UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE AUTOMAÇÃO E SISTEMAS

Davi da Silva Böger

SEGMENTAÇÃO DE *OVERLAYS* PAR A PAR COMO SUPORTE PARA MEMÓRIAS TOLERANTES A INTRUSÕES

Florianópolis

2012

Davi da Silva Böger

SEGMENTAÇÃO DE *OVERLAYS* PAR A PAR COMO SUPORTE PARA MEMÓRIAS TOLERANTES A INTRUSÕES

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

Florianópolis

2012

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

A ficha catalográfica é confeccionada pela Biblioteca Central.

Tamanho: 7cm x 12 cm

Fonte: Times New Roman 9,5

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

Davi da Silva Böger

SEGMENTAÇÃO DE *OVERLAYS* PAR A PAR COMO SUPORTE PARA MEMÓRIAS TOLERANTES A INTRUSÕES

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

Florianópolis, 07 de dezembro 2012.

Prof. Chefe, Dr. Jomi Fred Hübner
Coordenador

Prof. Dr. Joni da Silva Fraga
Orientador

Banca Examinadora:

Prof. Dr. Joni da Silva Fraga
Presidente

Prof. Dr. Jean-Marie Farines

Prof. Dr. Eduardo Adilio Pelinson Alchieri

Profa. Dra. Fabíola Gonçalves Greve

Prof. Dr. André Luiz Moura dos Santos

À minha pequenina Alice

AGRADECIMENTOS

Agradeço, primeiro, à minha família por todo o suporte e carinho recebidos durante toda a minha vida. Em especial, agradeço à minha namorada Celina pelo incentivo (e empurrões) nas horas de desânimo (e preguiça). Não teria conseguido sem ela.

Agradeço ao meu orientador, Professor Joni da Silva Fraga, pela excelente orientação, tanto nas reuniões quanto nas revisões dos textos. Agradeço também pela grande tolerância com os meus atrasos e lentidão em geral. Ao Eduardo Alchieri, agradeço pela enorme contribuição dada nas reuniões em que ele participou, na fase intermediária da elaboração desta pesquisa. Essas duas cabeças formam um grande time, com o qual tive a oportunidade de aprender.

Ao pessoal do STCFed, muito obrigado pelo apoio nas hora de aperto. Em especial, eu agradeço à Michelle por ter feito as coisas acontecerem no projeto. Apesar do atraso, eu agradeço ainda à Michelle por ter me ensinado a pesquisar e a escrever. Ela foi minha primeira orientadora de fato e na ocasião não lembrei de prestar os devidos agradecimentos.

Aos professores do DAS, em especial aos coordenadores e ao colegiado do PPGEAS, agradeço que tenham permitido prorrogar a minha defesa por todo esse tempo.

Agradeço ao CNPq e CAPES pelo apoio financeiro recebido durante a realização do mestrado.

Não é fácil dar banho num beija-flor.

Albran Kehlog Albran

RESUMO

As redes par a par (*peer-to-peer*, P2P) formam uma arquitetura de sistemas distribuídos que apresenta características de escalabilidade, abertura e dinamismo. Essas redes P2P foram inicialmente popularizadas por aplicações de compartilhamento de arquivos, porém hoje suas características as tornaram a base para construção de aplicações que necessitam de larga escala.

Apesar das vantagens das redes P2P, sua grande abertura e dinamismo trazem algumas dificuldades para a construção de certos tipos de aplicações. Entre os principais desafios estão a dificuldade em manter a consistência das informações com a possibilidade de entrada e saída de nós durante a execução e a necessidade de tolerar a participação de nós maliciosos que tem por objetivo corromper o sistema e impedir seu funcionamento. Esses desafios fizeram com que a maioria das aplicações sobre P2P sejam aplicações de armazenamento de informações que sofrem pouca ou nenhuma alteração durante a execução e que são autoverificáveis, isto é, é possível identificar modificações maliciosas ou acidentais pela análise do próprio conteúdo.

Dentro desse contexto, a proposta desta dissertação é a especificação de uma infraestrutura para a construção de aplicações arbitrárias, por meio de uma abstração de memória distribuída compartilhada, que tolere a participação de um número de nós maliciosos.

A ideia central consiste em aplicar técnicas de Replicação Máquina de Estados (RME) sobre a rede P2P. No entanto, RME apresenta problemas de escala pois o número de mensagens trocadas para coordenar as réplicas é de ordem quadrática. Assim sendo, a proposta é dividir a rede P2P em conjuntos de nós com tamanho limitado, denominados de segmentos, de forma a garantir o desempenho dos protocolos RME. Segmentos são dinâmicos, ou seja, podem aumentar ou diminuir à medida que nós entram e saem do sistema, porém a infraestrutura garante, por meio da união ou divisão de segmentos, que o tamanho permanece dentro dos limites estabelecidos.

O sistema foi elaborado como uma pilha de camadas com funcionalidades descritas na forma de operações e propriedades. As operações da segmentação foram implementadas por algoritmos em pseudocódigo, cujo funcionamento correto foi demonstrado em provas de lemas e teoremas. Uma análise crítica dos algoritmos esclareceu limitações e levantou os custos dos mesmos. A fim de demonstrar a expressividade da infraestrutura proposta, um espaço de tuplas foi construído utilizando as operações implementadas.

Palavras-chave: Redes Par a Par. Sistemas Distribuídos Dinâmicos. Segmentação. Memória Compartilhada. Tolerância a Intrusões.

ABSTRACT

Peer-to-peer (P2P) networks form a distributed system architecture that feature good scalability, openness and dynamism. Such networks were first made popular by file-sharing applications, although nowadays these features bacame the basis for the construction of applications that require scalability.
Even though P2P networks have some advantages, their openness and dynamism give raise to some difficulties in the construction of certain types of application. Among the most important challenges are the trouble to maintain consistency in face of constant nodes joining and leaving the system, and the need to tolerate the participation of malicious nodes whose purpose is to disrupt the system and to prevent its functioning. These challenges forced that most applications on P2P are storage applications where data is seldom changed and is self-verifying, i.e. it is possible to detect either malicious or accidental modifications by checking the data itself.
Within this context, our proposal in this dissertation is the specification of an infrastruture for the construction of arbitrary applications, by means of a shared memory abstraction, that tolerates the participation of a certain number of malicious nodes.
The central idea consists of leveraging State Machine Replication (SMR) techniques on top of P2P networks. The problem is SMR has scalability issues as the number of messages exchanged ikn replica coordination is quadratic. Given that, aor proposal is to split the P2P network in sets of limited size, called segments, in a way to ensure the SMR protocolos perform well. Segments are dynamic, i.e. they can grow or shrink as nodes join or leave the system, but the infrastruture guarantees, either by merging or splitting segments, that their size keeps within established limits.
The system was designed as a stack of layers whose functionality is defined by a set of operations and its properties. The operations of the segmentation layer were implemented by distributed algorithms written in pseudocode. The correct operation of these algorithms was shown by theorem proofs. Furthermore, a critical analysis of these algorithms clarified limitations and assessed their costs. In order to demonstrate the expressiveness of the proposed infrastructure, a tuple space was built using the implemented operations.

Keywords: Peer-to-Peer Networks. Dynamic Distributed Systems. Segmentation. Shared Memory. Intrusion Tolerance.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ALGORITMOS

SUMÁRIO

1 INTRODUÇÃO

Graças à evolução que vem experimentando as tecnologias ligadas à computação e à comunicação, diversos modelos computacionais se tornaram viáveis, a partir de modificações e expansões de modelos anteriores. Nesse contexto surgiram os Sistemas Distribuídos Dinâmicos (SDD) (MOSTEFAOUI et al., 2005), que representam uma flexibilização dos sistemas distribuídos tradicionais e estão em foco nas pesquisas da comunidade científica. Alguns exemplos de SDDs são as redes móveis *ad hoc* (MANETs), redes de sensores sem fio, grades computacionais oportunistas, redes *overlay* e redes par a par (*peer-to-peer*).

Os SDDs se caracterizam por não poderem contar com entidades permanentes no sistema (PIERGIOVANNI, 2005). As computações e protocolos devem ser executados corretamente mesmo diante da entrada e saída contínua de participantes, sem conhecimento a priori do número de nós ou até mesmo de um limite para o tamanho do sistema. Além disso, muitos dos SDDs são projetados para serem sistemas abertos, de forma que pode haver heterogeneidade tanto dos participantes quanto da rede.

Devido a essa heterogeneidade e ao grande dinamismo dos participantes, há diversas dificuldades para desenvolver aplicações para SDDs. Entre essas dificuldades, estão as possíveis falhas de participantes, a frequência de entrada e saída de participantes no sistema (*churn*) (GODFREY; SHENKER; STOICA, 2006) e a presença de participantes maliciosos tentando explorar vulnerabilidades do sistema. As pesquisas em SDDs tem como foco principal o desenvolvimento de algoritmos e protocolos que funcionem corretamente mesmo diante dessas dificuldades.

Um dos exemplos de SDDs mais estudados atualmente são as redes par a par (*peer-to-peer*, P2P). O uso desse paradigma foi bastante popularizado na Internet, principalmente por ser a base para as aplicações de compartilhamento de arquivos modernas. Com a evolução da arquitetura, diversas outras aplicações já foram desenvolvidas usando P2P, como multicast e sistemas de e-mail (STEINMETZ; WEHRLE, 2005). Apesar disso, redes P2P ainda são pouco utilizadas em aplicações mais complexas que poderiam se beneficiar da escalabilidade das mesmas (BALDONI et al., 2007). A grande maioria dos sistemas P2P são aplicações de disseminação de informações pouco mutáveis ou autoverificáveis.

As principais características que tornam as redes P2P uma arquitetura interessante para sistemas distribuídos são o uso eficiente dos recursos ociosos disponíveis na Internet e a capacidade de aumento do número de nós sem detrimento do desempenho. Algumas redes P2P oferecem primitivas de

comunicação com latência e número de mensagens de ordem logarítmica em relação ao número de nós (STOICA et al., 2001; ROWSTRON; DRUSCHEL, 2001). Essa escalabilidade permite que recursos disponíveis em uma grande quantidade de nós possa ser utilizado de maneira vantajosa.

Apesar das suas vantagens, as redes P2P apresentam desafios para o provimento de garantias de confiabilidade. Essas redes normalmente são formadas dinamicamente por nós totalmente autônomos que podem entrar e sair do sistema a qualquer momento. Essas características de dinamismo tornam difícil a manutenção da consistência das informações distribuídas no sistema. Além disso, essas redes não possuem uma gerência global, são redes de pares com grande abertura. Devido a essa abertura, as redes P2P podem conter participantes maliciosos que colocam em risco a segurança e o funcionamento das aplicações (SIT; MORRIS, 2002; WALLACH, 2003).

Por outro lado, em sistemas distribuídos há um grande número de trabalhos que apresentam soluções para coordenação de processos e manutenção de consistência de aplicações na presença de nós maliciosos. A maioria desses trabalhos faz uso de técnicas de Replicação Máquina de Estados (RME) (SCHNEIDER, 1990), que por sua vez são baseadas em soluções para o problema de consenso bizantino (LAMPORT; SHOSTAK; PEASE, 1982). RME é um mecanismo bastante geral para garantir o funcionamento de aplicações mesmo na presença de nós maliciosos, no entanto a utilização direta dessa estratégia para um grande número de nós não é viável: os protocolos para resolver consenso são de de troca de mensagens de ordem quadrática em relação ao número de nós, ou seja, não são apropriados para grandes sistemas.

A proposta deste trabalho é de aplicar técnicas de RME a redes P2P, de forma a prover uma infraestrutura que tenha a consistência garantida pela RME e a possibilidade de escala das redes P2P. Essa infraestrutura é definida por um conjunto de operações que permitem a construção de aplicações quaisquer utilizando um modelo de programação baseado em memória compartilhada distribuída.

Em sistemas grandes, compostos de muitos participantes, é comum a utilização da estratégia de dividir os componentes em grupos de tamanho gerenciável e prover algum mecanismo de coordenação intergrupos. Dessa forma, a infraestrutura proposta nessa dissertação divide um *overlay* P2P em grupos denominados de segmentos e aplica a RME a cada segmento. O desafio principal enfrentado na construção da solução consiste em dar suporte ao dinamismo das redes P2P, permitindo a entrada e saída de nós durante a execução. Assim, a segmentação gera grupos reconfiguráveis dinamicamente e coordenados entre si para garantir que o estado da aplicação continue disponível ao longo de toda a execução.

A infraestrutura proposta é definida formalmente como um conjunto

de operações com propriedades de funcionamento específicas. Essas operações fornecem um meio para construir aplicações que podem acessar todos recursos dos nós da rede P2P com a garantia de tolerância a um certo número de participantes maliciosos. A fim de demonstrar essa possibilidade, apresentamos um estudo de caso onde construímos um espaço de tuplas, uma estrutura de dados compartilhada que permite coordenação de processos de forma desacoplada (GELERNTER, 1985), usando apenas as operações fornecidas pela infraestrutura de segmentação. O espaço de tuplas fornece uma abstração interessante para coordenação em redes P2P, pois a larga escala e o alto dinamismo dificultam a comunicação ponto a ponto tradicional por trocas de mensagens.

1.1 MOTIVAÇÃO

As tecnologias e recursos de comunicação têm sofrido evolução constante nos últimos anos. Além de tornar a comunicação mais rápida e barata, essa evolução tornou possível a construção de novas aplicações e formas de interação. Dentre essas novas formas de aplicações estão aquelas denominadas de Sistemas Distribuídos Dinâmicos (MOSTEFAOUI et al., 2005; BALDONI et al., 2007), nas quais a composição e topologia do sistema sofre alterações durante a execução. Devido a esse dinamismo, existe uma dificuldade de implementar uma gerência centralizada da localização dos componentes, o que traz a necessidade de diluir a responsabilidade pela organização do sistema entre os componentes distribuídos. Essa diluição, juntamente com a heterogeneidade e dinamismo determinam a necessidade de fundamentar as aplicações em atributos de segurança de funcionamento bem definidos.

As características dos SDDs fazem com que seja um desafio inerente a esses sistemas projetar e desenvolver arquiteturas e algoritmos capazes de detectar e se adaptar a mudanças e permitir a reconfiguração em tempo de execução. Essa autoadaptação visa garantir que o sistema continuará a atender aos requisitos de funcionamento da aplicação sem deteriorar a qualidade de serviço.

Redes P2P se inserem nesse contexto na medida em que apresentam características de alto dinamismo e distribuição e são baseadas na distribuição da responsabilidade de manter as propriedades do sistema. As redes P2P diferem drasticamente dos sistemas cliente/servidor tradicionais por não centralizar as informações, bem como a gerência, em um único servidor ou *cluster* de servidores. Ao contrário, todos os nós participantes podem enviar e receber requisições e não há dependência fixa de um ponto central.

Com o surgimento das primeiras aplicações P2P de compartilhamento

de arquivos, ficou clara a necessidade de prover às pessoas meios para distribuir e não apenas consumir informações. A Internet e a Web baseadas em servidores de conteúdo centralizados e controlados por empresas não correspondem à estrutura social fora do mundo virtual, e por isso as redes P2P tiveram grande aceitação popular. A possibilidade de não apenas consumir mas também prover informação e recursos de maneira simples transformou a forma como as pessoas interagem no mundo virtual. Assim sendo, os SDDs e as redes P2P representam uma evolução no sentido de permitir um maior reflexo das estruturas sociais no mundo virtual.

1.2 OBJETIVOS

O objetivo geral dessa dissertação é a elaboração e avaliação de uma infraestrutura dinâmica, tolerante a faltas bizantinas que permita a construção de aplicações distribuídas de larga escala baseadas em modelo de programação de memória compartilhada. O dinamismo e a escalabilidade da infraestrutura devem ser alcançados de acordo com o uso de técnicas e filosofias de redes P2P (STEINMETZ; WEHRLE, 2005; VU; LUPU; OOI, 2010) e a tolerância a faltas bizantinas por meio do uso de técnicas de Replicação Máquina de Estados (SCHNEIDER, 1990).

Os objetivos específicos desta pesquisa, derivados do objetivo geral acima descrito, estão enumerados a seguir:

- estudar modelos de sistemas distribuídos dinâmicos e redes P2P, bem como abstrações de memória compartilhada, a fim de definir precisamente o problema e elaborar o modelo de sistema adotado;
- propor uma infraestrutura para construção de aplicações conforme o objetivo geral descrito acima e definir as propriedades garantidas pela infraestrutura;
- especificar o funcionamento da infraestrutura por meio de algoritmos distribuídos, definidos em um nível de detalhe que permita demonstrar a adequação às propriedades enumeradas;
- demonstrar por meio de provas de teoremas e lemas que os algoritmos elaborados se adequam às propriedades da infraestrutura proposta;
- analisar a complexidade e os custos dos algoritmos elaborados;
- realizar um estudo de caso por meio da construção de uma aplicação de memória compartilhada expressiva sobre a infraestrutura proposta.

1.3 MÉTODO

Dados os objetivos dessa dissertação, enumerados na Seção 1.2, pode-se notar que a contribuição principal é um estudo sobre algoritmos distribuídos para o problema da construção de aplicações distribuídas de memória compartilhada em ambientes dinâmicos. O resultado desse estudo foi a proposição de um suporte algorítmico para a construção de memórias compartilhadas tolerantes a intrusões nestes sistemas dinâmicos. Além disso, foi parte importante dos resultados uma avaliação criteriosa dos algoritmos elaborados que permitiu avaliar a correção e a expressividade dos mesmos.

A fim de se definir precisamente qual o problema a ser atacado, foi realizada uma consulta à literatura de sistemas distribuídos, incluindo livros e artigos de congressos ou periódicos, em especial relacionados a sistemas distribuídos dinâmicos e redes P2P. A partir do levantamento realizado, foi possível determinar uma carência de estudos sobre memórias compartilhadas de larga escala e tolerantes a intrusões. Dentro desse tema, ficou claro que se fazia necessário construir uma infraestrutura geral que permitisse a implementação de aplicações complexas.

Uma vez definido o problema e a abordagem utilizada, passamos a um estudo para encontrar soluções existentes para subproblemas e problemas relacionados aos objetivos dessa dissertação. Uma parte dos trabalhos havia sido encontrada na fase anterior, durante a determinação do problema de pesquisa. Desse outro levantamento resultou uma arquitetura em camadas, na qual a infraestrutura proposta faz uso de soluções conhecidas para os subproblemas. Nessa etapa ficou definida a estrutura geral da solução, incluindo o modelo de sistema, as primitivas utilizadas das camadas inferiores e as propriedades que a infraestrutura proposta deveria satisfazer.

O passo seguinte constituiu a elaboração dos algoritmos e a prova do funcionamento dos mesmos. Para tal foi utilizada uma abordagem iterativa, sendo que evoluções na escrita dos algoritmos permitiram formular demonstrações mais completas, ao mesmo tempo que a construção das provas exigiu uma análise mais aprofundada dos algoritmos, o que muitas vezes explicitou falhas ou a necessidade reformulação em termos mais simples. Outro resultado da análise minuciosa dos algoritmos foi um levantamento das limitações e casos excepcionais não tratados. Optou-se por não tratar nos algoritmos diversas situações de ocorrência menos frequente, que tornariam o entendimento dos procedimentos muito mais complexo. Esses casos foram explicitados e correções nos algoritmos para tratar essas situações foram descritas textualmente. Apesar disso, as provas de funcionamento dos algoritmos foram elaboradas de maneira geral, evitando a necessidade de ressalvas quanto aos casos excepcionais não cobertos pelos algoritmos e, portanto, se aplicam

também a esses casos.

O estudo de caso foi planejado tendo em vista a experiência do orientador deste trabalho sobre espaços de tuplas. As características dos espaços de tuplas se mostraram bastante interessantes para as redes P2P, e como essa abstração é um tipo de memória compartilhada, então houve um casamento perfeito com os objetivos da dissertação. A elaboração do estudo de caso se deu de maneira similar à etapa anterior, no entanto as operações e propriedades são aquelas clássicas de espaços de tuplas e foi dado um tratamento informal à análise do funcionamento dos algoritmos.

Os resultados parciais das etapas deste trabalho foram inicialmente redigidos no formato de artigo. Isso propiciou a submissão do trabalho para revisão de membros da comunidade científica (BÖGER et al., 2011, 2012). As revisões ajudaram a identificar pontos obscuros ou imprecisos. Posteriormente, esta dissertação foi elaborada a partir do texto do artigos, que deu origem principalmente ao Capítulo 3.

