma destas ações:

- Autenticidade de Requisições: a autenticidade das requisições é garantida por meio de assinaturas digitais, i.e., os clientes devem assinar suas requisições. Desta forma, qualquer réplica é capaz de verificar a autenticidade das requisições e uma proposta para ordenação, a qual contém a requisição a ser ordenada, somente é aceita por uma réplica correta após a autenticidade desta requisição ser verificada.

- Garantia de Ordenação de Requisições: caso uma requisição não seja ordenada dentro de um determinado tempo, o sistema força a troca da réplica líder. Para cada requisição *r* recebida em determinada réplica *i*, um tempo limite para ordenação é associado à *r*. Caso este tempo se esgotar, *i* envia *r* para todas as réplicas e define um novo tempo para sua ordenação. Isto garante que todas as réplicas recebem *r*, pois um cliente malicioso pode ter enviado *r* apenas para alguma(s) réplica(s). Caso este tempo se esgotar novamente, *i* solicita a troca de líder, que apenas é executada após $f + 1$ réplicas solicitarem esta mudança, impedindo que uma réplica maliciosa force trocas de líder.

A escolha do método de reconfiguração do sistema está diretamente relacionada com a forma de como o protocolo de ordenação das requisições é implementado no mesmo (LAMPORT et al., 2010). Neste sentido, a seguir descrevemos duas características fundamentais do protocolo de ordenação do BFT-SMART:

- Ordenação em lote: no BFT-SMART, cada instância do consenso define a ordem de entrega de um lote de requisições ao invés de apenas uma única requisição. As requisições de um lote devem ser entregues seguindo uma mesma ordem em todos as réplicas (ex.: entregues de acordo com a ordem de seus identificadores). Esta abordagem aumenta o desempenho do sistema, uma vez que várias requisições são entregues através da execução de uma única instância do consenso. No entanto, é possível que requisições de reconfiguração do sistema (que são ordenadas juntamente com as requisições de clientes – Seção 6.4) estejam localizadas no meio de determinado lote, "misturadas" com requisições de clientes, e isto deve ser previsto pelos protocolos de reconfiguração, como veremos na Seção 6.4.

- Sem concorrência na execução de instâncias do consenso: outra abordagem que visa aumentar o desempenho do sistema é a execução em paralelo de várias instâncias do algoritmo de consenso, de modo que várias requisições (ou lotes de requisições) sejam ordenados simultaneamente. Além desta abordagem trazer mais complexidade aos protocolos, a mesma possibilita que líderes faltosos degradem o desempenho do sistema, uma vez que é necessário executar instâncias do consenso (iniciadas por estes processos maliciosos) mesmo quando as propostas são para definir a ordem de entrega de requisições forjadas. Neste caso, a instância do consenso define a ordem para uma requisição *nop* (*no operation*), o que é necessário para não "travar" a entrega das mensagens (CASTRO; LISKOV, 2002). Em vista disso, o BFT-SMART opta por executar uma única instância do consenso por vez e preserva o desempenho do sistema através da ordenação em lotes (BESSANI et al., 2008). Como descrito por Lamport *et al.* (LAMPORT et al., 2010), esta abordagem também torna o protocolo de reconfiguração menos complexo (Seção 6.4) quando comparado com soluções onde várias instâncias do consenso são executadas em paralelo.

6.3.2 Usando o BFT-SMART

A forma de utilização do BFT-SMART, para programação de uma aplicação tolerante a faltas bizantinas através de replicação Máquina de Estados, é bastante simples. A Figura 14 apresenta a API para clientes e servidores, mostrando a classe que deve ser instanciada pelo clientes para acessar o sistema, bem como a classe que deve ser estendida pelos servidores para implementar o serviço replicado.

Para acessar o serviço replicado, um cliente do BFT-SMART apenas

deve instanciar uma classe *ServiceProxy* com um arquivo de configuração contendo o endereço (IP e porta) de cada um dos servidores, bem como suas chaves públicas. Então, sempre que o cliente desejar enviar alguma requisição para as réplicas (servidores), o mesmo deve invocar o método *invoke* especificando a requisição (serializada em um *array* de *bytes*) e indicando se tal requisição é apenas de leitura. Requisições de apenas leitura não modificam o estado das réplicas e, deste modo, não precisam ser ordenadas, sendo entregues diretamente para a aplicação.

```
//API do Cliente
public class ServiceProxy ... {
  ...
  public byte[] invoke(byte[] command, boolean readOnly);
  ...
}


//API do Servidor
public abstract class ServiceReplica ...{
   ...
   public abstract byte[] executeCommand(int clientId, long timestamp,
                                         byte[] nonces, byte[] command);
   public abstract byte[] serializeState();
   public abstract byte[] deserializeState(byte[] state);

}
```

Figura 14 – API do BFT-SMART para clientes e servidores.

Por outro lado, para implementar o servidor, cada réplica deve estender a classe *ServiceReplica* e implementar os métodos abstratos que são invocados quando uma requisição deve ser executada (é entregue pelo protocolo de ordenação) ou quando é necessário obter/atualizar o estado da réplica. Os métodos para atualização ou obtenção dos estados das réplicas são utilizados pelos mecanismos de *checkpoints* e transferência de estados, sendo indispensáveis para a recuperação de réplicas faltosas ou atualização de réplicas atrasadas (BESSANI et al., 2011).

Note que o método *executeCommand* também fornece o identificador do cliente, um *timestamp* e um conjunto de *nonces* randômicos, definidos pela réplica líder da instância do consenso que definiu a ordem de execução da requisição correspondente. Como é garantido que todas as réplicas recebem a requisição com o mesmo *timestamp* e conjunto de *nonces*, é possível implementar ações tipicamente não deterministas, como leitura do valor do *clock* ou geração de números randômicos, de forma determinista nas réplicas.

A Figura 15 apresenta as interações que ocorrem no sistema para a execução de uma requisição que deve ser ordenada, i.e., uma requisição que altera o estado das réplicas. Primeiramente, o cliente envia a requisição

assinada para todas as réplicas do sistema, através da classe *ServiceProxy* (método *invoke*). Após receber a requisição, as réplicas executam o protocolo de ordenação para definir uma ordem de entrega para a mesma (passo 2).

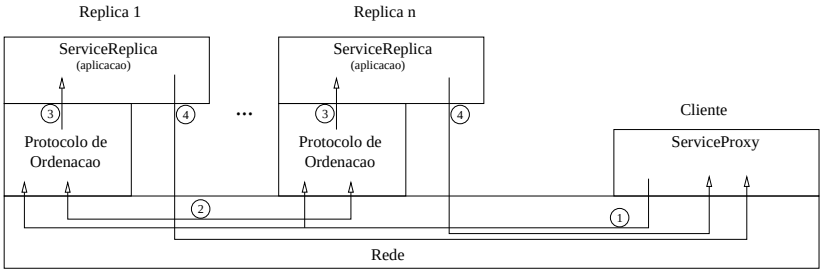


Figura 15 – Interações no BFT-SMART.

Quando as requisições anteriores já tiverem sido entregues em determinada réplica, a mesma executa o método *executeCommand* para entregar esta requisição para a aplicação (*ServiceReplica* - passo 3). Após executar a requisição, cada réplica envia a resposta para o cliente (passo 4), que recebe estas mensagens na classe *ServiceProxy*. Quando pelo menos $f + 1$ respostas iguais são recebidas, o protocolo termina e esta resposta é o resultado da execução do método *invoke*.

6.4 RECONFIGURANDO O BFT-SMART

Diversas maneiras de reconfigurar o conjunto de réplicas que implementam uma Máquina de Estados são discutidas por Lamport *et al.* (LAMPORT et al., 2010). Nestas abordagens, a própria Máquina de Estados é usada na definição da nova configuração do sistema, fazendo com que todas as réplicas concordem com a nova configuração (visão) de forma semelhante aos protocolos de *group membership* (CHOCKLER et al., 2001).

Seguindo esta técnica, a reconfiguração do BFT-SMART é bastante simples. Em termos gerais, o protocolo funciona da seguinte forma:

1. Primeiramente, algum processo envia uma requisição de reconfiguração do sistema, de forma idêntica a um cliente que acessa o sistema normalmente para execução de alguma operação pela Máquina de Estados.

2. Esta requisição é ordenada junto com as requisições dos clientes, i.e., a mesma é tratada como uma requisição normal pelo protocolo de or-

denação, que definirá a ordem de entrega para um lote de requisições que conterá tal requisição de reconfiguração.

3. Quando este lote de requisições for entregue em determinada réplica, a mesma executa todas as operações normais (requisições executadas pela aplicação) para então executar todas as operações de reconfiguração contidas neste lote. Note que um lote pode conter mais de um pedido de reconfiguração e todas as alterações no sistema, definidas pelas requisições deste lote, são processadas de uma só vez após a execução das requisições normais deste lote.

4. Uma nova configuração do sistema é definida e enviada para os clientes, bem como para os réplicas que estão entrando no sistema.

5. As réplicas que estão entrando no sistema atualizam seus estados a partir do estado das réplicas da configuração antiga, as quais fornecem seus estados obtidos até a execução da reconfiguração.

6. Por fim, todas as réplicas da nova configuração iniciam a execução da Máquina de Estados com o estado atual do sistema.

Como no BFT-SMART instâncias de consenso não são executadas em paralelo, não é necessário se preocupar com instâncias utilizando a configuração antiga, uma vez que a instância de consenso seguinte, que definirá a ordem de entrega para o lote de requisições seguinte, já é inicializada utilizando-se a nova configuração do sistema.

Estes procedimentos de reconfiguração podem ser entendidos da seguinte forma (LAMPORT et al., 2010). Considere uma sequência de Máquinas de Estados, onde a máquina anterior na sequência escolhe a configuração da máquina seguinte e é finalizada (parada). Então, a máquina seguinte é inicializada com esta configuração a partir do estado final da máquina anterior, e assim por diante.

Existem duas formas de reconfigurações suportadas pelo BFT-SMART: *(1)* uma onde a própria réplica (servidor) solicita sua entrada ou saída do sistema; e outra *(2)*, mais abrangente, onde uma terceira parte confiável (TTP - *trusted third party*) têm o poder de reconfigurar o sistema, requisitando entradas e saídas de réplicas e/ou solicitando alterações nos parâmetros de configuração do sistema.

6.4.1 Visões

Antes de discutir as formas de reconfiguração do BFT-SMART, vamos definir os componentes que formam as visões do sistema, uma vez que a

mesma é fundamental para o BFT-SMART ser reconfigurável, pois sempre que uma reconfiguração ocorre no sistema, uma nova visão mais atual é gerada para o mesmo. Esta nova visão reflete os pedidos de alterações que motivaram a execução da reconfiguração.

Sendo assim, cada visão v do sistema contém um identificador único ($v.id$), que nada mais é do que um contador que é incrementado na medida que reconfigurações são executadas. Desta forma, através de seus identificadores, podemos definir facilmente qual de duas visões é mais atual no sistema.

Além disso, cada visão contém os identificadores das réplicas (servidores) que fazem parte de tal visão, bem como o valor dos parâmetros reconfiguráveis no sistema. Atualmente, o único parâmetro reconfigurável no BFT-SMART, além do conjunto de servidores, é o limite f de falhas suportadas pelo sistema. Para cada visão v, o número de réplicas presentes nesta visão $v.n$ deve ser $v.n \geq 3v.f + 1$, onde $v.f$ representa o número de falhas de réplicas de v suportadas pelo sistema.

6.4.2 Reconfiguração a partir da própria Réplica

Este tipo de reconfiguração é mais restrita, onde uma determinada réplica (servidor) apenas é capaz de solicitar sua própria entrada ou saída do sistema. Deste modo, nenhum outro parâmetro do sistema pode ser alterado. A Figura 16 apresenta os métodos adicionados na API dos servidores (classe *ServiceReplica*), os quais permitem a solicitação de suas entradas (*join*) e/ou saídas (*leave*) do sistema.

```
//API do Servidor
public abstract class ServiceReplica ...{
   ...
   public void join();
   public void leave();
}
```

Figura 16 – Métodos adicionais para reconfiguração do BFT-SMART.

Na execução de qualquer um destes métodos, o servidor acessa o sistema como um cliente e envia a sua solicitação de reconfiguração, que é processada como descrito anteriormente. Após ordenada, esta requisição não é entregue para a aplicação, mas para um módulo de reconfiguração (*ReconfigurationManager*) onde uma nova configuração (visão) é gerada para o sistema. Por fim, uma resposta é enviada ao solicitante para informar a execução de sua requisição.

A Figura 17 apresenta as interações neste procedimento, onde os pas-

sos 3*b* e 4*b* representam a execução de uma requisição de reconfiguração, enquanto os passos 3*a* e 4*a* referem-se à execução de uma requisição normal. Através do protocolo de ordenação, estes passos são sincronizados em todas as réplicas presentes em determinada visão, i.e., as requisições são entregues para a aplicação ou para o módulo de reconfiguração de forma ordenada.

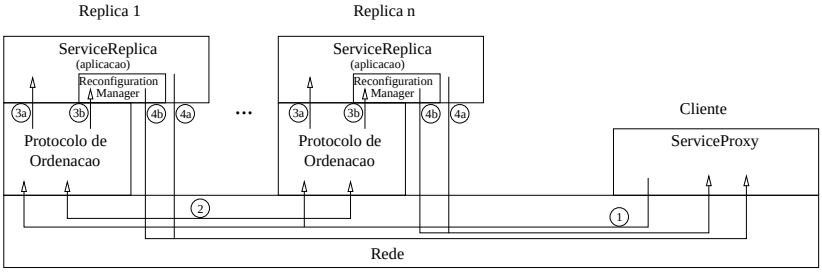


Figura 17 – Interações no BFT-SMART reconfigurável.

No caso de uma solicitação para entrada no sistema, a requisição contém o identificador e o endereço (IP e porta) do servidor solicitante, de modo que todos os servidores do sistema passam a conhecer este novo servidor. Já a resposta obtida desta solicitação contém a visão na qual o servidor entrou no sistema (nova visão gerada pelo *ReconfigurationManager*) e todos os demais dados necessários para a inicialização da réplica (estado e parâmetros de configuração do BFT-SMART). Além disso, as conexões são atualizadas em todos os servidores do sistema, de acordo com a nova visão do mesmo. Por fim, todos os servidores da nova visão passam a participar do sistema. Note que, todo este processamento é completamente transparente para o programador, que apenas deve invocar o método *join* na *ServiceReplica*.