1.4 ORGANIZAÇÃO DA DISSERTAÇÃO

O restante da dissertação está dividido nos seguintes capítulos:

- O Capítulo 2 apresenta os conceitos básicos que são utilizados ao longo dos demais capítulos. A principal função é introduzir e esclarecer a terminologia utilizada e apresentar alguns problemas e soluções relevantes no restante do texto;

- O Capítulo 3 descreve a infraestrutura proposta como solução para o problema de pesquisa. Inclui o modelo de sistema, a especificação da camada de segmentação, os algoritmos e provas, a discussão das limitações da solução e uma comparação com trabalhos relacionados;

- O Capítulo 4 contém um estudo de caso onde um espaço de tuplas é construído utilizando as operações providas pela camada de segmentação. São apresentados os conceitos básicos relacionados especificamente com espaços de tuplas e a construção apresentada, os algoritmos elaborados, um discussão de suas limitações e uma comparação com outros espaços de tuplas similares;

- O Capítulo 5 resume os resultados obtidos e compara os mesmos com os objetivos da dissertação. Além disso, contém uma lista de trabalhos que podem ser desenvolvidos a partir do que foi obtido neste trabalho.

2 CONCEITOS BÁSICOS

Este capítulo apresenta uma introdução sobre diversos aspectos, conceitos, problemas notáveis e soluções, relacionados à tolerância a intrusões em sistemas distribuídos dinâmicos. Não temos a intenção de prover descrições completas e detalhadas, nem a pretensão de ensinar o leitor esses assuntos. A ideia é enumerar os principais termos que serão utilizados no restante desta dissertação, e permitir que o leitor especialista no assunto se habitue à terminologia específica utilizada e compreenda melhor o foco do trabalho. Ao leitor casual, este capítulo cabe como introdução superficial, porém precisa, aos assuntos e termos tratados, e permite uma leitura mais fácil dos demais capítulos. Sempre apontamos as fontes das informações e referências que possibilitam um maior aprofundamento, caso seja necessário.

O restante do capítulo está dividido da seguinte forma: a Seção 2.1 apresenta as características básicas de sistemas distribuídos e compara alguns modelos de sistemas distintos. A Seção 2.2 introduz problemas fundamentais desses sistemas e cita algumas soluções da literatura. A Seção 2.3 fala de um tipo especial de sistemas distribuídos dinâmicos, as redes P2P, que são parte do contexto do problema tratado por esta dissertação. A Seção 2.4 conclui o capítulo.

2.1 SISTEMAS DISTRIBUÍDOS

Um **sistema distribuído** (SD) é considerado neste trabalho como formado por um conjunto de componentes, denominados de **processos**, que se comunicam e coordenam suas ações por meio de trocas de mensagens. Esses componentes executam de maneira concorrente, sem uso de um relógio global ou memória física compartilhada. Os componentes (processos) desses sistemas podem sofrer eventuais falhas que ocorrem de maneira independente.

Um programa simples, composto por apenas um processo, executa um algoritmo cujos passos são estritamente sequenciais. Esse algoritmo determina o estado e o comportamento do processo. Em um sistema distribuído, comportamento e estado de múltiplos processos são controlados por um **algoritmo distribuído** (ou protocolo), uma sequência de passos executada em cada um dos processos do sistema, incluindo a troca de mensagens entre processos. Mensagens são usadas para passar informações de um processo a outro ou para coordenar a execução de atividades distribuídas no sistema.

Em geral, não é possível determinar a velocidade com a qual cada processo progride, nem o tempo que as mensagens levam para chegar nos

processos de destino. Dependendo de como o sistema se comporta dentro dessa limitação, este é classificado segundo níveis de sincronismo (LYNCH, 1996; ATTIYA; WELCH, 2004). Os dois modelos mais simples derivam de condições opostas quanto ao nível de sincronismo do sistema: um sistema é dito **síncrono** se existem limites conhecidos para o tempo que um processo leva para executar um passo, para o tempo que uma mensagem ser transmitida e para o desvio do relógio local de um processo com relação ao tempo real; um sistema é dito **assíncrono** se nenhum desses limites existe, ou seja, um processo pode levar um tempo arbitrário para executar um passo (um milissegundo ou um dia ou mais), uma mensagem pode levar um tempo arbitrário para ser transmitida e o relógio local de um processo pode desviar do tempo real de maneira arbitrária.

Dentro de um modelo de sistema síncrono, é possível resolver problemas assumindo características temporais determinadas, por exemplo, usando temporizadores para detectar falhas de processos ou perda de mensagens. No entanto, é difícil construir um sistema com limites definidos, sendo necessário conhecer a priori os recursos de processamento e comunicação necessários para a execução dos algoritmos e garantir a disponibilidade desses recursos durante toda a execução. Se o sistema violar as premissas dos limites de tempo, a solução deixa de ser confiável.

Um sistema assíncrono representa um modelo mais "fraco", isto é, menos problemas podem ser resolvidos nesses sistemas. Um exemplo de problema que não pode ser resolvido em um sistema assíncrono é o consenso em que pelo menos um participante pode falhar (ver Seção 2.2.1). Qualquer solução que funcione em um sistema assíncrono também funciona em um sistema síncrono (CHANDRA; TOUEG, 1996). Por outro lado, certos problemas que podem ser resolvidos em sistemas síncronos, ou não admitem solução em sistemas assíncronos, ou a solução é mais complexa. Dentre os problemas que não podem ser resolvidos em sistemas totalmente assíncronos, há aqueles que podem ser resolvidos em sistemas que não sejam totalmente síncronos. Esses sistemas são denominados parcialmente síncronos (DWORK; LYNCH; STOCKMEYER, 1988) e possuem algumas premissas de limites de tempo, porém com menor rigidez que nos sistemas síncronos. Por exemplo, um sistema pode exigir que pelo menos durante certos períodos os limites de sincronismo sejam obedecidos.

Um outro aspecto importante de sistemas distribuídos diz respeito à possibilidade de ocorrência de faltas, ou seja, de desvios com relação ao comportamento esperado de um processo. Qualquer sistema real está sujeito a algum tipo de falta, seja devido a erros de programação ou configuração, seja por causa de desgaste de peças físicas ou catástrofes naturais. Existem várias classificações ou modelos de faltas, porém dois se destacam como mais

utilizados: as **faltas de parada** (*crash*) nas quais um processo pode parar definitivamente a execução do algoritmo antes do término do mesmo; e as **faltas bizantinas** ou arbitrárias, nas quais um processo pode desviar de maneira arbitrária do algoritmo distribuído executado e até mesmo agir de maneira a tentar prejudicar a execução dos demais processos maliciosamente.

Sistemas distribuídos podem ser tolerantes a faltas, ou seja, podem continuar funcionando corretamente mesmo que alguns processos exibam comportamento faltoso. Em geral, sistemas que toleram faltas bizantinas são mais complexos que sistemas que toleram faltas de parada, uma vez que soluções para o primeiro precisam antever uma gama muito maior de situações adversas produzidas por ações arbitrárias dos processos faltosos. Além disso, soluções no modelo de faltas bizantinas automaticamente se aplicam ao modelo de faltas de parada, mas o inverso não ocorre.

Um sistema distribuído deve conviver com falhas de processos e o tratamento desses componentes falhos envolve normalmente técnicas de reconfiguração. Essas reconfigurações são a forma do sistema se adaptar no sentido de manter suas funcionalidades mesmo na ocorrência de falhas parciais. Essas mudanças, no entanto, não se limitam somente a processos faltosos. Há situações em que mudanças dinâmicas na composição do sistema fazem parte da definição do próprio modelo do sistema, ou seja, o sistema prevê, dentro do comportamento correto de processos, a possibilidade de que alguma característica estrutura do sistema se altere (BALDONI et al., 2007). Essas adições ou remoções de processos devem ocorrer no sistema sem a necessidade de reiniciar a execução de um processamento distribuído. Sistemas que convivem com essas alterações são chamados de **sistemas distribuídos dinâmicos** (SDD). Um sistema que não admite alteração da sua estrutura durante toda a execução é chamado de **estático**.

Em uma classe grande de SDDs, chamados de sistemas com participantes desconhecidos, não é possível assumir o conhecimento do conjunto total de processos que fazem parte da composição do sistema (BALDONI et al., 2007). Além disso, em certas situações pode ocorrer o fenômeno de *churn* desmesurado, onde a frequência de entradas e saídas de processos é tão alta que todos os recursos do sistema são gastos para se adaptar às mudanças (GODFREY; SHENKER; STOICA, 2006). Por esses e outros motivos, SDDs formam um modelo de sistema mais "fraco" que o modelo de sistemas estáticos, ou seja, é mais difícil oferecer garantias e propriedades de funcionamento.

O exemplo mais importante de SDD é a própria Web. Processos são constantemente adicionados e removidos e o sistema como um todo se adapta e continua funcionando. Em um escopo menor, as redes par a par (P2P) são bastante populares e seguem um modelo de SDD. Nessas redes os processos

são autônomos e podem escolher o momento em que desejam entrar e sair do sistema. Na proposta central deste trabalho, lidamos com um SDD, porém o foco principal é em redes P2P.

2.2 PROBLEMAS FUNDAMENTAIS EM SISTEMAS DISTRIBUÍDOS

Esta seção introduz alguns problemas fundamentais em sistemas distribuídos. Esses problemas aparecem diversas vezes como parte de problemas maiores, de forma que são amplamente discutidos na literatura de sistemas distribuídos. As subseções a seguir apresentam enunciados clássicos de problemas encontrados na literatura e algumas soluções para diferentes modelos de sistema são discutidas superficialmente.

2.2.1 Consenso

O problema de **consenso** (FISCHER; LYNCH; PATERSON, 1983; CHANDRA; TOUEG, 1996) consiste em fazer com que os processos do sistema distribuído proponham valores e depois decidam sobre um dos valores propostos. Para se caracterizar o consenso, todos os processos devem decidir pelo mesmo valor. Formalmente, o consenso é definido em termos de duas primitivas: $propose(v)$ e $decide(v)$, onde v é retirado do conjunto de possíveis valores a serem propostos. Quando um processo executa $propose(v)$, é dito que esse processo está propondo v; similarmente, quando um processo executa $decide(v)$, é dito que esse processo está decidindo por v.

Em um sistema que não admite falhas de processos, a solução para o consenso é simples, bastando que cada processo comunique sua proposta aos demais e aguarde o recebimento de todas as propostas dos demais processos; na sequência todos os processos aplicam um critério determinado comum, por exemplo, escolher a proposta majoritária, para decidir por um valor dentre todas as propostas. Quando se admite a possibilidade de ocorrência de falhas, no entanto, um processo faltoso pode enviar propostas distintas aos demais e provocar inconsistência nas decisões. Por outro lado, um processo pode sofrer uma falta de parada e enviar as propostas para apenas uma parte dos demais processos, de forma que aqueles que não receberem a proposta do processo faltoso não podem decidir. No modelo de faltas de parada, o problema é definido pelas seguintes propriedades (CHANDRA; TOUEG, 1996):

- Terminação: em algum momento, todo processo correto termina por decidir algum valor;

- Integridade uniforme: todo processo decide no máximo uma vez;
- Acordo: o valor decidido por todos os processos corretos é o mesmo, isto é, se um processo correto decide por um valor, então qualquer outro processo correto decide pelo mesmo valor;
- Validade uniforme: se um processo decide um valor, este valor deve ter sido proposto por algum processo.

No modelo de faltas bizantinas as propriedades uniformes são impossíveis de garantir, devido ao comportamento arbitrário de nós faltosos. Estas são substituídas por alternativas mais fracas, onde somente processos corretos são restringidos.

Uma característica fundamental do problema de consenso é que não há solução para sistemas assíncronos se houver pelo menos um processo faltoso, independente do tipo de falta (FISCHER; LYNCH; PATERSON, 1983). Em vista dessa impossibilidade, as soluções para consenso encontradas na literatura propõem modelos de sistema não completamente assíncronos, mas parcialmente síncronos (DWORK; LYNCH; STOCKMEYER, 1988).

Em sistemas distribuídos estáticos há diversas soluções para o consenso, tanto tolerando faltas de parada (LAMPORT, 1998; LAMPSON, 2001) quanto tolerando faltas bizantinas (LAMPORT; SHOSTAK; PEASE, 1982; ATTIYA; DOLEV; GIL, 1984; CASTRO; LISKOV, 2002; MARTIN; ALVISI, 2006a; LAMPORT, 2011). Em sistemas dinâmicos, no entanto, ainda há relativamente menos soluções na literatura (CAVIN; SASSON; SCHIPER, 2004; GREVE; TIXEUIL, 2007, 2010; ALCHIERI et al., 2008).

2.2.2 Difusão Atômica

O problema da **difusão atômica** ou **difusão com ordem total** (CHANDRA; TOUEG, 1996; DÉFAGO; SCHIPER; URBÁN, 2004) (*total order multicast*) consiste em garantir que mensagens difundidas em um grupo de processos são entregues em todos os processos na mesma ordem. Soluções para esse problema fornecem primitivas de comunicação mais fortes que facilitam a construção de sistemas distribuídos tolerantes a faltas. Em especial, a difusão atômica permite a construção de sistemas por meio da Replicação Máquina de Estados (RME) (SCHNEIDER, 1990), descrita na Seção 2.2.3.

O problema de difusão atômica é caracterizado a partir de duas primitivas básicas (HADZILACOS; TOUEG, 1994): $TO\_multicast(G,m)$ difunde a mensagem *m* no conjunto de processos *G*; $TO\_deliver(m)$ consiste na entrega de uma mensagem *m*, previamente difundida em *G*, para a aplicação. As

propriedades da difusão com ordem total são as seguintes (HADZILACOS; TOUEG, 1994; CHANDRA; TOUEG, 1996):

- Validade: se uma mensagem m é difundida por um processo correto, então m é entregue por algum processo correto;
- Acordo: se algum processo correto entrega uma mensagem m, então todo processo correto entrega m;
- Integridade: para qualquer mensagem m, um processo correto somente entrega m se m foi previamente difundida e nenhuma mensagem é entregue mais de uma vez;
- Ordem total: se dois processos corretos entregam as mensagens m e m', então um processo entrega m antes de m' se e somente se o outro processo também entrega m antes de m'.

Chandra e Toueg (1996) demonstraram que a difusão atômica é equivalente ao problema de consenso no modelo de faltas de parada. Correia, Neves e Veríssimo (January 2006) demonstraram a mesma equivalência para faltas bizantinas. Dessa forma, as soluções citadas na Seção 2.2.1 podem ser transformadas em soluções para a difusão atômica.

2.2.3 Replicação Máquina de Estados

A **Replicação Máquina de Estados** (RME) (SCHNEIDER, 1990) é uma estratégia para construir aplicações tolerantes a faltas (de parada ou bizantinas). A RME consiste basicamente em replicar o estado da aplicação em um conjunto de servidores que sempre executam as mesmas operações na mesma ordem. Essa propriedade é garantida pelo uso de algoritmos de difusão atômica tolerante a faltas, de forma que os servidores corretos evoluam consistentemente e mantenham a aplicação funcionando corretamente. Se todos os servidores corretos partem de um mesmo estado inicial, executam as mesmas operações na mesma ordem e essas operações são deterministas, então todos os servidores corretos terminam por atingir o mesmo estado final. Essa propriedade é denominada de determinismo de réplica.

Por utilizar difusão atômica, e, por consequência, consenso, a RME está sujeita ao mesmo resultado de impossibilidade em sistemas assíncronos com uma única falta. Assim sendo, para a utilização de RME é necessário assumir um sistema parcialmente síncrono.

Em sistemas distribuídos estáticos, ou seja, quando o número de réplicas é fixo, os mesmos algoritmos de consenso citados na Seção 2.2.1 podem

ser aplicados. Por outro lado, em sistemas distribuídos dinâmicos é necessário prover meios para a entrada e saída de réplicas. Lamport, Malkhi e Zhou (2010) definem um mecanismo para permitir utilizar algoritmos de RME estáticos em sistemas dinâmicos. O mecanismo consiste em utilizar uma operação que para a execução da RME. Quando uma RME com uma certa composição de réplicas é parada, uma nova RME é formada, levando em conta as entradas e saídas de processos, e iniciada com o mesmo estado da RME anterior. Dessa forma, a RME dinâmica é dividida em uma sequência de RMEs estáticas.

2.2.4 Memória Compartilhada Distribuída

Memória compartilhada distribuída é uma abstração para compartilhamento de dados entre processos que não compartilham acesso à mesma memória física (LYNCH, 1996; ATTIYA; WELCH, 2004). Essa abstração permite o uso de um modelo de programação baseado em memória compartilhada, que possui vantagens sobre modelos baseados em trocas de mensagens. Uma das vantagens é a atualização automática de dados alterados por outro processo, mesmo considerando que essa visão de memória envolve a distribuição da informação em níveis mais baixos de abstração.

A forma mais simples de memória compartilhada são as **variáveis compartilhadas**: um valor simples que pode ser lido e escrito. Normalmente as variáveis compartilhadas são implementadas, em um sistema tolerante a faltas, por meio de **sistemas de quóruns** (GIFFORD, 1979). Um sistema de quóruns é um conjunto de processos contendo subconjuntos, denominados quóruns, tais que quaisquer dois quóruns apresentam intersecção de processos. Dessa forma, se a variável é alterada em todos os processos de um quórum, fica garantido que a alteração será refletida em todos os outros quóruns. Existem também sistemas de quóruns que toleram faltas bizantinas (MALKHI; REITER, 1998).

Uma característica interessante de sistemas de quóruns é que estes funcionam em sistemas assíncronos, ou seja, não necessitam de nenhuma premissa temporal. Além disso, existem soluções para sistemas dinâmicos tanto no contexto de faltas de parada (LYNCH; SHVARTSMAN, 2002; AGUILERA et al., 2009) quanto de faltas bizantinas (ALCHIERI et al., 2012). Apesar dessa característica, os sistemas de quóruns são menos poderosos que a Replicação Máquina de Estados, isto é, resolvem uma classe menor de problemas.

Além de variáveis compartilhadas, existem também outras abstrações de memória compartilhada mais complexas, onde estruturas compostas são

compartilhadas. De interesse para este trabalho são os **espaços de tuplas** (GELERNTER, 1985), onde vários conjuntos ordenados de dados são armazenados em um espaço compartilhado. Mais detalhes sobre os espaços de tuplas são apresentados na Seção 4.1.

2.3 REDES PAR A PAR

Redes par a par (*peer-to-peer* ou P2P), também chamadas de sistemas par a par, são um paradigma de computação distribuída onde os nós ou processos componentes do sistema trocam informações diretamente entre si sem a necessidade de um servidor central (STEINMETZ; WEHRLE, 2005; VU; LUPU; OOI, 2010). A tecnologia foi aplicada, há mais de trinta anos, na construção da USENET[1] em 1979, porém na época o número de usuários era relativamente pequeno e não foi possível reconhecer as maiores vantagens do P2P. Foi somente com a popularização da Internet, e posteriormente das aplicações de compartilhamento de arquivos, que as redes P2P permitiram a união de milhões de usuários em um único sistema.

O modelo de computação das redes P2P está baseado em dividir igualmente a responsabilidade de prover dados e serviços entre todos os participantes, de maneira que qualquer nó do sistema é tanto um cliente quanto um servidor, ou seja, tanto consome quanto disponibiliza recursos. A computação sobre uma rede P2P compartilha muitas das propriedades dos sistemas distribuídos em geral, no entanto, algumas características se diferenciam dos sistemas tradicionais (VU; LUPU; OOI, 2010):

- Simetria de papeis: todos os nós participantes do sistema têm a mesma função, sendo tanto cliente quanto servidor;

- Escalabilidade: sistemas P2P podem funcionar com bom desempenho mesmo quando o número de nós chega a milhares;

- Heterogeneidade: normalmente sistemas P2P não apresentam um controle de participação, logo são sistemas bastante abertos e sujeitos à participação de nós altamente díspares em termos de velocidade de processamento e de transmissão;

- Controle distribuído: seguindo a definição mais rígida, P2P requer descentralização total, ou seja, não devem existir estruturas centralizadas em redes P2P;

[1] http://www.usenet.net/

- Dinamismo: sistemas P2P normalmente são compostos de nós autônomos que podem decidir quando entrar e sair do sistema. Dessa forma a carga e os recursos disponíveis tendem a variar com o tempo.

Recentemente, houve um grande aumento nas pesquisas para desenvolver aplicações e sistemas P2P (VU; LUPU; OOI, 2010). De todas essas aplicações projetadas, duas classes podem ser definidas: aplicações de compartilhamento de recursos e aplicações de compartilhamento de dados. No primeiro caso, as aplicações utilizam recursos (ciclos de processador, espaço em disco, largura de banda) disponíveis nos nós do sistema. Em aplicações de compartilhamento de dados, usuários podem acessar, modificar ou trocar dados. Note que há uma diferença entre compartilhar dados e compartilhar recurso de armazenamento de dados: o primeiro caso, além do compartilhamento de armazenamento, também envolve controles de acesso, sistemas de notificação de alterações, etc.