6.4.3 Reconfiguração a partir da TTP

Este tipo de reconfiguração é mais abrangente, onde uma terceira parte confiável (TTP - *trusted third party*) é capaz de modificar tanto o grupo de réplicas que implementam a Máquina de Estados quanto as configurações do sistema, como por exemplo o limite f de faltas suportadas. A Figura 18 apresenta a API da TTP, onde podemos verificar quais são os métodos que devem ser usados para reconfigurações do sistema.

A TTP deve ser criada da mesma forma de um cliente normal (Seção 6.3.2). Posteriormente, para adicionar ou remover servidores, devemos utilizar os métodos *addServer* e *removeServer*, respectivamente. Para cada novo

servidor, é necessário informar seu identificador e seu endereço (IP e porta). Também é possível modificar o limite f de faltas suportadas pelo sistema através do método *setF*, onde esta alteração apenas é executada caso o número de réplicas na nova visão v continue obedecendo ao limite $v.n \geq 3v.f + 1$.

```
//API da TTP
public class TTP{
   ...
   public void addServer(int id, String ip, int port);
   public void removeServer(int id);
   public void setF(int f);
   public void executeUpdates();
   public void close();
}
```

Figura 18 – API da TTP.

Com o intuito de evitar que para cada alteração desejada seja necessário que o sistema execute uma nova reconfiguração, a TTP armazena todas as alterações solicitadas através destes métodos (*addServer*, *removeServer* e *setF*) e envia apenas uma única requisição de reconfiguração contendo todas estas alterações. Este processamento é executado através do método *executeUpdates*, onde a TTP acessa o sistema como um cliente (assim como descrito na seção anterior) para requisitar a reconfiguração do mesmo.

Quando a resposta é recebida na TTP, a mesma envia uma mensagem para os servidores que estão entrando no sistema, informando-os sobre a visão na qual tais servidores entraram no sistema e sobre os demais dados necessários para a inicialização destas réplicas (estado e parâmetros de configuração do BFT-SMART). Após isso, as réplicas atualizam suas conexões, de acordo com a nova visão do sistema. Por fim, todos os servidores da nova visão passam a participar do sistema. Note que, todo este processamento é completamente transparente para o programador, que apenas deve invocar o método *executeUpdates* da TTP.

Após a execução do método *executeUpdates*, a TTP fica pronta para ser usada novamente em outra reconfiguração, onde novas alterações serão executadas no sistema. No entanto, é possível finalizar a TTP através do método *close* que fecha todas as conexões da TTP com os servidores do sistema. De qualquer forma, posteriormente é possível criar outra TTP para uma nova reconfiguração do sistema.

6.4.4 Discussão

Esta seção apresenta algumas discussões sobre os protocolos do BFT-SMART reconfigurável.

6.4.4.1 Lidando com Reconfigurações

Da mesma forma dos protocolos apresentados no capítulo anterior, os clientes do BFT-SMART sempre devem acessar a configuração (visão) mais atual do sistema. Desta forma, um cliente c anexa em suas requisições o identificador da sua visão corrente e os servidores verificam (imediatamente antes da entrega e após a ordenação da requisição – passo 3a ou 3b da Figura 17) se tal cliente está utilizando a visão mais atual do sistema. Se este for o caso, a requisição é executada normalmente. Caso contrário, os servidores enviam uma resposta para c contendo a nova visão, e então, c envia novamente sua requisição considerando esta nova visão do sistema. Este processamento é completamente transparente para o programador, que apenas deve invocar as operações normalmente no *ServiceProxy* (Seção 6.3.2).

Note que, considerando v a visão atual do sistema, para uma requisição ser ordenada e entregue, a mesma deve atingir pelo menos um servidor correto de v, onde a autenticidade desta requisição é comprovada pela assinatura do cliente. Desta forma, um cliente que utiliza uma visão antiga w apenas terá sua requisição ordenada e obterá uma resposta contendo a visão atualizada v, caso $w \cap v$ contenha pelo menos um servidor correto. Para relaxar esta necessidade, as visões do sistema deveriam ser armazenadas também em algum lugar padrão, a partir do qual os clientes poderiam obtê-las, como discutido na seção seguinte.

6.4.4.2 Armazenamento das Visões

Atualmente, as visões do BFT-SMART são armazenadas apenas pelos próprios servidores que implementam o sistema. Desta forma, os clientes apenas atualizam suas visões nos casos onde suas requisições chegam em algum servidor correto da visão atual do sistema, como discutido na seção anterior.

No entanto, é factível que clientes permaneçam um longo período de tempo sem acessar o sistema, de forma que sua visão não contenha nenhum servidor da visão atual. Nestes casos, tais clientes não conseguem atualizar suas visões. Este mesmo problema ocorre com clientes que desejam acessar

o sistema somente após um conjunto reconfigurações levar o sistema para uma visão que não contém nenhum servidor da visão inicial. Note que este problema afeta apenas os clientes, uma vez que os servidores são sempre informados sobre as novas visões.

Para resolver este problema, é necessário que as visões sejam armazenadas em algum lugar padrão, a partir do qual qualquer cliente possa obtê-las. Neste sentido, é necessário que os servidores emitam certificados (conjunto de assinaturas) que atestem a autenticidade de uma visão. Para aumentar o grau de tolerância a faltas, as visão podem ser armazenadas em vários lugares ou em algum serviço tolerante a faltas bizantinas. Nesta abordagem, sempre que um cliente não tenha sua requisição atendida dentro de um determinado tempo, o mesmo deve verificar se está utilizando a visão atual do sistema.

6.5 CONSIDERAÇÕES FINAIS

Este capítulo apresentou nossos esforços na concretização de uma replicação Máquina de Estados dinâmica, onde o conjunto de réplicas, bem como o limite de faltas suportadas pelo sistema, pode ser reconfigurado em tempo de execução.

Seguimos uma das abordagens para reconfiguração de uma replicação Máquina de Estados apresentada por Lamport *et al.* (LAMPORT et al., 2010), onde adicionamos mecanismos que implementam esta abordagem no BFT-SMART. Além da reconfiguração propriamente dita, vários outros aspectos práticos devem ser considerados em uma implementação real, como a abertura e o término de conexões de forma segura (evitando que múltiplas conexões sejam estabelecidas para determinada réplica), que reflita a visão atual do sistema. Todos os códigos gerados podem ser obtidos na página do BFT-SMART (BESSANI et al., 2011).

7 CONCLUSÕES

O presente capítulo conclui esta tese. Primeiramente, uma visão geral sobre o trabalho é apresentada. Após isso, os objetivos desta tese são relembrados e as contribuições da mesma são abordadas. Por fim, algumas perspectivas de trabalhos futuros são expostas.

7.1 VISÃO GERAL DO TRABALHO

Este trabalho apresentou estudos sobre o desenvolvimento de protocolos para aplicações em sistemas distribuídos dinâmicos, que são caracterizados como sistemas onde tanto as topologias quanto o conjunto de participantes estão sujeitos a modificações arbitrárias, causadas principalmente pela entrada e saída voluntária de participantes no sistema ou por eventos como falhas.

Os protocolos apresentados nesta tese buscam propor soluções para diversos problemas identificados como peças fundamentais no desenvolvimento de aplicações distribuídas seguras e confiáveis. Todos estes protocolos foram projetados em conformidade com os atributos de segurança de funcionamento, i.e., mesmo que uma parte dos componentes presentes no sistema em um dado momento falhe de forma arbitrária (o tipo mais genérico de faltas), o mesmo continua funcionando segundo suas especificações.