Entre as principais aplicações de P2P estão compartilhamento de conteúdo (Gnutella[2] e Freenet[3]), computação científica (SETI@home[4] e Folding@home[5]), jogos (Net-Z[6] e Starcraft[7]), mensageiros instantâneos (Jabber[8] e Skype[9]), ambientes de trabalho cooperativos (Groove[10] e Magi[11]), *caching* e armazenamento colaborativos e bases de dados distribuídas.

Apesar dos benefícios e potenciais usos, os sistemas P2P, dadas suas características, também apresentam grandes desafios que ainda são alvo de pesquisas (VU; LUPU; OOI, 2010). Muitos desses desafios são resultado do grande dinamismo, heterogeneidade e abertura, pois nós autônomos podem entrar e sair do sistema a qualquer momento levando consigo recursos e dados, além de o sistema estar sujeito a participação de nós maliciosos que tentam corromper o funcionamento das aplicações.

A principal classificação de sistemas P2P divide em redes P2P não estruturadas e redes P2P estruturadas, conforme o tipo de organização de nós e objetos empregado (VU; LUPU; OOI, 2010). Nas redes P2P não estruturadas, a localização dos objetos não apresenta nenhum relacionamento com a estrutura subjacente, de forma que o custo de manutenção da estrutura é baixo, porém o custo da realização de buscas para encontrar objetos é alta.

[2] http://www.gnutella.com
[3] https://freenetproject.org/
[4] http://setiathome.berkeley.edu/
[5] http://folding.stanford.edu/
[6] http://www.proksim.com
[7] http://www.blizzard.com/
[8] http://www.jabber.org
[9] http://www.skype.com
[10] http://www.groove.net
[11] http://www.endeavors.com

Já nas redes P2P estruturadas, os objetos são distribuídos nos nós do sistema seguindo critérios bem definidos, por exemplo no nó que apresenta identificador numericamente mais próximo ao identificador do objeto. Nessas redes há um custo maior de manutenção da estrutura, porém a localização de objetos é bastante eficiente. As duas seções a seguir apresentam essas duas classes de sistemas em maior detalhe e citam os principais exemplos. Como uma rede P2P estruturada é a base da solução proposta nesta dissertação, será dado um foco maior nesse tipo de rede P2P.

2.3.1 Redes Par a Par Não Estruturadas

Nas redes P2P não estruturadas, como *Freenet* (CLARKE et al., 2001) e *Gnutella* o conteúdo ou os recursos disponibilizados por um nó não são de nenhuma forma associados ao seu endereço IP ou outro identificador (VU; LUPU; OOI, 2010). Além disso, não existe nenhuma topologia fixa para a rede. Usualmente, ao entrar na rede um nó contacta outro nó e obtém uma lista de endereços que formam o conjunto de vizinhos. Depois disso cada nó mantém sua lista independentemente. Isso simplifica o gerenciamento e reduz os custos com a manutenção da estrutura. Essa estratégia, no entanto, apresenta problemas de escalabilidade, já que uma vez que os nós não possuem conhecimento sobre a localização dos dados, buscas precisam ser enviadas a toda a rede.

O roteamento nessas redes normalmente faz uso de técnicas de inundação para realizar uma busca em um grande número de nós em curto intervalo de tempo, o que implica em um alto número de mensagens trocadas. Para aliviar o problema da inundação, a maioria das redes P2P não estruturadas anexa um valor TTL (*time to live*) para que as buscas possam ser eliminadas do sistema após um certo tempo ou número de saltos. Um problema que surge com o uso de TTLs passa a ser a escolha do valor ótimo que evite desperdício de largura de banda e tempo de resposta, mas ao mesmo tempo garanta que toda a rede seja buscada.

As duas formas mais básicas de busca são a busca em largura e busca em profundidade. Na busca em largura, ao receber um pedido de busca, o nó primeiro verifica se possui o dado requisitado. Em caso positivo, envia a resposta ao chamador. Caso o nó não possua o dado buscado, simplesmente encaminha a requisição a todos os seus vizinhos, exceto o remetente da busca. Nesse caso o TTL indica qual a profundidade máxima da busca. Nas primeiras versões do *Gnutella*, foi utilizada a busca em largura.

Na busca em profundidade, o TTL também especifica a profundidade máxima da busca, porém o nó realiza o encaminhamento da busca usando um

vizinho de cada vez, escolhendo primeiro o vizinho com maior probabilidade de conter a resposta. O sistema *FreeNet* (CLARKE et al., 2001) utiliza o esquema de busca em profundidade, juntamente com mecanismo de cache para armazenar resultados nos nós intermediários da busca.

Além desses dois mecanismos básicos, existem diversas outras estratégias de busca baseadas em heurísticas que visam otimizar situações comuns baseadas em modelos probabilísticos (LV et al., 2002; LI; WU, 2006).

2.3.2 Redes Par a Par Estruturadas

As redes par a par estruturadas possuem um mecanismo rígido para determinar a localização de objetos na rede, ou seja, objetos são armazenados em locais precisamente especificados (BALAKRISHNAN et al., 2003). Em geral, atribuem-se identificadores tanto a nós quanto a objetos e armazena-se os objetos em nós com o identificador mais próximo do identificador do objeto. Essa noção de proximidade varia de acordo com o sistema, porém o algoritmo de roteamento sempre garante que a cada salto a mensagem é encaminhada a um nó mais próximo do destino, de maneira a garantir a chegada em algum momento. Dessa forma, dada uma chave de busca, é possível determinar rápida e precisamente a localização de um objeto. A desvantagem dessa estratégia é a necessidade de mover dados quando ocorrem entradas e saídas de nós no sistema para manter as propriedades de localização. Além disso, a necessidade de uma topologia de rede implica em um certo custo de manutenção.

Normalmente as redes P2P estruturadas oferecem uma interface do tipo *distributed hash table* (DHT) (BALAKRISHNAN et al., 2003) que apresenta apenas uma operação: dada uma chave k, $lookup(k)$ encontra o endereço do nó responsável pela chave k. A partir da descoberta do nó, a aplicação pode manipular os dados relacionados à chave buscada fazendo invocações diretas. As DHTs apresentam bom desempenho para buscas pontuais, porém não dão suporte para buscas por intervalo. Para esse tipo de acesso existem redes P2P estruturadas baseadas em árvore, como o P-Grid (ABERER et al., 2003).

As redes P2P estruturadas variam de acordo com a topologia e com o tipo de roteamento utilizado. Esses aspectos estão relacionados com a construção e manutenção de tabelas de roteamento especiais que garantem o bom desempenho das buscas. As seções a seguir apresentam duas redes P2P estruturadas bastante citadas na literatura, *Chord* (STOICA et al., 2001) e *Pastry* (ROWSTRON; DRUSCHEL, 2001). Assim como a maioria das redes P2P, essas duas não foram construídas para tolerar participantes maliciosos (SIT;

MORRIS, 2002; WALLACH, 2003). Uma solução encontrada na literatura (CASTRO et al., 2002), baseada no *Pastry*, que funciona mesmo em um contexto de faltas bizantinas é apresentada na Seção 3.1.1.

2.3.2.1 *Chord*

Chord (STOICA et al., 2001) é um *overlay* P2P estruturado do tipo DHT. É um dos algoritmos de DHT mais elegantes por conta da sua simplicidade e eficiência. As chaves do DHT são números de l bits, no intervalo $[0, 2^l - 1]$, e formam um círculo módulo 2^l. Tanto nós como objetos possuem uma chave associada, normalmente calculada por meio da aplicação de uma função *hash* uniforme, como SHA-1[12]. Um objeto de chave k é armazenado no nó com o menor identificador maior que k, ou seja, o sucessor de k. Dessa forma, um nó é responsável por todos os objetos com chave menor que seu identificador e maior que o identificador do nó antecessor no círculo.

Chord possui duas formas de roteamento distintas: o roteamento simples e o roteamento escalável. Para o roteamento simples, cada nó armazena o endereço do nó sucessor no anel de identificadores. Ao receber uma mensagem endereçada para a chave k, o nó verifica se é o sucessor de k e em caso positivo recebe a mensagem. Se o nó não é o sucessor de k, então a mensagem é encaminhada ao sucessor do nó. Dessa forma a mensagem passa de sucessor em sucessor até chegar ao destino. Esse procedimento requer apenas armazenamento constante e um custo baixo de manutenção, porém leva um número de saltos linear no número de nós do sistema.

O roteamento escalável faz uso de uma estrutura de dados denominada de *finger table*. Se os identificadores contém l bits, então as tabelas de roteamento em cada nó possuem l entradas, indexadas por 0 a $l-1$. De maneira geral, se o nó possui identificador X, então a tabela de roteamento na entrada i contém o endereço do nó sucessor da chave $X + 2^i$, ou seja, a primeira entrada na tabela contém o endereço do nó responsável pela chave $X + 2^0 = X + 1$, o sucessor do nó, e as demais entradas da tabela ficam exponencialmente mais distantes de X.

Para encontrar o nó responsável por uma chave Y, é primeiro verificado se o nó atual é o sucessor de Y. Caso contrário, é procurado na tabela de roteamento a última entrada cuja chave associada seja menor que Y e a busca é repassada de maneira recursiva ao endereço registrado na tabela. Dessa forma, quanto mais distante for a chave procurada, maior será a distância percorrida pelo salto. Além disso, como as chaves das entradas da tabela crescem exponencialmente, o número de saltos dados tende a ser logarítmico

[12] http://tools.ietf.org/html/rfc3174

com relação ao tamanho do círculo de identificadores.

Para construir a tabela, um nó inicialmente busca pelo seu sucessor e preenche a primeira entrada. Cada entrada seguinte é construída realizando uma busca a partir do nó da entrada anterior, até que a tabela esteja completa. Para lidar com dinamismo, isto é, entradas e saídas de nós, um procedimento periódico é realizado para manter atuais as entradas da tabela: de tempos em tempos cada nó busca novamente os nós correspondentes às entradas da tabela substituindo os endereços anteriores para remover entradas de nós inativos e impedir a deterioração do desempenho.

2.3.2.2 *Pastry*

O *Pastry* (ROWSTRON; DRUSCHEL, 2001) é uma rede P2P estruturada do tipo DHT que garante roteamento eficiente de mensagens, usando uma tabela de roteamento no mesmo sentido do *Chord*. Apesar disso a estrutura difere em diversos aspectos. Um dos diferenciais do *Pastry* é construir as ligações com outros nós usando como critério de escolha a localidade de rede, isto é, dando preferência a nós com menor latência na troca de mensagens. Dessa forma, as mensagens devem percorrer uma distância física menor até o destino.

A estrutura subjacente do *Pastry* é um anel lógico ordenado pelos identificadores. Cada nó de uma rede *Pastry* possui um identificador de 128 bits, que indica sua posição em um anel lógico com valores de 0 a $2^{128} - 1$. Esse identificador é atribuído aleatoriamente, por exemplo pelo uso de uma função de *hash* criptográfica sobre o IP do nó ou sua chave pública, de forma a tornar provável a distribuição uniforme dos nós ao longo do anel.

Objetos possuem uma chave numérica no mesmo espaço de identificadores e são armazenados no nó com identificador numericamente mais próximo de sua chave. Em uma rede contendo N nós, *Pastry* pode rotear uma mensagem ao nó contendo um objeto em menos de $\lceil \log_{2^b} N \rceil$ passos, em uma execução sem faltas, sendo b um parâmetro global com valor típico de 4. Admitindo falhas de nós, a entrega ainda é garantida a menos que $\lfloor L/2 \rfloor$ nós adjacentes no anel lógico falhem simultaneamente, sendo L um parâmetro global com valores típicos de 16 ou 32.

Pastry utiliza um roteamento baseado em prefixo, isto é, para rotear uma mensagem destinada a uma chave k, a cada salto tenta-se encaminhar a mensagem para um nó cujo identificador possua o maior prefixo comum com k. Os identificadores e chaves são divididos em dígitos de 2^b bits, e o prefixo é medido em dígitos. Dessa forma, a cada salto o nó atingido possui identificador pelo menos um dígito mais próximo da chave destino. Por exemplo,

se identificadores possuem 128 bits e $b = 4$, então o roteamento deve levar no máximo $128/2^4 = 8$ passos. Há situações, no entanto, em que, em um passo do roteamento, o nó não conhece nenhum outro nó ativo cujo identificador possua prefixo comum com k um dígito maior que o seu prefixo comum com o próprio k. Nesse caso o algoritmo utiliza uma solução sub ótima, mas que ainda garante aproximação sucessiva com a chave, isto é, de todos os nós conhecidos pelo nó atual, encaminha-se a mensagem para aquele com o identificador numericamente mais próximo do destino.

Para suportar o procedimento de roteamento descrito , cada nó de uma rede Pastry mantém um estado composto de uma **tabela de roteamento**, um **conjunto de vizinhos** e um **conjunto folha** ($leafset$). O conjunto folha contém L nós com identificador mais próximo do identificador do nó atual, sendo $L/2$ nós com identificador menor e $L/2$ nós com identificador maior. Esse conjunto possui o papel de identificar quando o roteamento atingiu o destino, pois o nó testa se a chave de destino está dentro dos limites do conjunto folha, e nesse caso é entregue ao nó do conjunto com o identificador mais próximo da chave. Além disso, o conjunto folha garante que o roteamento irá progredir nos casos em que o roteamento por prefixos falha, pois a estrutura desse conjunto garante o conhecimento de algum nó com identificador mais próximo do destino, seja nos nós menores seja nos nós maiores.

A tabela de roteamento é composta de $\lceil \log_{2^b} N \rceil$ linhas com $2^b - 1$ entradas cada linha. Cada linha da tabela contém nós cujos identificadores possuem um prefixo comum com o nó atual progressivamente maior. As entradas da linha i possuem nós cujo identificador apresenta um prefixo comum com o identificador do nó atual de i dígitos. Além disso, em cada uma das $2^b - 1$ entradas da i-ésima linha da tabela, o nó da j-ésima entrada possui o valor j no dígito da posição $i+1$ do identificador. Dessa forma, para realizar o roteamento baseado em prefixo, primeiro o nó identifica o tamanho do prefixo comum do seu próprio identificador com a chave de destino. Se o número de dígitos em comum for l, então a l-ésima linha da tabela é consultada. Dessa linha, será usado como próximo passo do roteamento, o nó que possuir o mesmo dígito que a chave na posição $l+1$, ou seja, será escolhido um nó com prefixo comum um dígito maior.

A outra estrutura mantida pelos nós da rede *Pastry* é o conjunto de vizinhos. Esse conjunto não tem participação direta no roteamento e serve para prover parâmetros de localidade, isto é, o conjunto de vizinhos é um conjunto de nós fisicamente próximos.

Para entrar na rede *Pastry*, um nó precisa entrar em contato com outro nó que já seja participante da rede. Suponhamos que um nó X deseja entrar na rede a partir de um nó Y que já faz parte da rede. Para tal, Y envia uma mensagem especial usando o roteamento do *Pastry* para o nó com identifica-

dor mais próximo de X. Cada nó que recebe a mensagem, envia para X sua tabela de roteamento para que X construa a sua própria tabela e possa participar do roteamento. Como a mensagem é roteada com base nos prefixos de X, então a cada salto da mensagem especial o nó seguinte possui prefixo comum com X maior, ou seja, parte da sua tabela pode ser utilizada por X. Além disso, o nó destino, com identificador mais próximo de X, possui um conjunto folha que pode ser utilizado como base por X. O conjunto de vizinhos é construído procurando algum nó próximo de X, por exemplo usando IP *multicast*, e usando seu conjunto de vizinhos como base.

2.4 CONCLUSÃO DO CAPÍTULO

Neste Capítulo foram discutidos conceitos, problemas e soluções de sistemas distribuídos relacionados com o problema abordado nesta dissertação. A Seção 2.1 contém definições e propriedades de sistemas distribuídos e sistemas distribuídos dinâmicos (SDDs), que são o ambiente central sobre o qual as estruturas descritas nos Capítulos seguintes são construídas. Como o foco maior é em redes P2P, um tipo de SDD, a Seção 2.3 descreve as características específicas dessas redes, dando uma visão geral da taxonomia e apresentando exemplos relevantes dessas redes.

Como a intenção deste Capítulo não é de apresentar conteúdo didático, apenas descrições breves e superficiais foram apresentadas. No entanto, as referências que acompanham as descrições apontam para trabalhos relevantes da área, de forma que podem ser usadas, se necessário, para se chegar a um conhecimento bastante profundo dos assuntos tratados.

3 SEGMENTAÇÃO DE OVERLAYS PAR A PAR COMO SUPORTE PARA MEMÓRIAS TOLERANTES A INTRUSÕES

Neste capítulo, propomos uma infraestrutura para construção de aplicações de memória compartilhada sobre sistemas distribuídos de larga escala, tolerantes a presença de um certo número de nós maliciosos. Essa infraestrutura é construída por meio da aplicação de técnicas de Replicação Máquina de Estados (RME) (SCHNEIDER, 1990) a um *overlay* P2P estruturado, com a finalidade de aproveitar as características de ambas as técnicas, isto é, a tolerância a nós maliciosos da RME e a escalabilidade de sistemas P2P.

As premissas do sistema, as condições básicas para o funcionamento correto da infraestrutura proposta, estão descritas na Seção 3.1. Na sequência, são apresentados o *overlay* (Seção 3.1.1) e a RME (Seção 3.1.2), utilizados como base para a construção da proposta. As funcionalidades centrais da camada de segmentação são descritas na Seção 3.2 na forma de algoritmos. A mesma seção apresenta ainda provas do funcionamento dos algoritmos e uma análise sobre os custos relacionados aos mesmos. Ao fim do capítulo, na Seção 3.4, comparamos a infraestrutura proposta a outros trabalhos com alguma similaridade descritos na literatura.

3.1 MODELO DE SISTEMA

Consideramos um sistema distribuído formado por um conjunto possivelmente infinito Π de processos (ou nós), interconectados por enlaces de comunicação (*links*) formando desta maneira uma rede. Cada nó possui um endereço único de rede e pode enviar mensagens para qualquer outro nó, desde que conheça seu endereço. Um nó é considerado correto se age de acordo com a especificação dos protocolos nos quais participa. Um nó malicioso (ou bizantino (LAMPORT; SHOSTAK; PEASE, 1982)) não se comporta segundo as especificações, agindo muitas vezes de maneira arbitrária, em especial simplesmente parando em certos momentos. O sistema proposto neste trabalho tolera certo número de nós maliciosos durante sua execução. Assume-se que em qualquer momento da execução, no máximo f nós faltosos estão presentes no sistema. O parâmetro f é global e conhecido por todos os nós do sistema.

A camada de rede é acessada a partir de duas primitivas: A operação $Send(addr, m)$ envia a mensagem m para o nó de endereço $addr$. A operação $Receive(m)$ aguarda o recebimento de uma mensagem qualquer m. Os canais de comunicação da rede são ponto a ponto e confiáveis, ou seja, não há perda ou alteração de mensagens. O atraso na entrega das mensagens e as diferenças

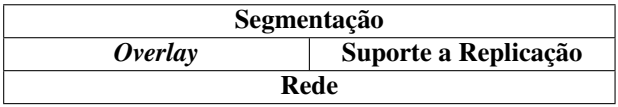

Figura 1: Camadas do sistema

de velocidades entre os nós do sistema respeitam um modelo de sincronia parcial (DWORK; LYNCH; STOCKMEYER, 1988), no qual é garantido a terminação de protocolos de Replicação Máquina de Estados que são usados nas camadas superiores. No entanto, não há garantia de sincronismo por toda a execução.

Imediatamente acima da rede, encontram-se duas camadas independentes que são usadas para construir a camada de segmentação proposta neste trabalho. A camada de *overlay*, descrita na Seção 3.1.1, implementa uma rede P2P sobre o sistema, com busca eficiente de nós distribuídos, e a camada de suporte à replicação, descrita na Seção 3.1.2, que fornece uma abstração de Replicação Máquina de Estados (RME) (SCHNEIDER, 1990) usada para garantir a disponibilidade e consistência das informações contidas no sistema. Em geral, os custos da RME não permitem que essa técnica seja aplicada diretamente a uma grande quantidade de nós, portanto neste trabalho dividimos o sistema em diversas RMEs independentes. A **camada de Segmentação**, proposta na Seção 3.2, faz uso dessas duas camadas e provê meios para invocar eficientemente qualquer RME do sistema. A Figura 3.1 apresenta as camadas do sistema.