O trabalho pode ser dividido em três partes. A primeira parte compreende os estudos acerca da possibilidade de solução do consenso bizantino em redes desconhecidas (BFT-CUP), onde os participantes do sistema obtêm diferentes visões parciais do mesmo. Este estudo adota o modelo de sistema dinâmico com chegadas finitas de participantes em cada execução dos protocolos. Neste sentido, mostramos quais são as condições necessárias e suficientes para resolver o BFT-CUP, considerando o grau de conhecimento que os participantes devem obter e o nível de sincronia que deve ser observado no sistema. Ainda nesta parte do trabalho, realizamos alguns experimentos com o objetivo de verificar se os valores teoricamente exigidos para a solução do BFT-CUP devem ser obrigatoriamente observados na prática.

A segunda parte deste trabalho propõe uma série de protocolos para quóruns bizantinos dinâmicos, os quais podem ser utilizados como memória compartilhada nestes ambientes. São apresentadas diversas formas de reconfiguração do conjunto de servidores que implementam o sistema de quóruns, as quais devem ser utilizadas de acordo com o nível de sincronia observado no sistema. Esta parte do trabalho também discute uma série de problemas que

são encontrados no gerenciamento de visões em qualquer sistema distribuído dinâmico, como as formas possíveis de autenticar visões e as maneiras de informar os clientes sobre as atualizações que acontecem na visão do sistema.

A última parte deste trabalho apresenta uma abordagem mais prática sobre sistemas distribuídos dinâmicos, apresentando e discutindo como suportar reconfigurações em uma replicação Máquina de Estados. Com este suporte, o sistema passa a fornecer interfaces que possibilitam sua reconfiguração através da atualização tanto do conjunto de servidores que implementam a Máquina de Estados quanto dos parâmetros de configuração da mesma.

7.2 REVISÃO DOS OBJETIVOS E CONTRIBUIÇÕES DA TESE

Os objetivos desta tese, enunciados na seção 1.2, são aqui relembrados, onde é indicado qual parte deste trabalho apresenta o estudo que visa cumprir cada objetivo. Além disso, esta tese apresentou uma série de contribuições, principalmente na área de tolerância a faltas bizantinas em sistemas distribuídos dinâmicos. Estas contribuições vão ao encontro do cumprimento dos objetivos estabelecidos e também são aqui abordadas.

1. **Analisar as implicações de segurança de funcionamento provenientes da execução de aplicações distribuídas em ambientes dinâmicos.**

   Em todos os protocolos apresentados nesta tese, são analisadas as questões de segurança envolvidas no funcionamento dos mesmos, abordando possíveis ataques que processos maliciosos podem executar contra o sistema. Desta forma, estas análises levantam uma série de problemas (ou vulnerabilidades) dos quais o sistema está sujeito. Estes problemas são resolvidos pelos protocolos propostos, sendo que são discutidas soluções alternativas para alguns deles.

2. **Propor algoritmos e protocolos que suportem e facilitem o desenvolvimento de aplicações distribuídas para ambientes distribuídos dinâmicos.**

   Todos os algoritmos e protocolos abordados nesta tese propuseram soluções para problemas fundamentais encontrados no desenvolvimento de aplicações distribuídas. Como estes algoritmos e protocolos foram projetados para ambientes distribuídos dinâmicos, os mesmos podem e devem ser usados como suporte no desenvolvimento de aplicações para estes ambientes.

3. **Propor algoritmos e protocolos para coordenação de participantes em sistemas distribuídos dinâmicos.**

O Capítulo 4 apresenta os estudos envolvendo o problema do consenso bizantino entre participantes desconhecidos, chamado BFT-CUP, onde são analisadas as condições necessárias e suficientes para resolver este problema em um modelo de sistema distribuído dinâmico com chegadas finitas de participantes em cada execução do protocolo. Além disso, também são apresentados alguns experimentos envolvendo os protocolos propostos. Em um outro cenário, a Máquina de Estados reconfigurável, apresentada no Capítulo 6, também pode ser usada para coordenação de participantes em sistemas distribuídos dinâmicos.

4. **Propor algoritmos e protocolos para armazenamento de dados em sistemas distribuídos dinâmicos (memória compartilhada).**

   O Capítulo 5 apresenta uma série de protocolos para reconfiguração de sistemas de quóruns bizantinos, possibilitando o emprego dos mesmos como alternativas para armazenamento de dados em ambientes distribuídos dinâmicos. Em um outro cenário, a Máquina de Estados reconfigurável, apresentada no Capítulo 6, também pode ser usada para armazenamento de dados em sistemas distribuídos dinâmicos.

5. **Avaliar os algoritmos e protocolos propostos através de simulações e/ou implementações de sistemas reais.**

   A maioria dos algoritmos e protocolos propostos foi avaliada. Os protocolos do BFT-CUP foram simulados nos mais diversos cenários de uma rede MANET, onde é possível verificar se as condições teoricamente necessárias para resolver o BFT-CUP realmente precisam ser verificadas na prática para que este problema admita solução. Além disso, projetamos e implementamos um protocolo de reconfiguração para Máquina de Estados, adicionando este suporte no BFT-SMART.

Uma parte das contribuições listadas acima foram publicadas (ALCHIERI et al., 2008a, 2008b, 2009a, 2009b, 2009c, 2011).

7.3 PERSPECTIVAS FUTURAS

Os estudos realizados nesta tese certamente não finalizam a discussão sobre tolerância a faltas e intrusões em sistemas distribuídos dinâmicos. Além do mais, as pesquisas nesta área são relativamente novas, quando comparadas com as pesquisas que consideram ambientes estáticos, e sendo este assunto muito amplo, dificilmente as opções de pesquisa que o envolvem se esgotarão.

Deste modo, alguns trabalhos relacionados com esta tese poderão ser explorados no futuro. O primeiro destes trabalhos é a realização de testes

mais amplos com o BFT-CUP, onde análises mais significativas a respeito destes protocolos poderiam ser apuradas. O Capítulo 4 apresenta um experimento realizado com o BFT-CUP em vários cenários de uma rede MANET, além de trazer uma análise analítica a respeito da utilização desta solução em uma rede VANET. Atualmente, estamos analisando o comportamento de uma rede VANET, formada pelos veículos da cidade de Porto - Portugal (Seção 4.8.2), através da variação de seus parâmetros como o alcance de transmissão (considerando a presença de obstáculos) e a densidade dos veículos, onde será possível verificar de uma forma mais precisa os cenários para os quais o BFT-CUP admite solução.

Nesta mesma linha, os protocolos do BFT-CUP poderiam ser explorados em outros cenários, como por exemplo em redes P2P. Além disso, outros experimentos com a presença de participantes maliciosos poderiam nos indicar quais deveriam ser os parâmetros de configuração do sistema para que o BFT-CUP admita solução nestes cenários, i.e., poderíamos prever quanto a mais de conectividade seria necessário quando os protocolos são executados na presença de participantes maliciosos. Note que nos experimentos apresentados neste texto, apesar do sistema ser configurado para tolerar um determinado número de faltas, não foram incluídos participantes maliciosos nas execuções dos protocolos.