A camada de Segmentação é dinâmica, isto é, nós podem entrar e sair do sistema durante a sua execução. Para tal, assumimos a existência de primitivas, nas camadas de Overlay e de Suporte a Replicação, necessárias para suportar o dinamismo da camada de Segmentação. Essas primitivas serão apresentadas em detalhes nas seções a seguir. Conforme será explicado em detalhes na seção 3.3, ainda neste capítulo, é necessário utilizar uma premissa que limita de alguma forma o dinamismo do sistema. Caso contrário, é impossível garantir a terminação das operações providas na camada de segmentação. A premissa utilizada é a mesma descrita em outros sistemas dinâmicos da literatura (p.ex. (AGUILERA et al., 2009)); esta determina que o número de entradas e saídas de nós do sistema seja finito. Será visto mais a frente que essa premissa é, na verdade, mais forte que o necessário, bastando apenas que o sistema passe por períodos de estabilidade (isto é, baixo dinamismo) suficientemente longos para permitir a terminação das operações e o progresso do sistema. No entanto, utilizaremos a premissa mais forte a fim de simplificar a demonstração das propriedades do sistema. Seguindo a classificação de Aguilera (2004), o sistema se encaixa no modelo M_3, isto é,

há infinitos processos em Π, inifinitos processos podem participar em uma mesma execução, porém em um intervalo finito de tempo, apenas um número finito de processos participa do sistema. Esse modelo é denominado **modelo de chegadas infinitas** ou **modelo de concorrência ilimitada**. Apesar disso, assumimos, para simplificar as provas dos algoritmos, que o sistema se comporta conforme o modelo M_2, isto é, Π é infinito porém apenas um número finito de nós participa do sistema em qualquer execução.

3.1.1 Camada de *Overlay*

Acima da camada de rede, assume-se a existência de um *overlay* que provê operações similares às redes P2P estruturadas, como *Pastry* (ROWSTRON; DRUSCHEL, 2001) e *Chord* (STOICA et al., 2001). Essas redes atribuem identificadores aos nós e distribuem estes em torno de um anel lógico. Nós conhecem outros nós com identificadores numericamente próximos, denominados de vizinhos, de forma a manter a estrutura do anel. Além disso, os nós possuem tabelas de roteamento que são usadas para contatar eficientemente nós distantes no anel. Aplicações se utilizam dessa estrutura definindo esquemas para a distribuição de objetos pelo overlay, normalmente atribuindo chaves aos objetos e armazenando esses objetos em nós com identificadores numericamente próximos a essas chaves. A busca por um objeto consiste em usar as tabelas de roteamento para encontrar nós com identificador próximo à chave associada ao mesmo.

Este trabalho utiliza o *overlay* tolerante a faltas bizantinas definido por Castro et al. (2002), o qual apresenta a propriedade de garantir alta probabilidade na entrega de mensagens a todos nós corretos na vizinhança de uma chave, mesmo se uma quantidade de nós do overlay for maliciosa. A confiabilidade da entrega é alcançada pelo envio de uma mensagem por múltiplas rotas e pelo uso de tabelas de roteamento especiais que aumentam a probabilidade de que essas rotas sejam disjuntas, ou seja, não tenham nós em comum. O número de rotas disjuntas, ao superar o limite estabelecido de faltas, fornece a garantia de entrega das mensagens com alta probabilidade. Análises e resultados experimentais (CASTRO et al., 2002) demonstram que a probabilidade de entrega de uma mensagem é de 99,9% se a proporção de nós maliciosos for de até 30%.

O funcionamento do *overlay* depende da geração e atribuição segura de identificadores a nós da rede, de forma que nós maliciosos não possam escolher seu identificador nem participar do *overlay* com múltiplas identidades. Essas propriedades são garantidas, por exemplo, pelo uso de certificados que associam o identificador do nó a seu endereço de rede e sua chave pública.

Esses certificados são emitidos por uma Autoridade Certificadora (AC) confiável, que também pode ser responsável por gerar identificadores aleatórios para os nós[1]. Na nossa infraestrutura, mantemos o uso desses certificados na camada de segmentação, onde são denominados de **certificados de nó**.

Nas seções a seguir, é assumido o uso do overlay definido por Castro et al. (2002), no entanto, outras construções P2P similares podem ser utilizadas sem alteração das camadas superiores da nossa proposta. O overlay deve suportar, então, para entrada e saída de um nó p, operações $OverlayJoin(C_p)$ e $OverlayLeave(C_p)$ que são concretizadas através da apresentação de seu certificado C_p; $OverlaySend(k,m)$ para enviar uma mensagem m para os nós vizinhos da chave k; e $OverlayDeliver(m)$ que entrega a mensagem para a camada superior nos nós de destino.

3.1.2 Camada de Suporte à Replicação

A camada de replicação, que implementa protocolos para Replicação Máquinas de Estados (RME) (SCHNEIDER, 1990), é usada pelas camadas superiores para prover garantias de disponibilidade e consistência das informações mesmo em presença de nós faltosos e maliciosos. Em ambientes com atividades intrusivas, MEs são mantidas através do uso de algoritmos de consenso tolerantes a faltas bizantinas (p.ex. (CASTRO; LISKOV, 2002)). No presente texto não é definido um algoritmo específico de suporte à RME. Qualquer um pode ser escolhido, desde que tolere f nós faltosos em uma composição total mínima de n_{MIN} nós ($n_{MIN} \geq 3f+1$ (LAMPORT; SHOSTAK; PEASE, 1982)).

A camada de replicação oferece às camadas superiores as operações: $TOMulticast(P,m)$ e $TODeliver(m)$. A primeira operação garante o envio de uma mensagem m aos processos do conjunto P e a segunda define a entrega de mensagens segundo ordenação total aos processos de P. As duas operações caracterizam, portanto, um *multicast* com ordenação total (*total order multicast*).

Como as redes P2P são dinâmicas, assume-se o uso de algoritmos com capacidade de reconfiguração, ou seja, algoritmos que permitam a modificação na composição de processos integrantes da RME. Lamport, Malkhi e Zhou (2010) definem formas simples para transformar um modelo de replicação estático em um dinâmico. No presente trabalho, é assumida a operação $TOReconfigure(P')$, que altera o parâmetro local da ME que indica o con-

[1] Não é necessário que esta AC seja uma PKI oficial. A mesma pode ser uma comissão de gestão do sistema, um administrador, etc. O identificador pode ser gerado a partir de uma função *hash* aplicada ao endereço de rede do nó ou à chave pública definida pelo certificado.

junto de processos. A chamada $TOReconfigureOk(P')$ notifica a camada superior o momento em que a chamada $TOReconfigure(P')$ é finalizada. É importante destacar que a chamada $TOReconfigure(P')$ é simplesmente um procedimento local no qual o nó para a execução da máquina de estados, altera a configuração relativa aos nós componentes e inicia a nova ME. Nenhuma mensagem com outros nós é trocada e todas as mensagens enfileiradas para ordenação na ME antiga são descartadas.

3.2 CAMADA DE SEGMENTAÇÃO

A camada de segmentação divide o anel lógico do *overlay* em segmentos compostos de nós contíguos, no qual cada segmento é responsável por um intervalo de chaves do *overlay*. Para fins de disponibilidade, todos os nós do mesmo segmento armazenam o mesmo conjunto de dados replicados. A consistência desses dados é mantida usando Replicação Máquina de Estados reconfigurável, provido pela camada de suporte à replicação.

Os segmentos são dinâmicos, ou seja, suas composições podem mudar com o tempo a partir da entrada e saída de nós. A cada reconfiguração, um novo conjunto de nós (denominado composição ou visão) passa a participar no segmento e a executar os algoritmos associados de suporte à RME. O número de nós de um segmento pode aumentar ou diminuir, logo para evitar que segmentos fiquem com número de nós abaixo do limite de n_{MIN} requerido pelos algoritmos de RME, pode ser necessário unir dois segmentos contíguos em um segmento maior. Por outro lado, o aumento do número de nós de um segmento para um valor muito superior a n_{MIN} pode comprometer o desempenho dos algoritmos de RME. Para aliviar o problema, segmentos grandes podem ser divididos em segmentos menores. É importante notar que reconfigurações ocorrem localmente nos segmentos e não no sistema inteiro.

O número máximo de nós em um segmento é um parâmetro global do sistema denotado n_{MAX}. Quando o segmento supera o número de n_{MAX} nós, é preciso dividir os membros em dois segmentos vizinhos, logo n_{MAX} deve ser maior ou igual a $2 \times n_{MIN}$. Além disso, é necessário adicionar uma tolerância δ ao valor de n_{MAX}, caso contrário, uma única saída (ou entrada) após uma divisão (ou união) de segmentos dispararia uma união (ou divisão). Esse fato poderia ser explorado por nós maliciosos para provocar sucessivas uniões e divisões, deteriorando o desempenho do sistema. Assim sendo, temos que $n_{MAX} = 2 \times n_{MIN} + \delta$. O valor de $\delta \geq 0$ deve ser estabelecido levando em conta a possibilidade de *churn* (malicioso ou não) e o desempenho da RME. Um valor maior de δ aumenta a tolerância ao *churn* e diminui a necessidade de reconfigurar segmentos, porém faz com que segmentos executem com um

número maior de nós, o que pode prejudicar o desempenho da RME. A escolha exata do δ depende de f e também de levantamentos experimentais dos custos não tratados neste trabalho.

Segmentos são descritos por certificados de segmento (S), que contém a lista de todos os certificados de nó (C_i) dos membros do segmento, um contador de configurações ($S.confId$) incrementado a cada reconfiguração da ME e um intervalo de chaves do *overlay* ($K(S) = [S.start, S.end)$) pelas quais o segmento é responsável. Um certificado de segmento possui, ainda, dois campos usados para permitir verificar a sua autenticidade: um conjunto de assinaturas ($S.\Sigma$) e um conjunto de certificados de segmento antigos ($S.history$). As assinaturas são feitas sobre os demais campos do certificado e são emitidas por $f+1$ nós dos segmentos que sofreram a reconfiguração que gerou o segmento S. Dessa forma, o conjunto $S.history$ forma uma cadeia de assinaturas que, partindo de um segmento inicial S_0, conhecido e aceito globalmente, garante a validade do certificado de S. O histórico dos segmentos cresce à medida que o sistema evolui, o que leva a um aumento dos custos relacionados à transferência e validação dos certificados. Na Seção 3.3 apresentamos uma estratégia para reduzir esses custos baseada no uso de um *cache* de certificados de segmento.

A Figura 2 ilustra as reconfigurações ocorridas em uma possível execução da segmentação, sendo $f = 1$, $n_{MIN} = 4$ e $n_{MAX} = 8$. A parte (a) ilustra o grafo induzido pelas reconfigurações, sendo que cada segmento representa um vértice e há uma aresta de um segmento S_i para um segmento S_j se S_j resulta da reconfiguração de S_i. A parte (b) ilustra um detalhamento da entrada e saída dos nós, bem como os intervalos de chaves cobertos pelos segmentos. Com base na execução ilustrada na Figura 2, três situações de reconfiguração distintas são descritas, a seguir:

1. O segmento inicial S_0 inicia a execução no instante t_0 sendo responsável por todas as chaves do *overlay*, ou seja, $K(S_0) = [0, k_{MAX}]$, onde k_{MAX} é a maior chave possível. S_0 possui $confId = 0$, indicando que é o segmento inicial. No instante t_0, S_0 possui 6 membros. Entre os instantes t_0 e t_1, outros 3 nós chegam no sistema, o que leva S_0 a se dividir, uma vez que o número de nós passaria a 9, que é maior que n_{MAX}. Na divisão são gerados os segmentos S_1 e S_2, ambos com $confId = 1$. Os membros de S_0, juntamente com os membros que recém entraram no sistema, são divididos entre os novos segmentos, de forma que S_1 e S_2 possuam pelo menos $n_{MIN} = 4$ nós. O nó atribuído a S_1 que possui o menor identificador, isto é, k_2, determina os intervalos de chaves dos dois novos segmentos: $K(S_2) = [0, k_2)$ e $K(S_1) = [k_2, k_{MAX}]$. Após determinar os parâmetros dos novos segmentos, $f + 1 = 2$ dos 6 nós de S_0 assinam os certificados de S_1 e S_2. Por fim, o certificado de S_0 é in-

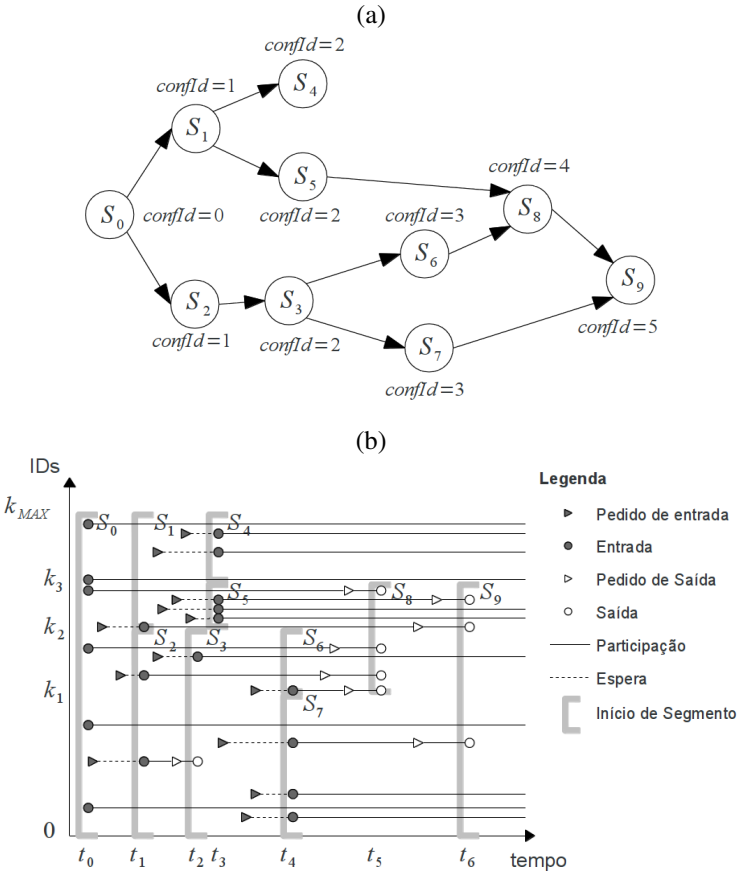

Figura 2: Dinâmica da segmentação em uma possível execução

cluído no histórico de S_1 e S_2, ou seja, $S_1.history = S_2.history = \{S_0\}$. O segmento S_0 termina no instante t_1, ao mesmo tempo em que os segmentos S_1 e S_2 iniciam a atividade. As divisões que ocorrem nos instantes t_3 e t_4 são similares;

2. Entre os instantes t_1 e t_2, um nó chega e um nó parte do segmento S_1, de forma que o número de nós se mantém e a reconfiguração não precisa realizar união ou divisão. Dessa forma, o conjunto de chaves cobertas pelo segmento não se altera pela reconfiguração, ou seja, $K(S_3) = K(S_2) = [0,k_2)$. O parâmetro $confId$ é incrementado, de forma que $S_3.confId = S_2.confId + 1 = 2$. Assim como no caso anterior, os nós de S_2 assinam o certificado de S_3 e S_2 é incluído no histórico de S_3. No instante t_2 o segmento S_2 termina e o segmento S_3 inicia. Somente nesse ponto o nó que pediu para sair do sistema pode parar de executar os protocolos de RME;

3. Entre os instantes t_4 e t_5, três dos quatro nós de S_6 pedem para deixar o segmento. Isso leva S_6 a iniciar uma união a fim de evitar resultar em menos de n_{MIN} nós. Para tal, os nós utilizam o *overlay* para buscar o segmento sucessor de S_6, isto é, o segmento S_5, responsável pela chave k_2. O conjunto de membros resultantes da união é formado pelos membros dos dois segmentos, levando em conta pedidos de entrada e saída de ambos. O conjunto de chaves do segmento resultante S_8 é dado pela união dos conjuntos de chaves de S_5 e S_6, ou seja, $K(S_8) = K(S_5) \cup K(S_6) = [k_1,k_3)$. O valor de $confId$ é o incremento do maior valor nos segmentos S_5 e S_6, de forma que $S_8.confId = \max\{S_5.confId, S_6.confId\} + 1 = \max\{2,3\} + 1 = 4$. Dois dos nós de S_5 e dois dos nós de S_6 assinam o certificado de S_8 e tanto S_5 quanto S_6 são incluídos no histórico de S_8. No instante t_6 ocorre outra união similar.

Essas três situações determinam os tipos básicos de reconfiguração que podem ocorrer e são detalhados nos algoritmos da Seção 3.2.3. Em casos extremos, certos cenários de entrada e saída de nós podem ocorrer de forma que seja necessário divir um segmento em mais de duas partes ou gerar mais de um segmento como resultado da união. Essas situações são melhor explicadas na Seção 3.3. Nenhum argumento perde generalidade por conta dessa possibilidade. Pelos cenários descritos é possível perceber que os nós ativos no sistema sempre pertencem ao segmento responsável pelo seu identificador no *overlay*. Dessa forma, o intervalo de chaves dos segmentos é determinado pelos identificadores dos nós participantes, mas de forma a garantir que todo o espaço de chaves é coberto.

As definições a seguir explicitam os aspectos básicos da segmentação.

Definição 3.1 (Segmento Inicial)**.** *S_0 é o **segmento inicial** do sistema, conhecido e aceito por todos os nós. $S_0.confId = 0$ e $K(S_0) = [0, k_{MAX}]$, onde k_{MAX} é o maior identificador possível no* overlay.

Definição 3.2 (Precedência de Segmentos)**.** *Sejam S e S' dois certificados de segmento válidos e distintos, diz-se que S **precede** S' ($S \prec S'$) se e somente se: (a) $S.confId < S'.confId$; (b) $K(S) \cap K(S') \neq \emptyset$; e (c) S' contém $f+1$ assinaturas de membros de S.*

Definição 3.3 (União e Divisão de Segmentos)**.** *Considerando S, S' e S'' certificados de segmentos válidos e distintos, então: (a) se $S' \prec S$ e $S'' \prec S$, então S é resultado da **união** de S' e S''; (b) se $S \prec S'$ e $S \prec S''$, então S' e S'' são resultado da **divisão** de S.*

Definição 3.4 (Validade de Segmento)**.** *Um certificado de segmento S é **válido** ($Valid(S)$) se é o segmento inicial ou se é gerado a partir da reconfiguração de segmentos válidos, ou seja:*

$$Valid(S) \triangleq S = S_0 \vee \exists S' : (Valid(S') \wedge S' \prec S)$$

Definição 3.5 (Atividade de Segmento)**.** *Um certificado de segmento S é **ativo** no instante t ($Active(t,S)$) se for válido e se a reconfiguração que originou S terminou antes de t e a reconfiguração que ocorre a partir de S não termina antes de t.*

Cada segmento S executa uma RME que é responsável por parte do estado da aplicação, determinado pelo intervalo de chaves $K(S)$. É responsabilidade da aplicação mapear de maneira consistente as chaves do *overlay* em partes do estado, de forma que a qualquer intervalo de chaves corresponda uma parte inequívoca do estado. Esse mapeamento é mantido mesmo com as partes do estado sendo divididas ou unidas conforme as reconfigurações dos segmentos. A camada de segmentação é responsável por transferir essas partes do estado a novos nós sempre que ocorre uma reconfiguração. Apesar disso, o estado da aplicação é manipulado como um objeto opaco, isto é, sua estrutura interna é desconhecida pelos algoritmos da camada de segmentação.

Para invocar uma requisição em uma máquina de estados, um cliente deve primeiro encontrar o segmento responsável pela máquina (usando a camada de *overlay*) e depois enviar a requisição para a máquina (usando a camada de Suporte a Replicação). É responsabilidade da aplicação construir operações pela composição de chamadas de busca e invocação de segmentos. Essa liberdade permite a elaboração de operações complexas na aplicação, envolvendo a coordenação de diversos segmentos, porém expõe certas características da camada de segmentação, como a possibilidade de uma requisição não ser executada devido ao uso de um certificado de segmento antigo.