Outro trabalho interessante seria a implementação dos protocolos que compõem o QUINCUNX, possibilitando a análise prática dos mesmos. De fato, neste trabalho propomos diferentes formas de reconfigurar um sistema de quóruns, mas não analisamos completamente os aspectos práticos envolvendo cada solução. Neste sentido, um trabalho interessante será a realização de uma análise prática acerca destas soluções para reconfiguração de sistemas de quóruns, onde será possível verificar as diferenças (principalmente relacionadas com o tempo necessário para a execução de uma reconfiguração) entre as mesmas.

Atualmente, nossa solução para replicação Máquina de Estados reconfigurável está sendo empregada no projeto *CloudFIT* (CLOUDFIT, 2011), onde as réplicas que implementam o sistema são substituídas periodicamente com o objetivo de: *(1)* fugir de ataques; *(2)* recuperação pró-ativa; *(3)* trazer o sistema para próximo dos clientes; entre outros. Neste sentido, além de uma aplicação prática para o nosso sistema, vários experimentos práticos estão sendo realizados com o mesmo.

Além das questões envolvendo a segurança de funcionamento dos protocolos, outro trabalho que não foi explorado nesta tese refere-se a escalabilidade dos mesmos em relação a quantidade de reconfigurações. Neste sentido, seria interessante verificar qual o comportamento destes protocolos de acordo com a taxa de *churn* e com o período para reconfigurações do sistema.

Por fim, como propomos uma série de protocolos fundamentais para o desenvolvimento de aplicações distribuídas em ambientes dinâmicos, uma perspectiva futura natural para este trabalho é o projeto e concretização destas aplicações, que deverão operar sobre os protocolos propostos. Neste sentido, dentre uma série de aplicações interessantes, podemos citar uma aplicação que têm como política de desempenho a localização dos servidores, onde os mesmos devem estar localizados o mais próximo possível dos clientes ativos (clientes que estão acessando o sistema) em determinado momento. Considerando que a maioria dos clientes ativos esteja localizada em regiões que se encontram no período diurno, esta aplicação percorreria o globo terrestre acompanhando o horário solar.

REFERÊNCIAS BIBLIOGRÁFICAS

AGUILERA, M. A pleasant stroll through the land of infinitely many creatures. *SIGACT News*, v. 35, n. 2, p. 36–59, 2004.

AGUILERA, M. K.; KEIDAR, I.; MALKHI, D.; SHRAER, A. Dynamic atomic storage without consensus. In: *Proceedings of the 28th ACM Symposium on Principles of distributed computing*. [S.l.: s.n.], 2009.

AGUILERA, M. K.; KEIDAR, I.; MALKHI, D.; SHRAER, A. Dynamic atomic storage without consensus. *JACM*, ACM, New York, NY, USA, v. 58, p. 7:1–7:32, 2011. ISSN 0004-5411.

ALCHIERI, E. A.; BESSANI, A. N.; FRAGA, J. S.; GREVE, F. Byzantine consensus with unknown participants. In: *OPODIS '08: Proceedings of the 12th International Conference on Principles of Distributed Systems*. Berlin, Heidelberg: Springer-Verlag, 2008. p. 22–40. ISBN 978-3-540-92220-9.

ALCHIERI, E. A.; BESSANI, A. N.; FRAGA, J. S.; GREVE, F. Consenso bizantino entre participantes desconhecidos. In: *Anais do IX Workshop de Teste e Tolerância a Falhas- WTF 2008*. Rio de Janeiro, RJ, Brasil: [s.n.], 2008. p. 169–182.

ALCHIERI, E. A.; BESSANI, A. N.; PEREIRA, F. C.; FRAGA, J. S. Proactive byzantine quorum systems. In: *Proceedings of the Confederated International Conferences, CoopIS, DOA, IS, and ODBASE 2009 on On the Move to Meaningful Internet Systems: Part I*. Berlin, Heidelberg: Springer-Verlag, 2009. (OTM '09), p. 708–725.

ALCHIERI, E. A. P.; BESSANI, A. N.; PEREIRA, F.; FRAGA, J. da S. Sistemas de quóruns bizantinos pró-ativos. In: *Anais do 27º Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos- SBRC 2009*. Recife, PE, Brasil: [s.n.], 2009.

ALCHIERI, E. A. P.; BESSANI, A. N.; PEREIRA, F.; FRAGA, J. da S. Sistemas de quóruns bizantinos pró-ativos. *Revista Brasileira de Redes de Computadores e Sistemas Distribuídos*, v. 2, p. 21–34, 2009.

ALCHIERI, E. A. P.; TOMELIN, L.; BESSANI, A. N.; FRAGA, J. da S. Aspectos práticos sobre o consenso bizantino entre participantes desconhecidos. In: *Anais do XII Workshop de Teste e Tolerância a Falhas- WTF 2011*. [S.l.: s.n.], 2011.

ALVISI, L.; PIERCE, E. T.; MALKHI, D.; REITER, M. K.; WRIGHT, R. N. Dynamic byzantine quorum systems. In: *DSN '00: Proceedings of the 2000 International Conference on Dependable Systems and Networks (formerly FTCS-30 and DCCA-8)*. Washington, DC, USA: IEEE Computer Society, 2000. p. 283. ISBN 0-7695-0707-7.

ASPNES, J. Randomized protocols for asynchronous consensus. *Distributed Computing*, Springer-Verlag, v. 16, n. 2-3, p. 165–175, 2003.

ATTIYA, H.; WELCH, J. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. 2nd. ed. [S.l.]: Wiley-Interscience, 2004. (Wiley Series on Parallel and Distributed Computing).

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

AWERBUCH, B.; HOLMER, D.; NITA-ROTARU, C.; RUBENS, H. An on-demand secure routing protocol resilient to byzantine failures. In: *WiSE '02: Proceedings of the 1st ACM workshop on Wireless security*. New York, NY, USA: ACM, 2002. p. 21–30. ISBN 1-58113-585-8.

BADACHE, N.; HURFIN, M.; MACEDO, R. Solving the consensus problem in a mobile environment. In: *Proc. of the IEEE International Performance, Computing, and Communications Conference – IPCCC'99*. [S.l.]: IEEE Press, 1999. p. 29–35.

BALDONI, R.; BERTIER, M.; RAYNAL, M.; TUCCI-PIERGIOVANNI, S. Looking for a definition of dynamic distributed systems. In: LNCS, S. V. (Ed.). *Proceeding of 9th International Conference on Parallel Computing Technologies - PaCT 2007*. Toulouse, France: [s.n.], 2007.

BAR-JOSEPH, Z.; KEIDAR, I.; LYNCH, N. Early-delivery dynamic atomic broadcast. In: *Proceedings of the 16th International Symposium on DIStributed Computing - DISC 2002*. Toulouse, France: [s.n.], 2002.

BARR, R.; HAAS, Z. J.; RENESSE, R. van. Jist: Embedding simulation time into a virtual machine. In: *Proceedings o EuroSim Congress on Modelling and Simulation*. [S.l.: s.n.], 2004.

BARR, R.; HAAS, Z. J.; RENESSE, R. van. Handbook on theoretical and algorithmic aspects of sensor, ad hoc wireless, and peer-to-peer networks. In: ____. [S.l.]: CRC Press, 2005. cap. Scalable Wireless Ad hoc Network Simulation, p. 297–311.