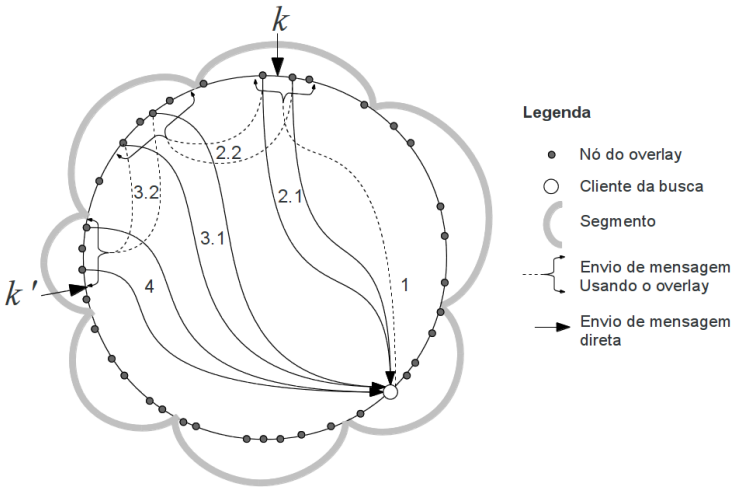

Figura 3: Exemplo de busca de segmentos

A busca de segmentos está baseada nas funcionalidades da camada de *overlay*. O procedimento básico para buscar os segmentos correspondentes a um intervalo de chaves $[k,k']$ consiste em usar o *overlay* para enviar uma mensagem aos nós próximos à chave k. Esses nós respondem ao cliente passando o certificado de segmento atual. Se o intervalo de busca se extender além do limite coberto pelo segmento, os nós repassam o pedido do cliente aos nós do próximo segmento. A Figura 3 ilustra as etapas de uma busca de segmentos: a etapa 1 consiste no envio pelo cliente da mensagem aos nós próximos à chave k; na etapa 2.1 os nós que recebem o pedido enviam o certificado de segmento na resposta ao cliente; como o intervalo de busca vai além do fim do segmento, os nós repassam a requisição da busca para o segmento seguinte (etapa 2.2); um procedimento similar ocorre nas etapas 3.1 e 3.2, ou seja, os nós do próximo segmento recebem a requisição de busca, enviam a resposta ao cliente e repassam para o próximo segmento; a etapa 4 consiste no envio do último certificado de segmento ao cliente, uma vez que este segmento cobre o fim do intervalo de busca.

A Figura 3.2 apresenta as operações de todas as camadas no sistema. As propriedades das operações das camadas inferiores foram descritas na seção 3.1. As propriedades das operações da camada de segmentação são enumeradas a seguir.

A operação $SegJoin(C_p)$ realiza a entrada do nó p na camada de segmentação por meio da apresentação do certificado de nó C_p. Esse certificado é

Segmentação	
$SegJoin(C_p)$, $SegLeave(C_p)$, $SegFind(k,k')$, $SegFindOk(S)$, $SegRequest(S_q,req)$, $SegDeliver(C_p,req)$, $SegResponse(C_p,resp)$ $SegNotify(C_p,m)$, $SegNotifyDeliver(S_q,m)$ $SegReconfigure()$, $SegGetAppState(k,k')$, $SegSetAppState(k,k',\langle state_0,\ldots,state_n\rangle)$	
Overlay	**Suporte a Replicação**
$OverlayJoin(C_p)$, $OverlayLeave(C_p)$, $OverlaySend(k,m)$, $OverlayDeliver(m)$	$TOMulticast(P,m)$, $TODeliver(m)$, $TOReconfigure(P')$, $TOReconfigureOk(P')$
Rede	
$Send(addr,m)$, $Receive(m)$	

Figura 4: Operações das Camadas do Sistema

o mesmo usado na camada de *overlay*, descrito na seção 3.1.1. A propriedade 3.1 (SegJoin), a seguir, trata da correção e terminação da operação *SegJoin*.

Propriedade 3.1 (SegJoin)**.** *Seja p um nó correto com certificado C_p, se p executa $SegJoin(C_p)$ em um instante t, então há um instante $t' \geq t$ no qual existe um segmento S_p ativo que contém p como membro, ou seja:*

$$\exists S_p : Active(t', S_p) \wedge C_p.id \in K(S_p) \wedge C_p \in S_p.members$$

A operação $SegLeave(C_p)$ trata da saída do nó p. O certificado C_p é o mesmo apresentado na entrada. A propriedade 3.2 (SegLeave), a seguir, enuncia a correção e terminação da operação *SegLeave*.

Propriedade 3.2 (SegLeave)**.** *Sejam p um nó correto com certificado C_p e S_p um segmento válido tal que $C_p \in S_p.members$ e $Active(t, S_p)$ para um instante t. Se p executa $SegLeave(C_p)$ em t, então há um instante $t' \geq t$ no qual S_p deixa de ser ativo e nenhum segmento derivado da reconfiguração do mesmo contém p como membro, isto é:*

$$\neg Active(t', S_p) \wedge \forall S', S_p \prec S' : C_p \notin S'.members$$

O operação $SegFind(k,k')$ busca todos os certificados de segmento responsáveis por chaves no intervalo $[k,k']$. Os certificados de segmentos encontrados são repassados à camada superior por meio da chamada *SegFindOk*. A propriedade 3.3 (SegFind) trata da correção e terminação da operação *SegFind*. A propriedade 3.4 (Reconfigurações Finitas SegFind) ga-

rante que se um nó correto executar uma busca de certificados infinitas vezes, em algum momento o resultado da busca estabiliza e os segmentos retornados permanecem ativos. Essa propriedade é importante para assegurar a terminação das operações de entrada e saída de nós (como será visto nas provas das propriedades destas), mas também das operações de aplicação, uma vez que estas acabam por realizar invocações em segmentos encontrados por um *SegFind*. Esta propriedade está diretamente relacionada com o limite imposto pela premissa de que um número finito de entradas e saídas ocorre no sistema, e da mesma forma, na prática é necessário apenas que os segmentos permaneçam estáveis por tempo suficiente para o processamento de requisições.

Propriedade 3.3 (SegFind)**.** *Se um nó correto p executa $SegFind(k,k')$, esta operação acaba por retornar (por meio de notificações $SegFindOk(S)$) um conjunto $\mathbb{S}$ de certificados de segmento válidos e que cobrem totalmente o intervalo da busca, isto é:*

$$(\forall S \in \mathbb{S} : Valid(S)) \wedge [k,k'] \subseteq \bigcup_{S \in \mathbb{S}} K(S)$$

Propriedade 3.4 (Reconfigurações Finitas SegFind)**.** *Existe um instante t tal que toda chamada $SegFind(k,k')$ executada por um nó correto p em $t' \geq t$ retorna um conjunto $\mathbb{S}$ de certificados de segmentos que permanecem ativos indefinidamente, ou seja:*

$$\forall t'' \geq t', S \in \mathbb{S} : Active(t'', S)$$

A operação $SegRequest(S_q, req)$ envia a requisição *req* ao segmento S_q usando a camada de suporte à replicação e retorna a resposta recebida. Se o certificado S_q for obsoleto, a operação retorna um valor distinto $\perp$. Caso contrário, a operação $SegDeliver(C_p, req)$ entrega a requisição nos nós do segmento de destino seguindo uma ordenação total. A operação $SegResponse(C_p, resp)$ simplesmente envia a resposta ao cliente usando a camada de rede diretamente. As propriedades 3.5 (Validade SegRequest), 3.6 (Acordo SegRequest) e 3.7 (Ordenação SegRequest) correspondem às propriedades de validade, acordo e ordem atômica do algoritmo de multicast com ordenação total utilizado na camada de suporte à replicação. A propriedade 3.8 (Terminação SegRequest) está relacionada à terminação de operações que façam uso de *SegRequest*, e é complementar à propriedade 3.4 (Reconfigurações Finitas SegFind). A propriedade 3.9 (Segmento Inativo SegRequest) garante que nós corretos não entregarão mensagens destinadas a segmentos inativos.

Propriedade 3.5 (Validade SegRequest)**.** *Sejam p e q nós corretos tais que C_p é o certificado de nó de p e q pertence a um segmento ativo S_q. Se q recebe o* upcall *$SegDeliver(C_p, req)$, então p executou $SegRequest(S_q, req)$.*

Propriedade 3.6 (Acordo SegRequest)**.** *Seja q um nó correto de um segmento ativo S_q. Se q recebe o* upcall *$SegDeliver(C_p, req)$, então todo nó correto de S_q também recebe o mesmo* upcall *$SegDeliver(C_p, req)$.*

Propriedade 3.7 (Ordenação SegRequest)**.** *Seja q um nó correto de um segmento ativo S_q. Se q recebe $SegDeliver(C_i, req_1)$ antes de $SegDeliver(C_j, req_2)$, então todo nó correto de S_q também recebe $SegDeliver(C_i, req_1)$ antes de $SegDeliver(C_j, req_2)$.*

Propriedade 3.8 (Terminação SegRequest)**.** *Seja p um nó correto. Existe um instante t tal que, se S_q é um certificado de segmento obtido por uma chamada $SegFind(k, k')$ executada em $t' \geq t$, então uma requisição req enviada por uma chamada $SegRequest(S_q, req)$, executada por p, acabará por ser entregue por meio de $SegDeliver(C_p, req)$ em todos os nós corretos de S_q.*

Propriedade 3.9 (Segmento Inativo SegRequest)**.** *Seja p um nó com certificado C_p. Se p executa $SegRequest(S_q, req)$ no instante t e S_q não está ativo nesse instante ($\neg Active(t, S_q)$), então nenhum nó correto de S_q recebe o* upcall *$SegDeliver(C_p, req)$.*

A chamada *SegResponse* deve ser enviada pelos nós de um segmento para que uma operação *SegRequest* termine retornando a resposta correta. Apesar disso, em certas aplicações pode ser conveniente possibilitar aos nós do segmento enviar notificações assíncronas a um cliente, por exemplo em operações cuja execução pode levar mais tempo que o esperado pelo cliente[2]. Para isso, definimos a primitiva $SegNotify$, que envia uma mensagem diretamente a um cliente. No lado do cliente, uma notificação recebida é representada pela chamada $SegNotifyDeliver$. A operação $SegNotify$ é bastante simples, consistindo no envio direto de uma mensagem a um nó, utilizando para isso a camada de rede. No nó cliente, qualquer mensagem recebida de pelo menos $f+1$ nós de um mesmo segmento é repassada com $SegNotifyDeliver$. Juntamente com a confiabilidade dos canais de comunicação, isso garante a integridade, validade e entrega das notificações. Devido a simplicidade, não formalizamos as propriedades da operação de notificação, no entanto os algoritmos são apresentados na Seção 3.2.1.

A operação $SegReconfigure()$ é um procedimento interno da camada de segmentação, executado em intervalos de tempo determinados, que efetiva

[2]conforme descrito mais a frente na Seção 3.2.1, um cliente precisa usar um temporizador para sair da espera em casos onde o segmento invocado sofreu uma reconfiguração antes de executar a requisição do cliente

as entradas e saídas de nós conforme as chamadas de *SegJoin* e *SegLeave*. É nessa operação também que as uniões e divisões de segmentos são efetivadas. Apesar de ser um procedimento interno e não haver nenhuma propriedade pública imediatamente correspondente, a operação *SegReconfigure* apresenta características enunciadas na forma de lemas que contribuem para a prova das outras propriedades do sistema.

As operações *SegGetAppState* e *SegSetAppState* servem, respectivamente, para obter e alterar o estado da aplicação que executa acima da camada de segmentação. Na primeira, a camada de segmentação obtém o estado (ou parte deste), porém somente os valores associados às chaves no intervalo $[k,k']$. Já na operação $SegSetAppState(k,k',\langle state_1,\ldots,state_n\rangle)$, a camada de segmentação altera o intervalo de chaves coberto para $[k,k']$ e atualiza o estado da aplicação para ser a união das partes $state_i$. Quando ocorre uma união de segmentos, as partes do estado da aplicação dos segmentos antigos devem ser unidas no segmento novo. Nesse caso, a camada de segmentação passa mais de um valor $state_i$ para a aplicação para que essa camada realize a união do estado. A própria camada de segmentação não é capaz de realizar essa união do estado porque não conhece a estrutura interna do mesmo.

As propriedades 3.5 (Validade SegRequest), 3.6 (Acordo SegRequest), 3.7 (Ordenação SegRequest), 3.8 (Terminação SegRequest) e 3.9 (Segmento Inativo SegRequest) garantem que dentro de uma mesma Máquina de Estados, isto é, durante o período em que um segmento S está ativo, todos os nós corretos de S executam as mesmas operações na mesma sequência, logo, se todos os nós partem de um mesmo estado inicial (imediatamente após a reconfiguração que resultou em S), alcançam o mesmo estado final (imediatamente antes da reconfiguração que termina S). A propriedade 3.10 (Consistência do Estado), a seguir, garante que o estado global da aplicação permanece inalterado na ocorrência de reconfigurações. Essa propriedade não está diretamente relacionada a nenhuma operação pública da camada de segmentação, no entanto assegura um invariante crucial para o correto funcionamento da infraestrutura.

Propriedade 3.10 (Consistência do Estado)**.** *Sejam S_i, $i \in [1,n]$, os segmentos envolvidos em uma reconfiguração que gera os segmentos S'_j, $j \in [1,m]$: se $appState$ é a união dos estados armazenados nos segmentos S_i imediatamente antes da reconfiguração iniciar e $appState'$ é a união dos estados armazenados em S'_j imediatamente após o término da reconfiguração, então $appState = appState'$*[3].

[3] A união e comparação de igualdade dos estados são utilizados de maneira informal nesta propriedade. A ideia não é garantir a identidade dos estados, mas indicar que nada se perde na transferência de estado realizada durante as reconfigurações

Tabela 1: Notações utilizadas nos algoritmos

Notação	Significado
$\#A$	número de elementos do conjunto A
$\#_b A$	número de elementos do conjunto A iguais a b
$K(S)$	$[S.start, S.end)$
$m \leftarrow \langle \alpha \rangle \sigma_i$	$\sigma_i \leftarrow Sign(\langle \alpha \rangle)$ $m \leftarrow \langle \alpha, \sigma_i \rangle$
$C_i \in S_q$	pertinência de C_i no segmento S_q $(C_i \in S_q.members)$

As seções a seguir apresentam os algoritmos da camada de segmentação. Cada algoritmo é apresentado na forma de pseudocódigo, acompanhado de uma descrição textual. Após a descrição de todos os algoritmos, seguem as provas de lemas e teoremas relacionados às propriedades do sistema. Sugerimos que o leitor primeiro entenda o funcionamento de todos os algoritmos para somente então iniciar a leitura das provas.

A Tabela 1 apresenta algumas notações e variáveis globais utilizadas nos algoritmos. Abaixo são descritas as estruturas de dados mantidas pelos nós do sistema e acessíveis globalmente a todos os algoritmos.

- $C_p = \langle addr, pubK, id \rangle \sigma_{CA}$: é o certificado do nó p, no qual $addr$ é o endereço de rede do mesmo, $pubK$ e id são, respectivamente, a chave pública e o identificador de p no *overlay*. Esse certificado é o mesmo utilizado na camada de *overlay*;

- $privK_p$: é a chave privada do nó p correspondente à chave pública presente em C_p e é usada em assinaturas digitais pelo nó. Assume-se que mensagens recebidas com assinaturas inválidas não são processadas por nós corretos;

- $S_p = \langle members, confId, start, end, \Sigma, history \rangle$: é o certificado do segmento atual de um nó p, no qual $members$ é o conjunto dos certificados de nó dos membros atuais do segmento, $confId$ é o contador de configurações, $[start, end) = K(S_p)$ é o intervalo de chaves do *overlay* pelas quais o segmento é responsável, Σ é um conjunto de assinaturas e $history$ é a cadeia de certificados representando o caminho de evolução do segmento atual;

- *notifs*: conjunto que armazena notificações recebidas pelo nó, vindas de segmentos.

- *changes*: conjunto de alterações na lista de membros a serem aplicadas na próxima reconfiguração. A entrada do nó *i* é denotada por $\langle +, i \rangle$ e a saída por $\langle -, i \rangle$;

- *reconfCount*: contador de nós que solicitam reconfiguração na configuração atual;

3.2.1 Busca e Invocação de Segmentos

Para invocar uma RME, um nó precisa primeiro obter o certificado do segmento que executa essa máquina de estados. A busca de segmentos é realizada pela operação *SegFind* (Algoritmo 1), que tem como parâmetro de entrada um intervalo de chaves e retorna um conjunto de certificados dos segmentos responsáveis pelas chaves nesse intervalo. A operação encontra certificados fazendo primeiro uma busca no *overlay* pela primeira chave do intervalo (linha 4). Os nós que receberem a mensagem de busca pelo *overlay* respondem enviando seus certificados de segmento (linha 17) e repassando a mesma para segmentos vizinhos caso o intervalo de chaves buscado se estenda além do seu próprio segmento (linhas 18 a 20). A cada certificado de segmento recebido pelo cliente (linha 6), a operação chama a função $Valid(S)$, que verifica se a cadeia de segmentos que acompanha *S* é válida (linha 7). Se o encadeamento e as assinaturas forem válidos, a operação verifica se foram recebidos $f+1$ certificados idênticos (linha 9, atualiza a cobertura de chaves (linha 10) e invoca $SegFindOk(S)$ para notificar a camada superior (linha 11). A operação no cliente termina quando todo o intervalo de busca for coberto pelos certificados de segmento recebidos (teste da linha 5).

De posse de um certificado de segmento, o nó pode executar a operação *SegRequest* (Algoritmo 2) fazendo uso do suporte à RME. Essa operação consiste em iniciar um temporizador (linha 2), invocar o segmento usando a primitiva $TOMulticast$ (linha 5), passando o $confId$ do certificado que o cliente conhece (linha 3), e aguardar $f+1$ respostas idênticas de membros distintos (linhas 6 a 13). Se o certificado conhecido pelo cliente for antigo (linha 18), os servidores não repassarão a requisição para a aplicação e o cliente não receberá $f+1$ respostas, o que provocará o estouro do temporizador e a operação irá retornar $\perp$, indicando uma exceção (linhas 14 e 16). A operação não trata das exceções e deixa essa responsabilidade para as camadas superiores. Se o certificado de segmento usado na invocação for atual, a requisição é entregue para a camada superior pela chamada *SegDeliver* (linha 19). As

Algoritmo 1 Busca de segmentos (*SegFind*)

```
1:  operation SegFind(k, k')
    /* Código do cliente p */
        /* Chaves cobertas pelos certificados recebidos */
2:      keys ← ∅
        /* Lista de certificados recebidos */
3:      resps ← ∅
        /* Envia mensagem assinada usando o overlay */
4:      OverlaySend(k, ⟨FIND, C_p, k, k'⟩σ_p)
        /* Testa cobertura de chaves recebidas */
5:      while [k, k'] ⊈ keys do
            /* Aguarda resposta de algum servidor q */
6:          wait for Receive(⟨FIND_OK, C_q, S_q⟩σ_q)
            /* Verifica validade do segmento */
7:          if K(S_q) ∩ [k, k'] ≠ ∅ ∧ Valid(S_q) then
8:              resps ← resps ∪ {⟨C_q, S_q⟩}
9:              if #_{⟨*, S_q⟩} resps = f + 1 then
                    /* Atualiza cobertura de chaves */
10:                 keys ← keys ∪ K(S_q)
                    /* Notifica segmento encontrado */
11:                 SegFindOk(S_q)
12:             end if
13:         end if
14:     end while
    /* Código do servidor q */
15:     upon OverlayDeliver(⟨FIND, C_p, k, k'⟩σ_p) do
16:         if [k, k'] ∩ K(S_q) ≠ ∅ then
                /* Envia resposta assinada */
17:             Send(C_p.addr, ⟨FIND_OK, C_q, S_q⟩σ_q)
                /* Testa se segmento não cobre intervalo */
18:             if k' ≥ S_q.end then
                    /* Repassa mensagem de busca ao segmento sucessor */
19:                 OverlaySend(S_q.end, ⟨FIND, C_p, k, k'⟩σ_p)
20:             end if
21:         end if
22:     end upon
23: end operation
```

Algoritmo 2 Invocação de segmentos (*SegRequest*)

```
1:  operation SegRequest(S_q, req)
    /* Código do cliente p */
        /* Inicia temporizador */
2:      StartTimer(req)
        /* Identificador de configuração conhecido por p */
3:      knownId ← S_q.confId
        /* Conjunto de respostas recebidas */
4:      resps ← ∅
        /* Envia requisição com ordenação total */
5:      TOMulticast(S_q.members, ⟨REQ, C_p, knownId, req⟩σ_p)
        /* Resposta recebida de q */
6:      upon Receive(⟨RESP, C_q, resp_q⟩σ_q) do
7:          if C_q ∈ S_q then
8:              resps ← resps ∪ {resp_q}
                /* Testa se recebeu f + 1 respostas iguais */
9:              if #_{resp_q} resps = f + 1 then
10:                 return resp_q
11:             end if
12:         end if
13:     end upon
        /* Menos de f + 1 resposta recebidas antes do timeout */
14:     upon Timeout(req) do
15:         return ⊥
16:     end upon
    /* Código do servidor q */
17:     upon TODeliver(⟨REQ, C_p, knownId, req⟩σ_p) do
            /* Testa se cliente conhece confId atual */
18:         if knownId = S_q.confId then
                /* Entrega requisição para aplicação */
19:             SegDeliver(C_p, req)
20:         end if
21:     end upon
22: end operation

23: operation SegResponse(C_p, resp_q)
    /* Código do servidor q */
        /* Envia resposta diretamente ao cliente */
24:     Send(C_p.addr, ⟨RESP, C_q, resp_q⟩σ_q)
25: end operation
```

respostas a requisições são enviadas pela operação *SegResponse*, que consiste no envio da mensagem de resposta diretamente ao cliente (linha 24).