BAZZI, R. A.; DING, Y. Non-skipping timestamps for Byzantine data storage systems. In: *Proc. of 18th Int. Symposium on Distributed Computing, DISC 2004*. [S.l.: s.n.], 2004. (LNCS, v. 3274), p. 405–419.

BELLARE, M.; ROGAWAY, P. Random oracles are practical: A paradigm for designing efficient protocols. In: *Proc. of the 1st ACM Conference on Computer and Communications Security*. [S.l.: s.n.], 1993. p. 62–73.

BEN-OR, M. Another advantage of free choice: Completely asynchronous agreement protocols (extended abstract). In: *Proceedings of the 2rd Annual ACM Symposium on Principles of Distributed Computing*. [S.l.: s.n.], 1983. p. 27–30.

BESSANI, A. N.; ALCHIERI, E. A. P.; CORREIA, M.; FRAGA, J. da S. Depspace: a byzantine fault-tolerant coordination service. *SIGOPS Oper. Syst. Rev.*, ACM, New York, NY, USA, v. 42, n. 4, p. 163–176, 2008. ISSN 0163-5980.

BESSANI, A. N.; ALCHIERI, E. A. P.; SOUZA, P. *BFT-SMART: High-performance Byzantine-Fault-Tolerant State Machine Replication*. 2011. Http://code.google.com/p/bft-smart/.

BETTSTETTER, C. On the minimum node degree and connectivity of a wireless multihop network. In: *Proceedings of the 3rd ACM international symposium on Mobile ad hoc networking & computing - MobiHoc'02*. [S.l.: s.n.], 2002.

BRACHA, G. An asynchronous $\lfloor (n-1)/3 \rfloor$-resilient consensus protocol. In: *Proceedings of the 3rd ACM symposium on Principles of Distributed Computing*. [S.l.: s.n.], 1984. p. 154–162. ISBN 0-89791-143-1.

BRACHA, G.; TOUEG, S. Asynchronous consensus and broadcast protocols. *Journal of ACM*, v. 32, n. 4, p. 824–840, 1985.

CAO, J.; RAYNAL, M.; TRAVERS, C.; WU, W. The eventual leadership in dynamic mobile networking environments. In: *PRDC '07: Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing*. Washington, DC, USA: IEEE Computer Society, 2007. p. 123–130. ISBN 0-7695-3054-0.

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

CAVIN, D.; SASSON, Y.; SCHIPER, A. *Reaching Agreement with Unknown Participants in Mobile Self-Organized Networks in Spite of Process Crashes*. [S.l.], jun. 2005.

CAVIN, D.; SASSON, Y.; SCHIPER, A. Consensus with unknown participants or fundamental self-organization. In: *Proc. of the 3rd Int. Conf. on Ad hoc Networks and Wireless - ADHOC-NOW 2004*. [S.l.: s.n.], 2004. p. 135–148.

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

CHARRON-BOST, B.; SCHIPER, A. *Uniform consensus is harder than consensus (extended abstract)*. Lausanne, Switzerland, maio 2000.

CHOCKLER, G. V.; KEIDAR, I.; VITENBERG, R. Group communication specifications: A comprehesive study. *ACM Computing Surveys*, v. 33, n. 4, p. 427–469, dez. 2001.

CLEMENT, A.; WONG, E.; ALVISI, L.; DAHLIN, M.; MARCHETTI, M. Making byzantine fault tolerant systems tolerate byzantine faults. In: *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*. [S.l.]: USENIX Association, 2009. p. 153–168.

CLOUDFIT. *CloudFIT: Byzantine Fault Tolerance for the Cloud*. 2011. Http://cloudfit.di.fc.ul.pt.

CORREIA, M.; NEVES, N. F.; VERíSSIMO, P. From consensus to atomic broadcast: Time-free Byzantine-resistant protocols without signatures. *The Computer Journal*, v. 49, n. 1, jan. 2006.

COSTA, V. F.; GREVE, F. G. P.; ; TIXEUIL, S. Consenso ft-cup em redes manets: Uma abordagem prática. In: *Anais do 27º Simpósio Brasileiro de Redes de Computadores - SBRC 2009*. [S.l.: s.n.], 2009.

COSTA, V. F.; GREVE, F. G. P. Aspectos práticos da realização do consenso ft-cup em redes móveis ad-hoc. In: *Anais VIII Workshop de Teste e Tolerância a Falhas- WTF 2007*. Belém, PA, Brasil: [s.n.], 2007.

CRISTIAN, F. Agreeing on who is present and who is absent in a synchronous distributed system. In: *Proceedings of the 18th International Conference on Fault-Tolerant Computing*. [S.l.: s.n.], 1988.

CRISTIAN, F. Reaching agreement on processor group membership in synchronous distributed systems. *Distributed Computing*, v. 4, n. 4, p. 175–187, abr. 1991.

CURINO, C.; GIANI, M.; GIORGETTA, M.; GIUSTI, A.; MURPHY, A. L.; PICCO, G. P. Tinylime: Bridging mobile and sensor networks through middleware. In: *PERCOM '05: Proceedings of the Third IEEE International Conference on Pervasive Computing and Communications*. Washington, DC, USA: IEEE Computer Society, 2005. p. 61–72. ISBN 0-7695-2299-8.

DESMEDT, Y.; FRANKEL, Y. Threshold criptosystems. In: *Proceedings of the 9th Annual International Cryptology Conference on Advances in Cryptology - CRYPTO'89*. [S.l.]: Springer-Verlag, 1990. p. 307–315.

DIERKS, T.; ALLEN, C. *The TLS Protocol Version 1.0 (RFC 2246)*. jan. 1999. IETF Request For Comments.

DINGER, J.; HARTENSTEIN, H. Defending the sybil attack in p2p networks: Taxonomy, challenges, and a proposal for self-registration. In: *ARES '06: Proceedings of the First International Conference on Availability, Reliability and Security*. Washington, DC, USA: IEEE Computer Society, 2006. p. 756–763. ISBN 0-7695-2567-9.

DJENOURI, D.; KHELLADI, L.; BADACHE, A. A Survey of Security Issues in Mobile Ad Hoc and Sensor Networks. *Communications Surveys & Tutorials, IEEE*, v. 7, n. 4, p. 2–28, 2005.

DOLEV, D. The Byzantine generals strike again. *Journal of Algorithms*, n. 3, p. 14–30, 1982.

DOUCEUR, J. The sybil attack. In: *Proceedings of the 1st International Workshop on Peer-to-Peer Systems*. [S.l.: s.n.], 2002.

DWORK, C.; LYNCH, N. A.; STOCKMEYER, L. Consensus in the presence of partial synchrony. *Journal of ACM*, v. 35, n. 2, p. 288–322, abr. 1988.

FERREIRA, M.; aO, H. C.; FERNANDES, R.; TONGUZ, O. K. Stereoscopic aerial photography: an alternative to model-based urban mobility approaches. In: *Proceedings of the sixth ACM international workshop on VehiculAr InterNETworking - VANET'09*. [S.l.: s.n.], 2009.

FERREIRA, M.; FERNANDES, R.; aO, H. C.; VIRIYASITAVAT, W.; TONGUZ, O. K. Self-organized traffic control. In: *Proceedings of the seventh ACM international workshop on VehiculAr InterNETworking - VANET'10*. [S.l.: s.n.], 2010.