A operação *SegRequest* utiliza um temporizador para garantir que o cliente não ficará aguardando indefinidamente por uma resposta no caso em que um certificado de segmento ultrapassado é utilizado. Nesse contexto, pode acontecer, no entanto, de o temporizador estourar, não porque o segmento invocado reconfigurou, mas sim porque a operação executada levou mais tempo que o tempo de espera estipulado. Para evitar esse problema, definimos a operação *SegNotify* que pode ser utilizada para enviar respostas assíncronas a um cliente. Se uma operação da aplicação pode levar um longo tempo para terminar, é possível enviar uma resposta, usando *SegResponse*, imediatamente após o recebimento da requisição. Essa resposta, no entanto, não contém o resultado da requisição, mas uma indicação de que o resultado será enviado posteriormente por meio de uma notificação. Dessa forma, o cliente termina a espera com a garantia de que a requisição foi efetivamente ordenada na Máquina de Estados e passa a aguardar (sem nenhum temporizador) uma notificação contendo a resposta da requisição.

Algoritmo 3 Notificação de nós (*SegNotify*)

```
1:  operation SegNotify(C_p, m_q)
    /* Código do servidor q */
        /* Envia mensagem diretamente ao nó p usando a rede */
2:      Send(C_p.addr, ⟨NOTIFY, C_q, S_q, m_q⟩σ_q)

    /* Código do cliente p */
3:      upon Receive(⟨NOTIFY, C_q, S_q, m_q⟩σ_q) do
4:          if Valid(S_q) ∧ C_q ∈ S_q then
                /* Acumula mensagem recebida */
5:              notifs ← notifs ∪ {⟨NOTIFY, C_q, S_q, m⟩σ_q}
                /* Verifica se recebeu f + 1 mensagens iguais */
6:              if #_⟨NOTIFY, C_i, S_q, m_q⟩σ_i notifs = f + 1 then
                    /* Entrega notificação para camada superior */
7:                  SegNotifyDeliver(S_q, m_q)
                    /* Limpa acumulador para evitar múltiplas entregas */
8:                  notifs ← notifs \ {⟨NOTIFY, C_i, S_i, m_i⟩σ_i :
                                                S_i = S_q ∧ m_i = m_q}
9:              end if
10:         end if
11:     end upon
12: end operation
```

A operação *SegNotify* está descrita no Algoritmo 3. No lado do servidor, a operação consiste simplesmente em enviar uma mensagem assinada diretamente ao cliente usando a rede (linha 2). O cliente, ao receber uma mensagem de notificação (linha 3), acumula essa mensagem no conjunto $notifs$ (linha 5) e verifica se $f+1$ mensagens iguais de nós do mesmo segmento foram recebidas (linha 6). Em caso positivo, a mensagem é entregue à aplicação (linha 7) e o conjunto $notifs$ é limpo de maneira a evitar a re-entrega da mesma mensagem mais de uma vez (linha 8).

Não definimos nem provamos nenhuma propriedade específica para a operação *SegNotify*, no entanto é fácil perceber que basta que haja $f+1$ nós corretos no segmento para garantir que a notificação chegará ao nó de destino. Além disso, a verificação da validade do segmento (linha 4), juntamente com o teste de $f+1$ recebimentos (linha 6) garantem que nós corretos não podem enviar notificações falsas.

3.2.2 Entrada e Saída de Nós

A entrada do nó p na camada de segmentos se dá pela operação *SegJoin*(C_p) (Algoritmo 4), na qual C_p é o certificado de nó de p. Antes de entrar em algum segmento, p invoca *OverlayJoin*(C_p) e entra no *overlay* (linha 2). Depois, p busca o certificado do segmento responsável por seu identificador $C_p.id$ (linha 5) e invoca o mesmo passando uma mensagem *JOIN* (linha 7). Após obter uma resposta válida, p aguarda o recebimento do certificado de segmento e do estado da aplicação (linha 9), enviados por $f+1$ membros do segmento. O estado é repassado à camada superior (linha 11) e a operação $TOReconfigure$ é chamada (linha 12), alterando os nós participantes da RME para o novo segmento de p. Quando os membros do segmento recebem a mensagem *JOIN* do nó p (linha 13), simplesmente registram o pedido de p no conjunto *changes* (linha 15) e respondem (linha 16). A reconfiguração ocorre posteriormente, na operação *SegReconfigure*, conforme descrito no Algoritmo 6.

A saída de nós é realizada pela operação *SegLeave* (Algoritmo 5). O nó envia uma mensagem $LEAVE$ (linha 4) para os membros do seu segmento e continua a atender requisições até o término da próxima reconfiguração, notificada pela operação $TOReconfigureOk$ (linha 6). Depois o nó para de atender requisições e invoca *OverlayLeave* (linha 7). Nós corretos são obrigados a aguardar a reconfiguração para garantir o funcionamento dos algoritmos de replicação. De maneira similar à operação de entrada, os nós que recebem a mensagem $LEAVE$ registram o pedido de (linha 10) e a reconfiguração propriamente dita se dá pela operação *SegReconfigure* (Algoritmo 6).

Algoritmo 4 Entrada de nós (*SegJoin*)

```
1:  operation SegJoin(C_p)
    /* Código do cliente p */
        /* Entra no overlay */
2:      OverlayJoin(C_p)
3:      resp ← ⊥
        /* Repete até receber resposta de confirmação */
4:      while resp = ⊥ do
            /* Busca segmento de entrada */
5:          SegFind(C_p.id, C_p.id)
6:          wait for SegFindOk(S_q)
            /* Envia requisição ordenada para entrada */
7:          resp ← SegRequest(S_q, ⟨JOIN, C_p⟩)
8:      end while
        /* Transferência do estado da aplicação */
        /* Recebe estado e segmento novo enviados pelos membros do segmento antigo (Algoritmo 8, linha 8; Algoritmo 9, linhas 14 e 17; Algoritmo 10, linhas 27 e 15) */
9:      wait for SegNotifyDeliver(⟨STATE, S_q, appState_q⟩)
        /* Atualiza segmento */
10:     S_p ← S_q
        /* Atualiza estado da aplicação */
11:     SegSetAppState(K(S_q), appState_q)
        /* Reconfigura algoritmo de RME */
12:     TOReconfigure(S_p.members)

    /* Código do servidor q */
13:     upon SegDeliver(C_p, ⟨JOIN, C_p⟩) do
14:         if C_p.id ∈ K(S_q) then
                /* Registra pedido de entrada e envia confirmação */
15:             changes ← changes ∪ {⟨+, C_p⟩}
16:             SegResponse(C_p, ⟨JOIN_OK⟩)
17:         end if
18:     end upon
19: end operation
```

Algoritmo 5 Saída de nós (*SegLeave*)

```
1:  operation SegLeave(C_p)
    /* Código do cliente p */
2:      resp ← ⊥
        /* Repete até requisição ser confirmada (S_p pode se tornar obsoleto
    por causa de uma reconfiguração concorrente) */
3:      while resp = ⊥ do
            /* Envia requisição de saída */
4:          resp ← SegRequest(S_p, ⟨LEAVE, C_p⟩)
5:      end while
        /* Aguarda reconfiguração do algoritmo de RME (Algoritmo 8, linha
    10; Algoritmo 9, linha 24; Algoritmo 10, linhas 29 e 17 ) */
6:      wait for TOReconfigureOk(S)
        /* Sai do overlay */
7:      OverlayLeave(C_p)

    /* Código do servidor q */
8:      upon SegDeliver(C_p, ⟨LEAVE, C_p⟩) do
9:          if C_p ∈ S_q then
                /* Registra pedido de saída e envia confirmação */
10:             changes ← changes ∪ {⟨−, C_p⟩}
11:             SegResponse(C_p, ⟨LEAVE_OK⟩)
12:         end if
13:     end upon
14: end operation
```

3.2.3 Reconfiguração de Segmentos

A reconfiguração de um segmento pode ocorrer por dois eventos distintos: (a) quando $f+1$ nós executam a operação *SegReconfigure* (Algoritmo 6); ou (b) quando o segmento é requisitado a se unir com seu vizinho. No caso (a) a operação *SegReconfigure* é invocada após certo tempo, indicado pelo estouro do temporizador *ReconfigTimeout* (linha 2). Nesse momento, caso haja entradas ou saídas para tratar, o nó assina e dissemina no segmento uma mensagem de tentativa de reconfiguração (linha 3). Essas tentativas de reconfiguração (recepções de mensagens *TRY_RECONFIG*) são acumuladas pelos nós do segmento e quando algum nó recebe $f+1$ dessas tentativas (linha 8), dispara uma requisição ordenada para a reconfiguração, enviando juntamente as tentativas assinadas como prova (linha 9). Como as tentativas não são ordenadas, é possível que a requisição de reconfiguração seja invocada mais de uma vez, porém o teste da linha 14 garante que a reconfiguração de fato somente ocorre uma vez por segmento. Além disso, a reconfiguração é ordenada juntamente com os pedidos de entrada e saída, o que garante que todos os nós corretos possuem o mesmo conjunto *changes* e, portanto, calcularão o mesmo conjunto de membros.

O código da reconfiguração consiste inicialmente em calcular o novo conjunto de membros (linha 15) e checar o tamanho deste a fim de detectar se será necessário dividir ou unir segmentos, de acordo com o tamanho do conjunto e com os parâmetros globais n_{MIN} e n_{MAX}. Se o número de membros for inferior a n_{MIN}, o procedimento *Merge* é invocado (linha 16), que efetua a união do segmento (Algoritmo 10). Se o número de membros for superior a n_{MAX}, o procedimento *Split* é invocado (linha 17), que realiza a divisão do segmento (Algoritmo 9). Caso o número de membros esteja entre n_{MIN} e n_{MAX}, ocorre uma reconfiguração simples (Algoritmo 8), efetuada pela chamada do procedimento *SimpleReconfig* (linha 18). Após a realização da reconfiguração, o conjunto *changes* de entradas e saídas é esvaziado (linha 20), bem como o contador de tentativas de reconfiguração (linha 21).

O caso (b) ocorre quando um segmento, ao reconfigurar, determina que o número de nós resultante das entradas e saídas será menor que n_{MIN}. Nessa situação, nós desse segmento geram e assinam pedidos de união que são enviados em uma invocação ordenada ao segmento sucessor no *overlay*. No segmento sucessor, esses pedidos são validados e se houver $f+1$ pedidos válidos de nós distintos, o segmento sucessor dá início ao procedimento de união propriamente dito. O procedimento de união completo é explicado na Seção 3.2.3.3.

As subseções 3.2.3.1, 3.2.3.2 e 3.2.3.3, a seguir, apresentam, respectivamente, os algoritmos dos procedimentos de reconfiguração simples, di-

Algoritmo 6 Reconfiguração de segmentos (*SegReconfigure*)

```
1:  operation SegReconfigure()
    /* Código no cliente p */
        /* Estouro do temporizador de reconfiguração */
2:      upon Timeout(RECONF) ∧ changes ≠ ∅ do
            /* Envia tentativa de reconfiguração */
3:          Send(C_i.addr, ⟨TRY_RECONF, C_p, S_p⟩σ_p), ∀C_i ∈ S_p
4:      end upon
        /* Recebimento de tentativa de reconfiguração */
5:      upon Receive(⟨TRY_RECONF, C_i, S_i⟩σ_i) do
            /* Testa parâmetros da tentativa */
6:          if C_i ∈ S_p ∧ S_i = S_p then
7:              reconfCount ← reconfCount ∪ {⟨TRY_RECONF, C_i, S_i⟩σ_i}
                /* Testa se recebeu f + 1 tentativas e inicia reconfiguração */
8:              if #reconfCount ≥ f + 1 then
9:                  TOMulticast(S_p.members,
                                ⟨RECONF, C_p, reconfCount⟩σ_p)
10:             end if
11:         end if
12:     end upon

    /* Código do servidor q */
13:     upon TODeliver(⟨RECONF, C_p, reconfCount_p⟩σ_p) do
            /* Valida conjunto de tentativas */
14:         if #reconfCount_p ≥ f + 1 ∧ ∀⟨TRY_RECONF, C_i, S_i⟩σ_i
                    ∈ reconfCount_p : C_i ∈ S_i ∧ S_i = S_q then
                /* Calcula novo conjunto de membros */
15:             newMembers ← (S_q.members ∪ {C_i : ⟨+, C_i⟩ ∈ changes}) \
                        {C_i : ⟨−, C_i⟩ ∈ changes}
                /* Testa necessidade de união ou divisão */
16:             if #newMembers < n_MIN then Merge(newMembers)
17:             else if #newMembers > n_MAX then Split(newMembers)
18:             else SimpleReconfig(newMembers)
19:             end if
                /* Limpa entradas/saídas e tentativas de reconfiguração */
20:             changes ← ∅
21:             reconfCount ← ∅
22:         end if
23:     end upon
24: end operation
```

visão de segmento e união de segmentos. Antes disso, porém, descrevemos um procedimento utilizado nos três tipos de reconfiguração, especificamente a geração, disseminação e coleta de assinaturas sobre certificados de segmentos novos. Esse procedimento, *ExchSigs* (Algoritmo 7) é utilizado justamente no processo de geração de um novo certificado de segmento.

O procedimento *ExchSigs* (Algoritmo 7) recebe uma lista de n certificados de segmento não assinados $S^i_{uns}, i \in [1,n]$ e devolve, para cada certificado novo S^i_{uns}, um conjunto Σ_i contendo $f+1$ assinaturas de membros distintos do segmento antigo. Essa generalização quanto ao número de certificados assinados permite que o procedimento seja utilizado tanto na reconfiguração simples quanto na união ou divisão de segmentos, independentemente do número de segmentos resultantes. Implicitamente, assumimos que sempre que o procedimento *ExchSigs* é utilizado por um nó correto, todos os demais nós corretos do mesmo segmento acabam por executar *ExchSigs* passando a mesma lista de certificados não assinados, de forma a permitir a execução correta e a terminação. Essa característica será explorada em detalhes na demonstração do Lema 3.6, Seção 3.2.4.

O primeiro passo é assinatura de cada segmento pelo nó q (linha 3). Na sequência, essas assinaturas são concatenadas em uma mensagem que é enviada, usando os canais ponto a ponto da camada de rede, para todos os nós do mesmo segmento do nó q (linha 6). As mensagens disseminadas são coletadas (linha 10) até que $f+1$ assinaturas válidas (linha 12) sobre cada certificado sejam recebidas. Por fim, esses conjuntos de $f+1$ assinaturas são retornados (linha 17).

3.2.3.1 Reconfiguração Simples

Uma reconfiguração simples (Algoritmo 8) consiste em gerar um novo certificado de segmento, levando em conta as entradas e saídas registradas no conjunto *changes* pelas operações *SegJoin* (Algoritmo 4) e *SegLeave* (Algoritmo 5), transferir o estado para os novos membros e alterar a composição da máquina de estados para iniciar o novo segmento. Como na reconfiguração simples o número de nós participantes do novo segmento está dentro dos limites n_{MIN} e n_{MAX}, então não ocorre união nem divisão e o intervalo de chaves do segmento continua o mesmo.

Inicialmente, o procedimento $SimpleReconfig$ calcula um novo identificador de configuração incrementando o identificador do segmento atual (linha 2). O conjunto *newMembers*, recebido como parâmetro, e o intervalo de chaves, que é o mesmo do segmento atual, são concatenados com o novo identificador para formar um segmento não assinado que é passado para o

Algoritmo 7 Assinatura de certificados de segmento (*ExchSigs*)

1: **procedure** $ExchSigs(S^1_{uns}, \ldots, S^n_{uns})$
   /* Código do servidor q */
   /* Gera assinaturas locais sobre os certificados */
2: **for all** $i \in [1,n]$ **do**
3: $\quad \sigma^i_q \leftarrow Sign(\langle S^i_{uns}.members, S^i_{uns}.confId, K(S^i_{uns})\rangle)$
4: **end for**
   /* Dissemina assinaturas para os demais membros do segmento */
5: **for all** $C_i \in S_q$ **do**
6: $\quad Send(C_i.addr, \langle NEW\_SIG, C_q, \sigma^1_q, \ldots, \sigma^n_q\rangle)$
7: **end for**
8: $\Sigma^i \leftarrow \emptyset, \forall i \in [1,n]$
   /* Repete até receber $f+1$ assinaturas válidas para cada segmento */
9: **while** $\exists i \in [1,n] : \#\Sigma^i < f+1$ **do**
   /* Recebe mensagem contendo assinaturas */
10: $\quad$ **wait for** $Receive(\langle NEW\_SIG, C_j, \sigma^1_j, \ldots, \sigma^n_j\rangle)$
   /* Valida assinaturas recebidas */
11: $\quad$ **for all** $i \in [1,n] : \#\Sigma^i < f+1$ **do**
12: $\quad\quad$ **if** $ValidSig(\sigma^i_j, \langle S^i_{uns}.members, S^i_{uns}.confId, K(S^i_{uns})\rangle$ **then**
13: $\quad\quad\quad \Sigma^i \leftarrow \Sigma^i \cup \{\sigma^i_j\}$
14: $\quad\quad$ **end if**
15: $\quad$ **end for**
16: **end while**
   /* Retorna conjuntos de $f+1$ assinaturas */
17: **return** $\langle \Sigma^1, \ldots, \Sigma^n \rangle$
18: **end procedure**

procedimento *ExchSigs* (linha 3). Esse procedimento gera, dissemina e coleta $f+1$ assinaturas dos demais membros do segmento atual. O histórico do novo segmento é caculado incluindo o segmento atual (linha 4). Na sequência, o novo segmento é montado pela concatenação dos parâmetros calculados e das assinaturas (linha 5). O estado da aplicação é obtido pela chamada *SegGetAppState* (linha 6) e envia aos novos membros, conforme indicado no conjunto *changes* (linha 8). Por fim, os membros do segmento atual entram na nova Máquina de Estados usando a chamada $TOReconfigure$ (linha 10).

Algoritmo 8 Reconfiguração simples (sem união ou divisão) de segmento ($SimpleReconfig$)

```
 1: procedure SimpleReconfig(newMembers)
    /* Código do servidor q */
        /* Calcula novo identificador de configuração */
 2:     newConfId ← S_q.confId + 1
        /* Gera e dissemina assinatura (Algoritmo 7) */
 3:     newΣ ← ExchSigs(⟨newMembers, newConfId, K(S_q)⟩)
        /* Inclui certificado antigo no histórico do novo */
 4:     newHistory ← S_q.history ∪ {S_q}
        /* Gera novo certificado assinado */
 5:     S_q ← ⟨newMembers, newConfId, K(S_q), newΣ, newHistory⟩
        /* Obtém estado da aplicação */
 6:     appState_q ← SegGetAppState(K(S_q))
        /* Envia estado aos novos membros */
 7:     for all C_i : ⟨+, C_i⟩ ∈ changes do
 8:         SegNotify(C_i, ⟨STATE, S_q, appState_q⟩)
 9:     end for
        /* Reconfigura algoritmo de RME */
10:     TOReconfigure(S_q.members)
11: end procedure
```

3.2.3.2 Divisão de Segmento

A divisão de segmentos (Algoritmo 9) ocorre quando o número de nós em um segmento excede uma constante n_{MAX}. Essa constante é conhecida globalmente e indica um número de nós a partir do qual é aconselhável formar duas MEs. A divisão em si consiste, inicialmente, em dividir tanto o conjunto de membros quanto o intervalo de chaves em dois segmentos, gerar certificados para esses segmentos, dividir e transferir o estado corretamente

aos novos membros. Depois, cada nó verifica em qual novo segmento se insere (caso não tenha invocado *SegLeave*) e altera o seu certificado de segmento e estado da aplicação em consequência. Por fim, a Máquina de Estados é reconfigurada parando a máquina atual e iniciando as máquina relativas aos novos segmentos.

O procedimento *Split* inicia com a divisão do conjunto total de membros em dois subconjuntos de maneira que um subconjunto $members^{<}$ conterá metade dos nós com identificadores inferiores (linha 2) e o outro subconjunto $members^{>}$ conterá a outra metade com identificadores superiores (linha 3). Cada segmento será responsável por um intervalo de chaves definido da seguinte forma: seja $[k,k')$ o intervalo de chaves do segmento antigo e p o membro que possui o menor identificador do subconjunto $members^{>}$ (linha 4), então $K^{<} = [k, C_p.id)$ e $K^{>} = [C_p.id, k')$ (linha 5). O $confId$ dos novos certificados será o sucessor do segmento atual (linha 6). A assinatura é similar ao caso da reconfiguração simples, porém são dois certificados assinados. Dessa forma, o procedimento *ExchSigs* é invocado passando dois certificados de segmento não assinados e retorna dois conjuntos de assinaturas (linha 8). A transferência do estado também se dá de maneira similar à reconfiguração simples, porém a um novo membro é enviado apenas o estado referente ao segmento do qual fará parte. Para cada segmento novo, o certificado é gerado (linhas 9 e 10), a parte relevante do estado da aplicação é obtida (linhas 11 e 12) e enviada aos novos membros do segmento em questão (linha 14 e 17). Após a transferência do estado, o nó verifica a qual dos novos segmentos este pertencerá (linha 19) e atualiza o estado da aplicação para descartar o estado relativo ao outro intervalo de chaves (linhas 20 e 22). A reconfiguração da RME ocorre de maneira similar à reconfiguração simples (linha 24).

3.2.3.3 União de Segmentos

A união de segmentos é um pouco mais complexa que a divisão, pois a reconfiguração envolve duas máquinas de estados em execução. A união é iniciada para evitar que o número de nós do segmento fique menor que os n_{MIN} necessários para manter as propriedades da RME. O segmento envia pedidos de união ao seu segmento sucessor no *overlay*. Depois ambos trocam informações para calcular os parâmetros do novo segmento, assinar o respectivo certificado e unir as partes do estado da aplicação. Por fim esse estado unido é enviado aos novos membros e a máquina de estados é reconfigurada.

A união é apresentada em duas listagens, os Algoritmos 10 e 11, uma vez que envolve dois procedimentos em segmentos distintos. Além disso, descrevemos a união em duas partes: o **início da união**, no qual os segmen-

Algoritmo 9 Divisão de segmentos (*Split*)

```
1:  procedure Split(newMembers)
    /* Código do servidor q */
        /* Divide membros dos novos certificados */
2:      members^< ← {metade dos C_i ∈ newMembers com C_i.id menor}
3:      members^> ← newMembers \ members^<
        /* Calcula intervalos de chaves */
4:      midK ← min{C_i.id : C_i ∈ members^>}
5:      K^< ← [S_q.start, midK); K^> ← [midK, S_q.end)
        /* Calcula novo identificador de configuração */
6:      newConfId ← S_q.confId + 1
        /* Inclui certificado antigo no histórico */
7:      newHistory ← S_q.history ∪ {S_q}
        /* Gera e dissemina assinaturas dos certificados (Algoritmo 7) */
8:      ⟨newΣ^<, newΣ^>⟩ ← ExchSigs(
            ⟨members^<, newConfId, K^<⟩, ⟨members^>, newConfId, K^>⟩)
        /* Gera novos certificados */
9:      S_q^< ← ⟨members^<, newConfId, K^<, newΣ^<, newHistory⟩
10:     S_q^> ← ⟨members^>, newConfId, K^>, newΣ^>, newHistory⟩
        /* Obtém estado da aplicação dividido */
11:     appState^< ← SegGetAppState(K(S^<))
12:     appState^> ← SegGetAppState(K(S^>))
        /* Envia certificado e estado aos novos membros */
13:     for all C_i ∈ S^< : ⟨+, C_i⟩ ∈ changes do
14:         SegNotify(C_i, ⟨STATE, S^<, appState^<⟩)
15:     end for
16:     for all C_i ∈ S^> : ⟨+, C_i⟩ ∈ changes do
17:         SegNotify(C_i, ⟨STATE, S^>, appState^>⟩)
18:     end for
        /* Verifica em qual segmento q está e altera estado */
19:     if C_q ∈ S^< then
20:         S_q ← S^<; SegSetAppState(K(S^<), appState^<)
21:     else
22:         S_q ← S^>; SegSetAppState(K(S^>), appState^>)
23:     end if
        /* Reconfigura algoritmo de RME */
24:     TOReconfigure(S_q.members)
25: end procedure
```

Algoritmo 10 União de segmentos (*Merge*) (parte 1 de 2)

```
    /* Código do servidor q */
 1: procedure Merge(newMembers)
        /* Calcula f + 1 membros com id menor */
 2:     lowIdNodes ← {C_i ∈ S_q : C_i.id entre os f + 1 menores}
        /* Envia pedido assinado para nós com id menor */
 3:     ∀C_i ∈ lowIdNodes : Send(C_i.addr, ⟨MERGE, C_q, S_q, newMembers⟩σ_q)
        /* Somente nós com id menor coletam pedidos e invocam união */
 4:     if C_q ∈ lowIdNodes then
            /* Recebe f + 1 pedidos de união válidos assinados */
 5:         mergeCount ← ∅
 6:         while #mergeCount < f + 1 do
 7:             wait for Receive(⟨MERGE, C_i, S_i, newMembers_i⟩σ_i)
                /* Verifica se pedido é valido */
 8:             if S_i = S_q ∧ C_i ∈ S_q ∧ newMembers_i = newMembers then
                    /* Acumula pedidos de união válidos */
 9:                 mergeCount ← mergeCount ∪ {⟨MERGE, C_i, S_i, newMembers_i⟩σ_i}
10:             end if
11:         end while
            /* Invoca união no segmento sucessor passando pedidos como prova */
12:         resp ←⊥
            /* Repete até receber resposta */
13:         while resp =⊥ do
                /* Busca segmento sucessor */
14:             SegFind(S_q.end, S_q.end)
15:             wait for SegFindOk(S_r)
                /* Envia pedido de união ordenado */
16:             resp ← SegRequest(S_r, ⟨MERGE, S_q, newMembers, mergeCount⟩)
17:         end while
18:     end if
        /* Aguarda segmento sucessor executar parte da união */
19:     wait for SegNotifyDeliver(⟨MERGE^part, S_r^part, appState_r⟩)
        /* Troca assinaturas com membros do segmento local */
20:     newΣ_q^part ← ExchSigs(⟨S_r^part.members, S_r^part.confId, K(S_r^part)⟩)
        /* Monta certificado completo juntando assinaturas */
21:     S_q ← ⟨S_r^part.members, S_r^part.confId, K(S_r^part), S_r^part.Σ ∪ newΣ_q^part, S_r^part.history⟩
        /* Obtem estado da aplicação */
22:     appState_q ← SegGetAppState(K(S_q))
        /* Envia certificado, estado da aplicação e pedidos de união */
23:     ∀C_i ∈ S_r : SegNotify(C_i, ⟨MERGE^part_OK, S_q, appState_q⟩)
        /* Altera estado da aplicação para união dos estados parciais */
24:     SegSetAppState(K(S_q), ⟨appState_q, appState_r⟩)
        /* Obtem estado da aplicação completo */
25:     appState ← SegGetAppState(K(S_q))
        /* Envia estado e certificado completos para novos membros */
26:     for all C_i ∈ S_q : ⟨+, C_i⟩ ∈ changes do
27:         SegNotify(C_i, ⟨STATE, S_q, appState⟩)
28:     end for
        /* Reconfigura parâmetros da ME */
29:     TOReconfigure(S_q.members)
30: end procedure
```

Algoritmo 11 União de segmentos (*Merge*) (parte 2 de 2)

```
    /* Código do servidor r */
 1: upon SegDeliver(C_q, ⟨MERGE, S_q, newMembers_q, mergeCount_q⟩) do
        /* Valida pedido de união (def. de ValidCount na linha 21) */
 2:     if S_q.end = S_r.start ∧ C_q ∈ S_q ∧ ValidCount(mergeCount_q, newMembers_q, S_q) then
            /* Calcula conjunto de membros da união */
 3:         newMembers ← newMembers_q ∪ (S_r.members ∪ {C_i : ⟨+, C_i⟩ ∈ changes})
                                \ {C_i : ⟨−, C_i⟩ ∈ changes}
            /* Calcula novos parâmetros do certificado */
 4:         newConfId ← max{S_q.confId, S_r.confId} + 1
 5:         newK ← K(S_q) ∪ K(S_r)
            /* Troca assinaturas com membros do segmento local */
 6:         newΣ_r^part ← ExchSigs(⟨newMembers, newConfId, newK⟩)
            /* Gera certificado parcial contendo só assinaturas locais */
 7:         S_r^part ← ⟨newMembers, newConfId, newK, newΣ_r^part,
                            S_q.history ∪ S_r.history ∪ {S_q, S_r}⟩
            /* Obtem estado parcial da aplicação */
 8:         appState_r ← SegGetAppState(K(S_q))
            /* Envia certificado e estado parciais para seg. antecessor */
 9:         ∀C_i ∈ S_q : SegNotify(C_i, ⟨MERGE^part, S_r^part, appState_r⟩)
            /* Aguarda seg. antecessor terminar união */
10:         wait for SegNotifyDeliver(⟨MERGE^part_OK, S_q, appState_q⟩)
            /* Guarda certificado completo */
11:         S_r ← S_q
            /* Altera estado da apl. para união dos estados parciais */
12:         SegSetAppState(K(S_r), ⟨appState_q, appState_r⟩)
            /* Obtem estado da aplicação completo */
13:         appState ← SegGetAppState(K(S_r))
            /* Envia estado e certificado para novos membros */
14:         for all C_i ∈ S_r : ⟨+, C_i⟩ ∈ changes do
15:             SegNotify(C_i, ⟨STATE, S_r, appState⟩)
16:         end for
            /* Reconfigura parâmetros da ME */
17:         TOReconfigure(S_r.members)
            /* Limpa pedidos de reconfiguração e changes */
18:         reconfCount ← ∅; changes ← ∅
19:     end if
20: end upon

    /* Definição do predicado ValidCount */
21: ValidCount(mergeCount_q, newMembers_q, S_q) ≜
                #mergeCount_q ≥ f + 1 ∧
                ∀⟨MERGE, C_i, S_i, newMembers_i⟩σ_i ∈ mergeCount_q :
                            S_i = S_q ∧ C_i ∈ S_i ∧ newMembers_i = newMembers_q
```

tos se comunicam a fim de garantir a execução do procedimento; e a **união propriamente dita**, na qual os segmentos trocam informações para efetivar a união.

No **início da união**, os nós do **segmento iniciador** escolhem $f+1$ nós para serem os acumuladores dos pedidos de união. Esses acumuladores são responsáveis por invocar o **segmento sucessor** passando os pedidos de união assinados como prova. O uso de $f+1$ nós garante que pelo menos um é correto e vai terminar por realizar a invocação do segmento sucessor. Os nós escolhidos para serem acumuladores são os $f+1$ com identificador de nó menor (Alg. 10, linha 2) e cada nó envia a estes um pedido de união assinado (Alg. 10, linha 2). O código das linhas 4 a 18 do Algoritmo 10 somente é executado pelos nós acumuladores. Esse código consiste em coletar (Alg. 10, linha 7), validar (Alg. 10, linha 8) e acumular os pedidos assinados no conjunto *mergeCount*. Depois de obter os $f+1$ pedidos de união, os nós acumuladores realizam a invocação ordenada do segmento sucessor. O segmento sucessor é buscado usando o *overlay* (Alg. 10, linha 14) e invocado passando o conjunto de membros calculado pelo Algoritmo $SegReconfigure$ (Alg. 6, linha 15) no segmento iniciador e o conjunto *mergeCount* (Alg. 10, linha 16). A invocação é repetida até que uma resposta seja recebida, para lidar com reconfigurações que possam ocorrer no segmento sucessor de maneira concorrente. Do lado do segmento sucessor, esses pedidos são validados (Alg. 11, linha 2) para dar início à reconfiguração a fim de poder realizar a união.

Quando o segmento sucessor recebe os pedidos de união necessários, este inicia a **união propriamente dita**. Como recebe o conjunto de nós do segmento iniciador, o segmento sucessor pode calcular o seu conjunto de membros e determinar quais serão os nós participantes do segmento resultante da união (Alg. 11, linha 3). Além disso, o segmento sucessor calcula também o identificador e o intervalo de chaves do novo segmento (Alg. 11, linhas 4 e 5). Essas informações, o conjunto de membros, o identificador e o intervalo de chaves, são os dados assinados do certificado de segmento, logo os nós do conjunto sucessor podem trocar assinaturas (Alg. 11, linha 6) e formar um certificado parcial (Alg. 11, linha 7), isto é, que ainda precisa da assinatura dos nós do segmento iniciador da união. Esse certificado parcial é enviado aos nós do segmento iniciador juntamente com o estado da aplicação relativo ao segmento sucessor (Alg. 11, linha 9). De sua parte, os nós do segmento sucessor aguardam o segmento iniciador finalizar a geração do certificado e enviar sua parte do estado (Alg. 11, linha 10).

Do lado do segmento iniciador, a **união propriamente dita** inicia com os nós aguardando o recebimento do certificado parcial e do estado do segmento sucessor (Alg. 10, linha 19). De posse do certificado parcial, os nós trocam assinaturas de maneira similar ao segmento sucessor (Alg. 10, linha

20). Com todas as assinaturas, o segmento iniciador pode montar o certificado completo do segmento resultante da união (Alg. 10, linha 21). Na sequência, esse certificado é enviado junto com parte do estado da aplicação novamente ao segmento sucessor (Alg. 10, linha 23).

Nesse momento, os nós dos dois segmentos possuem o certificado e o estado da aplicação completos, que logo na sequência repassam à aplicação (Alg. 10, linha 24 e Alg. 11, linha 12) e enviam aos novos membros (Alg. 10, linha 27 e Alg. 11, linha 15). Depois disso, ambos segmentos alteram os parâmetros do algoritmo de RME para terminar a máquina de estados do segmento antigo e iniciar a máquina do segmento resultante da união (Alg. 10, linha 29 e Alg. 11, linha 17). No segmento sucessor, finalmente os conjuntos *changes* e $reconfCount$ são esvaziados uma vez que uma reconfiguração acabou de ser executada (Alg. 11, linha 18). Isso também é executado no segmento iniciador, porém pelo Algoritmo $SegReconfigure$ (linhas 20 e 21) após o término do procedimento

3.2.4 Provas de Funcionamento dos Algoritmos

Nesta seção tratamos das provas do funcionamento correto das operações da camada de segmentação. Para tal, demonstramos que os algoritmos das Subseções 3.2.1, 3.2.2 e 3.2.3, apresentadas anteriormente, satisfazem as propriedades apresentadas na Seção 3.2. As provas estão associadas a teoremas que correspondem às propriedades do sistema, bem como lemas relacionados a subpropriedades utilizadas nos teoremas. O uso de lemas divide as provas em unidades gerenciáveis e simplifica a leitura e compreensão das mesmas. As provas assumem que as operações das camadas inferiores, utilizadas nos algoritmos da camada de segmentação, satisfazem as características descritas na Seção 3.1.

Antes de provarmos as propriedades das operações públicas da camada de segmentação, enunciamos e demonstramos uma série de lemas relacionados a propriedades e invariantes das reconfigurações do sistema. Esses lemas são essenciais para demonstrar os teoremas que serão apresentados na sequência e estão diretamente ligados ao funcionamento interno da segmentação.

O Lema 3.1 (Mesmo *changes* SegReconfigure), a seguir, refere-se às alterações no conjunto *changes*. É importante para o funcionamento da reconfiguração que todos os nós corretos calculem o mesmo conjunto de entradas e saídas e, portanto, cheguem às mesmas conclusões quanto ao tipo de reconfiguração e ao conjunto de membros dos novos segmentos.

Lema 3.1 (Mesmo *changes* SegReconfigure)**.** *Em toda reconfiguração, todo*

nó correto de um mesmo segmento possui o mesmo valor no conjunto changes.

Demonstração. O conjunto *changes* é alterado nos seguintes pontos: (a) itens são incluídos como resposta às chamadas *SegJoin* (Algoritmo 4, linha 15) e *SegLeave* (Algoritmo 5, linha 10); (b) o conjunto *changes* é esvaziado na execução da reconfiguração (Algoritmo 6, linha 20 e Algoritmo 11, linha 18). No caso (a), nós de um mesmo segmento executam essas alterações em resposta a requisições ordenadas na Máquina de Estados, logo pelas propriedades de ordenação total e acordo, todos os nós executam as mesmas alterações. No caso (b), fica claro que no início da atividade de todo segmento, todo nó correto inicia com *changes* vazio. Dessa forma, em todo nó correto de todo segmento, *changes* começa igual após a reconfiguração e sofre as mesmas inclusões de pedidos de entrada e saída. Como a reconfiguração é realizada por operações ordenadas na ME (Algoritmo 6, linha 13, Algoritmo 11, linha 1), então o conjunto *changes* necessariamente tem o mesmo valor em todos os nós do segmento. □

O Lema 3.2 (Início SegReconfigure), a seguir, estabelece a característica de que sempre que há um pedido de entrada ou saída, uma reconfiguração termina por ocorrer. Isso permite demonstrar a terminação das operações *SegJoin* e *SegLeave*.

Lema 3.2 (Início SegReconfigure)**.** *Em qualquer segmento S, se algum nó correto possui changes* $\neq \emptyset$*, então uma reconfiguração termina por ocorrer.*

Demonstração. O temporizador $Timeout(RECONF)$ é periódico, ou seja, estoura em intervalos de tempo definidos em todo nó correto. Como *changes* $\neq \emptyset$, o nó passa no teste do Algoritmo 6 (linha 2) (e pelo Lema 3.1 (Mesmo *changes* SegReconfigure), todos os nós corretos também passam). Portanto, se um nó correto realiza uma tentativa de reconfiguração, todos os outros nós também terminam por fazer o mesmo. Como há pelo menos $2f + 1$ nós corretos, o número de tentativas é suficiente para iniciar a reconfiguração. □

Lema 3.3 (Acordo SegReconfigure)**.** *Dada uma requisição de reconfiguração, todo nó correto do segmento concorda quanto ao tipo de reconfiguração (simples, divisão ou união).*

Demonstração. O tipo de reconfiguração, conforme especificado no Algoritmo 6 (linhas 16 a 19), depende apenas do número de nós participantes resultante da reconfiguração, ou seja, o número de membros atuais mais as alterações descritas no conjunto *changes*. Como ambos são iguais em todo nó correto (Lema 3.1 (Mesmo *changes* SegReconfigure)), todo nó correto faz a mesma decisão. □

Os Lemas 3.4 (Manutenção da Cobertura SegReconfigure) e 3.5 (Cobertura Total SegReconfigure), a seguir, correspondem a um invariante importante do sistema, de que, para toda chave *k* do *overlay*, sempre há exatamente um segmento responsável por essa chave. Esse Lema é importante para provar as propriedades da busca de segmentos.

Lema 3.4 (Manutenção da Cobertura SegReconfigure)**.** *Sejam S_i, $i \in [1,n]$, os segmentos envolvidos em uma reconfiguração que gera os segmentos S'_j, $j \in [1,m]$: (i) se os intervalos de chaves cobertos pelos S_i são adjacentes (ou seja, sem "buracos" e não sobrepostos), então os intervalos de chaves cobertos pelos S'_j também são adjacentes; e (ii) a união dos intervalos de chaves cobertos pelos S_i é igual ao intervalo coberto pelos S'_j, ou seja:*

$$\bigcup_{i=1}^{n} K(S_i) = \bigcup_{j=1}^{m} K(S'_j)$$

Demonstração. Pelo Lema 3.3 (Acordo SegReconfigure), todos os nós corretos dos segmentos S_i concordam quanto ao tipo de reconfiguração. Portanto, há três casos a se considerar:

(a) na reconfiguração simples (Algoritmo 8), $n = m = 1$, ou seja, existe apenas um segmento S_1 e um segmento S'_1 tais que $S_1 \prec S'_1$. Por construção (linha 5), temos que $K(S_1) = K(S'_1)$, ou seja, a cobertura se mantém idêntica;

(b) na divisão (Algoritmo 9), existe um segmento inicial S_1 e mais de um segmento resultante da reconfiguração, sendo $S'_1 = S^<, S'_2 = S^>$. Por construção (linhas 4 e 5), temos que a divisão do espaço de chaves entre os segmentos é feita de maneira que $K(S_1) = K(S^<) \cup K(S^>)$ e $K(S^<) \cap K(S^>) = \emptyset$, ou seja, os intervalos resultantes são adjacentes e cobrem todo o intervalo do segmento que se dividiu;

(c) na união (Algoritmos 10 e 11), há apenas um segmento resultante S'_1 e mais de um segmento inicial sendo $S_1 = S_q$ o segmento iniciador (que executa o Alg. 10) e $S_2 = S_r$ o segmento sucessor (que executa o Alg. 11, linhas 2 a 19). Por construção (Alg. 10, linha 14), temos que o segmento sucessor possui intervalo adjacente ao segmento iniciador. Além disso, também por construção (Alg. 11, linha 5), temos que $K(S'_1) = K(S_q) \cup K(S_r)$, portanto a cobertura se mantém idêntica.