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

FOSTER, I. What is the grid? a three point checklist. *Grid Today*, v. 1, n. 6, p. 22–25, 2002.

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

FRIEDMAN, R.; MOSTEFAOUI, A.; RAYNAL, M. Simple and efficient oracle-based consensus protocols for asynchronous Byzantine systems. *IEEE Transactions on Dependable and Secure Computing*, v. 2, n. 1, p. 46–56, 2005.

FRIEDMAN, R.; RAYNAL, M.; TRAVERS, C. *Two Abstractions for Implementing Atomic Objects in Dynamic Systems*. [S.l.], jun. 2005.

FRIEDMAN, R.; RAYNAL, M.; TRAVERS, C. Two abstractions for implementing atomic objects in dynamic systems. In: *PODC '05: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*. New York, NY, USA: ACM, 2005. p. 354–354.

GELERNTER, D. Generative communication in Linda. *ACM Transactions on Programing Languages and Systems*, v. 7, n. 1, p. 80–112, jan. 1985.

GIFFORD, D. Weighted voting for replicated data. In: *Proc. of the 7th ACM Symposium on Operating Systems Principles*. [S.l.: s.n.], 1979. p. 150–162.

GILBERT, S.; LYNCH, N.; SHVARTSMAN, A. Rambo ii: Rapidly reconfigurable atomic memory for dynamic networks. *Proceedings of International Conference on Dependable Systems and Networks - DSN'03*, p. 259–268, June 2003.

GODFREY, P. B.; SHENKER, S.; STOICA, I. Minimizing churn in distributed systems. *SIGCOMM Comput. Commun. Rev.*, ACM Press, New York, NY, USA, v. 36, n. 4, p. 147–158, 2006. ISSN 0146-4833.

GREVE, F.; ARANTES, L.; SENS, P. What model and what conditions to implement unreliable failure detectors in dynamic networks? In: *Proceedings of the 3rd International Workshop on Theoretical Aspects of Dynamic Distributed Systems*. New York, NY, USA: ACM, 2011. (TADDS '11), p. 13–17. ISBN 978-1-4503-0946-2.

GREVE, F.; SENS, P.; ARANTES, L.; SIMON, V. A failure detector for wireless networks with unknown membership. In: *Proceedings of the 17th international conference on Parallel processing - Volume Part II*. Berlin, Heidelberg: Springer-Verlag, 2011. (Euro-Par'11), p. 27–38. ISBN 978-3-642-23396-8.

GREVE, F.; TIXEUIL, S. Conditions for the solvability of fault-tolerant consensus in asynchronous unknown networks: Invited paper. In: *III ACM SIGACT-SIGOPS International Workshop on Reliability, Availability, and Security. Co-located with the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*. [S.l.: s.n.], 2010.

GREVE, F. G. P.; TIXEUIL, S. Knowledge connectivity vs. synchrony requirements for fault-tolerant agreement in unknown networks. In: *Proc. of the Int. Conf. on Dependable Systems and Networks - DSN 2007*. [S.l.: s.n.], 2007. p. 82–91.

GUERRAOUI, R. Indulgent algorithms (preliminary version). In: *Proceedings of the 19^o annual ACM symposium on Principles of distributed computing*. New York, NY, USA: ACM, 2000. (PODC '00), p. 289–297.

GUERRAOUI, R.; RODRIGUES, L. *Lecture Notes in Distributed Computing (Preliminary Draft)*. Berlin: Springer-Verlag, 2003.

HADZILACOS, V.; TOUEG, S. *A Modular Approach to the Specification and Implementation of Fault-Tolerant Broadcasts*. New York - USA, maio 1994.

HERLIHY, M. Wait-free synchronization. *ACM Transactions on Programing Languages and Systems*, v. 13, n. 1, p. 124–149, jan. 1991.

ILYAS, M.; DORF, R. C. (Ed.). *The Handbook of Ad Hoc Wireless Networks*. Boca Raton, FL, USA: CRC Press, Inc., 2003. ISBN 0-8493-1332-5.

KEIDAR, I.; SUSSMAN, J.; MARZULLO, K.; DOLEV, D. Moshe: A group membership service for wans. *ACM Trans. Comput. Syst.*, ACM, v. 20, p. 191–238, 2002. ISSN 0734-2071.

KONG, L.; MANOHAR, D. J.; AHAMAD, M.; SUBBIAH, A.; SUN, M.; BLOUGH, D. M. Agile store: Experience with quorum-based data replication techniques for adaptive byzantine fault tolerance. In: *SRDS '05: Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems*. Washington, DC, USA: IEEE Computer Society, 2005. p. 143–154. ISBN 0-7695-2463-X.

KONG, L.; SUBBIAH, A.; AHAMAD, M.; BLOUGH, D. M. A reconfigurable byzantine quorum approach for the agile store. In: *SRDS '03: Proceedings of the 22th IEEE Symposium on Reliable Distributed Systems*. Los Alamitos, CA, USA: IEEE Computer Society, 2003.

KOTZANIKOLAOU, P.; MAVROPODI, R.; DOULIGERIS, C. Secure multipath routing for mobile ad hoc networks. *Wireless On-demand Network Systems and Services, 2005. WONS 2005. Second Annual Conference on*, p. 89–96, 2005.

LAMPORT, L.; SHOSTAK, R.; PEASE, M. The Byzantine generals problem. *ACM Transactions on Programing Languages and Systems*, v. 4, n. 3, p. 382–401, jul. 1982.

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

LAMPORT, L. On interprocess communication (part II). *Distributed Computing*, v. 1, n. 1, p. 203–213, jan. 1986.

LAMPORT, L. The part-time parliament. *ACM Transactions Computer Systems*, v. 16, n. 2, p. 133–169, maio 1998. ISSN 0734-2071.

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

LAMPORT, L.; MALKHI, D.; ZHOU, L. Reconfiguring a state machine. *SIGACT News*, ACM, v. 41, p. 63–73, 2010.

LAMPSON, B. The abcd's of paxos. In: *Proceedings of the 20th annual ACM Symposium on Principles of Distributed Computing*. [S.l.: s.n.], 2001. p. 13. ISBN 1-58113-383-9.

LIMA, M. de; GREVE, F. Protocolo Assíncrono para Detecção de Falhas Bizantinas em Sistemas Distribuídos Dinâmicos. In: *27º Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*. [S.l.: s.n.], 2009.

LIMA, M. S. de; GREVE, F.; ARANTES, L.; SENS, P. The time-free approach to byzantine failure detection in dynamic networks. *Dependable Systems and Networks Workshops (Workshop on Recent Advances in Intrusion-Tolerant Systems)*, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 3–8, 2011.

LISKOV, B.; RODRIGUES, R. S. M. Tolerating byzantine faulty clients in a quorum system. In: *The 26th IEEE International Conference on Distributed Computing Systems - ICDCS 2006*. [S.l.: s.n.], 2006.

LORCH, J. R.; ADYA, A.; BOLOSKY, W. J.; CHAIKEN, R.; DOUCEUR, J. R.; HOWELL, J. The smart way to migrate replicated stateful services. In: *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on*

Computer Systems 2006. New York, NY, USA: ACM, 2006. (EuroSys '06), p. 103–115. ISBN 1-59593-322-0.

LOUREIRO, A. A.; NOGUEIRA, J. M.; RUIZ, L. B.; MINI, R. A. de F.; NAKAMURA, E. F.; FIGUEIREDO, C. M. S. Redes de sensores sem fio. In: *Anais do 21o. Simpósio Brasileiro de Redes de Computadores - SBRC 2003 (Minicurso)*. Natal, RN, Brasil: [s.n.], 2003.

LYNCH, N.; SHVARTSMAN, A. A. Rambo: A reconfigurable atomic memory service for dynamic networks. In: *16th International Symposium on Distributed Computing - DISC*. [S.l.: s.n.], 2002. p. 173–190.

LYNCH, N. A. *Distributed Algorithms*. [S.l.]: Morgan Kauffman, 1996.

MALKHI, D.; REITER, M. Byzantine quorum systems. *Distributed Computing*, v. 11, n. 4, p. 203–213, out. 1998.

MALKHI, D.; REITER, M. Secure and scalable replication in Phalanx. In: *Proc. of 17th Symposium on Reliable Distributed Systems*. [S.l.: s.n.], 1998. p. 51–60.

MARTIN, J.-P.; ALVISI, L. A framework for dynamic byzantine storage. In: *DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks*. Washington, DC, USA: IEEE Computer Society, 2004. p. 325. ISBN 0-7695-2052-9.

MARTIN, J.-P.; ALVISI, L. *A Framework for Dynamic Byzantine Storage*. [S.l.], 2004.

MARTIN, J.-P.; ALVISI, L. Fast Byzantine consensus. *IEEE Transactions on Dependable and Secure Computing*, v. 3, n. 3, p. 202–215, 2006.

MERRITT, M.; TAUBENFELD, G. Computing with infinitely many processes: under assumptions on concurrency and participation. In: *Proceedings of the 14th International Conference on Distributed Computing*. [S.l.]: Springer-Verlag, 2000. p. 164–178.

MOSTéFAOUI, A.; RAYNAL, M.; TRAVERS, C.; PATTERSON, S.; AGRAWAL, D.; ABBADI, A. E. From static distributed systems to dynamic systems. In: *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems - SRDS 2005*. Washington, DC, USA: IEEE Computer Society, 2005. p. 109–118. ISBN 0-7695-2463-X.

MURPHY, A.; PICCO, G.; ROMAN, G.-C. LIME: A middleware for physical and logical mobility. In: *Proc. of the 21st Int. Conference on Distributed Computing Systems, ICDCS'01*. [S.l.: s.n.], 2001. p. 524–533.

PAPADIMITRATOS, P.; HAAS, Z. Secure routing for mobile ad hoc networks. In: *Proceedings of SCS Communication Networks and Distributed Systems Modeling and Simulation Conference (CNDS 2002)*. [S.l.: s.n.], 2002.

PIERGIOVANNI, S. T. *Concurrent Connectivity Maintenance with Infinitely Many Processes*. Tese (Doutorado) — MIDLAB - Università di Roma "La Sapienza", November 2005. <http://www.dis.uniroma1.it/ midlab>.

RAYNAL, M. A short introduction to failure detectors for asynchronous distributed systems. *SIGACT News*, ACM, New York, NY, USA, v. 36, p. 53–70, 2005. ISSN 0163-5700.

RICCIARDI, A. M.; BIRMAN, K. P. Using process groups to implement failure detection in asynchronous environments. In: *ACM Symposium on Principles of Distributed Computing*. Montreal - Quebec - Canada: [s.n.], 1991. p. 341–353.

RIVEST, R. L.; SHAMIR, A.; ADLEMAN, L. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, v. 21, n. 2, p. 120–126, 1978. ISSN 0001-0782.

RODRIGUES, R.; LISKOV, B. *Rosebud: A Scalable Byzantine-Fault-Tolerant Storage Architecture*. [S.l.], 2004.

RODRIGUES, R.; LISKOV, B.; SHRIRA, L. The design of a robust peer-to-peer system. In: *EW10: Proceedings of the 10th workshop on ACM SIGOPS European workshop*. New York, NY, USA: ACM, 2002. p. 117–124.

SANTI, P. Topology control in wireless ad hoc and sensor networks. *ACM Comput. Surv.*, ACM, v. 37, p. 164–194, 2005. ISSN 0360-0300.

SCHNEIDER, F. B. Implementing fault-tolerant service using the state machine aproach: A tutorial. *ACM Computing Surveys*, v. 22, n. 4, p. 299–319, dez. 1990.

SEBA, H.; BADACHE, N.; BOUABDALLAH, A. Solving the consensus problem in a dynamic group: An approach suitable for a mobile environment. In: *ISCC '02: Proceedings of the Seventh International Symposium on Computers and Communications (ISCC'02)*. Washington, DC, USA: IEEE Computer Society, 2002. p. 327. ISBN 0-7695-1671-8.

SENS, P.; GREVE, F.; ARANTES, L.; BOUILLAGUET, M.; SIMON, V. Um Detector de Falhas Assíncrono para Rede Móveis e Auto-Organizáveis.

In: *26º Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*. [S.l.: s.n.], 2008.

SHOUP, V. Practical threshold signatures. In: *Advances in Cryptology: EUROCRYPT 2000, Lecture Notes in Computer Science*. [S.l.]: Springer-Verlag, 2000. v. 1807, p. 207–222.

STEINMETZ, R.; WEHRLE, K. *Peer-to-Peer Systems and Applications*. [S.l.]: Springer-Verlag, 2005. ISBN 3-540-29192-x.

STUTZBACH, D.; REJAIE, R. Understanding Churn in Peer-to-Peer Networks. In: *IMC '06: Proceedings of the 6th ACM SIGCOMM on Internet measurement*. New York, NY, USA: ACM Press, 2006. p. 189–202. ISBN 1-59593-561-4.

TOMELIN, L.; ALCHIERI, E. A. P.; BESSANI, A. N.; FRAGA, J. da S. Consenso bizantino em manets. In: *Anais do XXIV Congresso Regional de Iniciação Científica e Tecnológica em Engenharia - CRICTE 2010*. [S.l.: s.n.], 20010.

TOUEG, S. Randomized Byzantine Agreements. In: *Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing*. [S.l.: s.n.], 1984. p. 163–178.

VERíSSIMO, P.; NEVES, N. F.; CORREIA, M. P. Intrusion-tolerant architectures: Concepts and design. In: *Architecting Dependable Systems*. [S.l.]: Springer-Verlag, 2003, (Lecture Notes in Computer Science, v. 2677).

VORA, A.; NESTERENKO, M.; TIXEUIL, S.; DELAET, S. *Universe Detectors for Sybil Defense in Ad Hoc Wireless Networks*. [S.l.], maio 2008.

WONG, T. M.; WANG, C.; WING, J. M. Verifiable secret redistribution for archive systems. In: *Proceedings of the 1th International IEEE Security in Storage Workshop*. [S.l.: s.n.], 2002.

ZHAO, F.; GUIBAS, L. *Wireless Sensor Networks: An Information Processing Approach*. [S.l.]: Morgan Kaufmann, 2004. ISBN 1-55860-914-8.

ZHOU, L.; SCHNEIDER, F.; RENNESSE, R. V. COCA: A secure distributed online certification authority. *ACM Transactions on Computer Systems*, v. 20, n. 4, p. 329–368, nov. 2002.

ZIELINSKI, P. *Paxos at War*. Cambridge, UK, jun. 2004.