Nos três casos é possível ver que os segmentos resultantes cobrem as mesmas chaves dos segmentos antigos e nenhuma sobreposição é criada. □

Lema 3.5 (Cobertura Total SegReconfigure)**.** *Em qualquer instante t e para toda chave k do* overlay*, existe exatamente um segmento S tal que: $Active(t,S) \wedge k \in K(S)$*

Demonstração. Pela Definição 3.5 (Atividade de Segmento), temos que um segmento somente deixa de ser ativo quando este passa por uma reconfiguração. Além disso, os intervalos de chaves cobertos pelos segmentos também somente são alterados nas reconfigurações. Assim sendo, demonstramos o lema por meio de uma indução sobre as reconfigurações do sistema, ou seja, sobre o grafo induzido pela relação $\prec$.

O caso base é $K(S_0) = [0, k_{MAX}]$, no qual existe exatamente um segmento ativo que cobre todas as chaves do *overlay*. Esse caso certamente satisfaz o invariante do lema. O passo de indução é dado pelo Lema 3.4 (Manutenção da Cobertura SegReconfigure). □

O Lema 3.6 (ExchSigs) prova que o procedimento de troca de assinaturas, utilizado para gerar novos certificados de segmento, termina sempre que executado por todos os nós corretos usando os mesmos parâmetros e sempre retorna $f+1$ assinaturas de nós do segmento sobre os certificados passados. Esse lema é importante para demonstrar que as reconfigurações, e por consequência as entradas e saídas de nós, terminam e geram certificados válidos.

Lema 3.6 (ExchSigs)**.** *Se todos os nós corretos de um segmento S executam ExchSigs com os mesmos parâmetros, então o procedimento retorna $f+1$ assinaturas geradas por nós de S para cada parâmetro passado.*

Demonstração. O único ponto de espera no Algoritmo 7 é o recebimento de $f+1$ assinaturas (linhas 9 a 16). Esse laço de repetição somente termina quando $f+1$ assinaturas são recebidas para todos os certificados de segmento que estão sendo assinados. Como supomos que todo nó correto executa o procedimento com os mesmos parâmetros, então todo nó correto gera assinaturas sobre os mesmos certificados (linha 3). Da mesma forma, todo nó correto dissemina essas assinaturas para todos os outros nós do mesmo segmento (linha 6). Sendo assim, como os canais de comunicação são confiáveis e temos pelo menos $2f+1$ nós corretos no segmento, fica garantido que todo nó correto recebe pelo menos $2f+1$ conjuntos de assinaturas válidas sobre os certificados de segmento assinados, o suficiente para sair do laço de repetição e terminar o procedimento. Por construção (linha 17) as $f+1$ assinaturas recebidas para cada certificado passado como parâmetro são retornadas. □

Lema 3.7 (Validade de Certificados SegReconfigure)**.** *Sejam S_i, $i \in [1,n]$, os segmentos envolvidos em uma reconfiguração que gera os segmentos S'_j, $j \in$*

$[1,m]$*: se os certificados dos segmentos* S_i *são válidos, então os certificados dos segmentos* S'_j *também são válidos.*

Demonstração. A Definição 3.4 (Validade de Segmento) exige que os certificados de segmento gerados nas reconfigurações sejam assinados por $f+1$ nós dos segmentos antigos (pelo menos uma assinatura de um nó correto). Pelo Lema 3.3 (Acordo SegReconfigure), todos os nós corretos dos segmentos S_i concordam quanto ao tipo de reconfiguração executado. Portanto, há três casos a se considerar:

(a) na reconfiguração simples (Algoritmo 8), os nós geram o novo certificado de segmento incluindo as assinaturas retornadas pelo procedimento *ExchSigs* (linha 3). Certamente todo nó correto do segmento invoca *ExchSigs* com os mesmos parâmetros, gerados diretamente a partir do certificado de segmento antigo e do conjunto *changes* que é igual pelo Lema 3.1 (Mesmo *changes* SegReconfigure). Assim sendo, pelo Lema 3.6 (ExchSigs), temos que essas assinaturas são de fato dos nós do segmento antigo e geradas a partir das informações do certificado novo;

(b) na divisão de segmento (Algoritmo 9) a execução é similar ao caso da reconfiguração simples, porém mais de um certificado de segmento é gerado. O argumento, no entanto, é o mesmo, isto é, as assinaturas geradas com *ExchSigs* a partir dos mesmos parâmetros (linha 8) são usadas na geração dos certificados (linhas 9 e 10);

(c) na união de segmentos (Algoritmos 10 no segmento iniciador e 11 no segmento sucessor) a situação também é similar aos casos anteriores, em que as assinaturas são geradas por *ExchSigs*, porém há dois procedimentos distintos e dois conjuntos de assinaturas são gerados. No Algoritmo 11 (linha 6), os parâmetros para *ExchSigs* são construídos levando em conta valores recebidos nos pedidos de união (linha 1). Como os pedidos são ordenados, todos os nós corretos recebem os mesmos valores. No mais, os parâmetros levam em conta o conjunto *changes* e o certificado do segmento, que são iguais nos nós corretos do segmento. Dessa forma, no Algoritmo 11 *ExchSigs* é invocado com os mesmos parâmetros e gera as assinaturas corretamente. No Algoritmo 10, os parâmetros de *ExchSigs* são recebidos do outro segmento por meio de $SegNotifyDeliver$ (linha 19. Pelas características da operação de notificação, temos que a entrega somente é realizada se for recebida de $f+1$ nós do mesmo segmento, ou seja, pelo menos um nó correto. No Algoritmo 11 todos os nós corretos enviam a notificação a todos os nós do segmento inciador (linha 9), o que garante o recebimento dos

mesmos parâmetros em todos os nós. Assim sendo, o conjunto de assinaturas geradas no Algoritmo 10 também contém $f + 1$ assinaturas sobre o certificado correto.

Por esses três casos, temos que em todas as reconfigurações são incluídas $f + 1$ assinaturas de nós dos segmentos antigos nos certificados de segmento gerado. Como o lema supõe que os segmentos antigos são válidos, então pela Definição 3.4 (Validade de Segmento) os segmentos resultantes também são válidos. □

O Lema 3.8 (Consistência SegReconfigure), a seguir, garante que todos os nós corretos do mesmo segmento executam as operações sobre o mesmo estado da aplicação, o que garante que as respostas são iguais. Esse fato é importante para garantir que uma requisição termina corretamente.

Lema 3.8 (Consistência SegReconfigure)**.** *Em qualquer segmento ativo S: dada uma requisição para a Máquina de Estados, todo nó correto executa essa requisição sobre o mesmo valor do estado da aplicação e o mesmo certificado de segmento S.*

Demonstração. Provamos esse teorema por meio de uma indução sobre as requisições do sistema, da seguinte forma: o caso base é o do segmento inicial, no qual todos os nós possuem o mesmo certificado de segmento e estado inicial da aplicação (ambos cobrem todas as chaves do *overlay*; o passo de indução é dado pela hipótese de que dado que todos os nós corretos de um segmento partem de um mesmo estado e certificado, então a execução de uma requisição ordenada qualquer leva os nós corretos a um mesmo estado e certificado, ou seja, o invariante não é quebrado. Claramente as operações da camada de aplicação não violam o passo de indução, uma vez que estas não alteram o certificado de segmento e os algoritmos de RME garantem a execução ordenada em todos os nós corretos (ou seja, a alteração do estado é consistente em todos os nós corretos). Resta então mostrar que as operações de reconfiguração sempre geram novos segmentos nos quais todos os nós corretos possuem o mesmo certificado e estado da aplicação.

Em todas as reconfigurações, nós corretos somente são adicionados por meio da operação *SegJoin* (Algoritmo 4). Dessa forma, o certificado de segmento e estado somente são aceitos pelos nós corretos adicionados se $f + 1$ cópias idênticas são recebidas (conforme operação $SegNotifyDeliver$, Algoritmo 3, teste da linha 6). Dessa forma, para que os nós adicionados aos segmentos possuam o mesmo certificado e estado, precisamos demonstrar que, seja qual for o tipo de reconfiguração, os segmentos adicionados recebem o certificado e estado corretos de $f + 1$ nós do segmento antigo.

Sabemos que, por conta dos algoritmos de RME, todos os nós corretos do segmento concordam quanto à ordem de execução da requisição de reconfiguração. Além disso, pelo Lema 3.3 (Acordo SegReconfigure) sabemos que todos os nós corretos também concordam quanto ao tipo de reconfiguração executado. Vamos, a seguir, verificar cada caso e mostrar que o certificado de segmento e estado são transferidos de maneira a garantir o invariante do lema:

- em uma reconfiguração simples (Algoritmo 8), o estado não é dividido ou unido, nem os limites do intervalo de chaves coberto são alterados, logo basta que o certificado e estado sejam transferidos aos nós adicionados ao novo segmento. A transferência ocorre na linha 8, que ocorre incondicionalmente no algoritmo. Assim sendo, todo nó correto do segmento antigo envia o seu certificado e estado a todo nó adicionado ao segmento novo. Supondo a validade do invariante para o segmento antigo, então o certificado e estado enviados pelos nós são idênticos, logo os nós adicionados recebem mais de $f + 1$ cópias do mesmo estado e são notificados com *SegNotifyDeliver*. Assim, todos os nós do novo segmento partirão do mesmo estado e com o mesmo certificado de segmento;

- em uma divisão de segmento (Algoritmo 9), o estado é dividido conforme os segmentos novos gerados. Tanto os nós do segmento antigo quanto os nós adicionados na reconfiguração podem fazer parte de qualquer dos segmentos novos gerados, conforme a pertinência de seus identificadores. No Algoritmo 9, o estado é dividido conforme os intervalos de chaves dos novos segmentos (linhas 11 e 12), e esses estados são enviados, juntamente com os respectivos certificados de segmento, a todos os nós adicionados (linhas 14 e 17). Isso garante o recebimento do certificado e estado por todos os nós corretos adicionados. Na sequência, os nós corretos do segmento antigo, conforme a pertinência de seus identificadores (teste da linha 19), alteram seus certificados de segmento e estado para corresponder aos enviados anteriormente para os nós adicionados (linhas 20 e 22). Dessa forma fica garantido que tanto nós do segmento antigo quanto nós adicionados aos novos segmentos possuem o mesmo certificado e estado;

- em uma união de segmentos (Algoritmos 10 e 11), os estados dos segmentos antigos são unidos para o segmento novo. Tanto os nós dos segmentos antigos quanto os nós adicionados por todos os segmentos unidos fazem parte do mesmo segmento e recebem o mesmo certificado e estado. Primeiro mostramos que os nós dos segmentos antigos

passam a possuir o mesmo certificado e estado após a união e depois mostramos que os nós adicionados recebem esses mesmos certificado e estado. Os estados são trocados entre os nós dos segmentos antigos por meio de operações $SegNotify$ (Algoritmo 11, linhas 8 e 9, Algoritmo 10, linhas 22 e 23). Em todos os segmentos antigos, os estados parciais são concatenados da mesma forma e repassados à aplicação (Algoritmo 10, linhas 24, Algoritmo 11, linha 12), ou seja, os nós dos segmentos antigos passam a ter o mesmo estado. O certificado do segmento resultante da união é gerado por um segmento (Algoritmo 10, linha 21) e enviado aos demais segmentos juntamente com o estado parcial (Algoritmo 10, linha 23). Nos demais segmentos antigos, o certificado do novo segmento é recebido e armazenado (Algoritmo 11, linhas 10 e 11). Assim sendo, os nós dos segmentos antigos passam a ter o mesmo estado e mesmo certificado de segmento. Quanto aos nós adicionados, cada segmento antigo age independentemente, pois os nós adicionados aguardam resposta dos segmentos específicos aos quais requisitaram a entrada. Os segmentos antigos enviam o estado e certificado de segmento que foi trocado durante a união, que mostramos ser o mesmo para todos (Algoritmo 10, linha 27 e Algoritmo 11, linha 15). Assim sendo, todos os nós corretos, tanto dos segmentos antigos quanto adicionados, iniciam no novo segmento com o mesmo estado e certificado de segmento.

Conforme mostrado nos casos acima, seja qual for o tipo de reconfiguração, o invariante do lema é mantido. □

O Lema 3.9 (Terminação SegReconfigure) garante que toda reconfiguração termina e é essencial para demonstrar que nós sempre conseguem entrar e sair do sistema, já que esses eventos ocorrem no término de reconfigurações.

Lema 3.9 (Terminação SegReconfigure)**.** *Toda reconfiguração iniciada termina em todo nó correto.*

Demonstração. A reconfiguração de um segmento pode ser iniciada por dois eventos distintos: (i) pelo estouro do temporizador *RECONF* nos nós do próprio segmento ou (ii) por um pedido de união do segmento antecessor. No caso (i), a reconfiguração propriamente dita inicia quando $f+1$ nós enviam a mensagem com rótulo *TRY_RECONF*, o que leva à ordenação da mensagem *RECONF* na ME. Como o temporizador acaba por estourar em todo nó correto, pelas propriedades da RME, sabemos que todo nó correto acaba por executa o código associado à reconfiguração (Algoritmo 6, linhas 13 a 23). No Algoritmo 6, dentro da reconfiguração não há nenhuma instrução bloqueante, logo se a operação não terminar deve ser por conta de alguma instrução

nos sub procedimentos (a) *SimpleReconfig*, (b) *Split* ou (c) *Merge*. A partir do Lema 3.3 (Acordo SegReconfigure), sabemos que todo nó correto concorda quanto ao sub procedimento de reconfiguração que deve ser executado. Analisamos, a seguir, cada uma das possibilidades:

(a) na reconfiguração simples (Algoritmo 8), o único ponto de espera no qual seria possível bloquear a terminação da reconfiguração simples é a troca de assinaturas (linha 3). Todo nó correto calcula o mesmo *newMembers*, bem como o mesmo identificador (linha 2) e intervalo de chaves, uma vez que derivam esses parâmetros do mesmo certificado de segmento antigo. Desse modo, temos que a troca de assinaturas será invocada em todo nó correto com os mesmos parâmetros e pelo Lema 3.6 (ExchSigs) fica garantido o término da reconfiguração;

(b) a divisão de segmentos (Algoritmo 9) apresenta característica similar à reconfiguração simples, ou seja, somente a troca de assinaturas (linha 8 seria capaz de bloquear indefinidamente a reconfiguração, porém pelo mesmo motivo do caso (a) os parâmetros calculados dos certificados de segmento gerados são os mesmos e a troca de assinaturas termina;

(c) a diferença da união (Algoritmo 10) com relação aos casos anteriores está no fato de a união envolver a comunicação com a máquina de estados do segmento sucessor. Dessa forma, o término da união depende da evolução de duas MEs comunicantes. Este caso (c) será tratado no parágrafo abaixo juntamente com o caso (ii) no qual a reconfiguração é iniciada por um pedido de união.

O segmento que inicia o procedimento da união (caso (i.c)) é chamado de **segmento iniciador**. O segmento ao qual o segmento iniciador se une (caso (ii)) é denominado **segmento sucessor**. A seguir, dividimos a prova de que uma reconfiguração de união termina em duas etapas: (1) primeiro provamos que algum segmento sucessor de fato aceita o pedido do segmento iniciador; (2) depois provamos que os segmentos se comunicam de maneira a garantir que ambos terminam seus procedimentos:

(1) precisamos provar que algum segmento ativo, sucessor do segmento iniciador, acaba por realizar a união, o que ocorre, segundo o Algoritmo 11 (linha 2), quando os nós do segmento recebem uma requisição contendo $f+1$ pedidos de união válidos. Os nós do segmento iniciador começam o procedimento *Merge* (Algoritmo 10) enviando os pedidos de união aos $f+1$ nós com identificador menor no segmento, denominados de **nós acumuladores** (linhas 2 e 3). Esses pedidos são

validados e acumulados pelos nós acumuladores. Ao receber $f+1$ pedidos válidos, os nós acumuladores buscam e invocam o segmento sucessor (linhas 13 a 17). Nessa invocação, os nós acumuladores passam na requisição o conjunto de novos membros calculados para a reconfiguração. Nos passos anteriores desta prova mostramos que todos os nós corretos calculam o mesmo conjunto, logo fica garantido que os pedidos de união válidos serão todos idênticos. Como são escolhidos $f+1$ nós acumuladores, fica assegurado que pelo menos um é correto e realiza a invocação ao segmento sucessor. O laço de repetição executado pelos nós acumuladores, juntamente com o Lema 3.5 (Cobertura Total SegReconfigure) e os Teoremas 3.17 (Terminação SegRequest) e 3.18 (Segmento Inativo SegRequest), garante que em algum momento o segmento sucessor recebe uma requisição contendo os pedidos de união assinados. No segmento sucessor (Algoritmo 11), o conjunto de pedidos *mergeCount* é verificado para garantir que foi gerado por nós de um segmento válido (linha 2). Como o conjunto *mergeCount* possui $f+1$ pedidos válidos, a validação passa e o segmento sucessor entra na união propriamente dita;

(2) após enviar o pedido de união a algum segmento sucessor, todos os nós corretos do segmento iniciador passam a aguardar uma notificação do segmento sucessor contendo o estado e certificado parciais (Algoritmo 10, linha 19). Essa notificação é enviada pelos nós do segmento sucessor após a geração dos parâmetros do certificado, troca de assinaturas e obtenção do estado (Algoritmo 11, linha 9). Pelos mesmos argumentos dos passos anteriores desta prova, a terminação da troca de assinaturas é garantida. Assim sendo, o segmento iniciador acaba por receber a notificação com o certificado parcial. Na sequência, a espera se inverte e o segmento sucessor passa a aguardar uma notificação do segmento iniciador contendo o estado parcial da aplicação e o certificado completo do segmento resultante da união (Algoritmo 11, linha 10). O segmento iniciador, logo após receber o certificado parcial, realiza a troca de assinaturas (que também termina pelos mesmos motivos já expostos acima), monta o certificado de segmento completo, obtém o estado da aplicação e envia a notificação com o certificado gerado (Algoritmo 10, linha 23). Dessa forma, o segmento sucessor acaba por receber a notificação com o certificado completo. Depois disso, não há mais nenhum ponto de bloqueio e a terminação da união está garantida.

□

O Lema 3.10 (Reconfigurações Finitas SegReconfigure), a seguir, está relacionado à premissa de que o número de entradas e saídas do sistema é

finito. Provamos que, dada essa premissa, o número de reconfigurações do sistema também é finito. Esse Lema será usado para demonstrar propriedades de terminação da busca e invocação de segmentos.

Lema 3.10 (Reconfigurações Finitas SegReconfigure)**.** *O número de reconfigurações do sistema é finito.*

Demonstração. Assumimos que o número de entradas e saídas do sistema é finito. Portanto, o número de chamadas a *SegJoin* e *SegLeave* é finito. Consequentemente, após a reconfiguração executada após a última entrada ou saída, o Algoritmo 6 esvazia o conjunto changes (linha 20) e este permanece vazio em todo nó correto do sistema, pois somente uma chamada *SegJoin* (Algoritmo 4, linha 15) ou *SegLeave* (Algoritmo 5, linha 10) pode provocar uma inclusão nesse conjunto. Nenhum nó correto inicia uma reconfiguração se o conjunto *changes* for vazio no momento em que o temporizador de reconfigurações estoura, conforme o teste da linha 2. Por fim, nenhum nó correto pode receber mais de f tentativas de reconfiguração, portanto nenhum nó correto invoca a máquina de estados para enviar um pedido de reconfiguração (linhas 8 a 10). Mesmo que um nó malicioso execute a linha 9, a chamada não pode ter as $f+1$ tentativas assinadas e não passa no teste da linha 14. Outro ponto no qual uma reconfiguração pode ser iniciada é uma invocação a um segmento pedindo uma união de segmentos, no entanto $f+1$ pedidos são necessários para provocar a união (Alg. 11, linha 2), ou seja, o segmento predecessor deve ter executado *SegReconfigure*, o que é impossível segundo o argumento anterior. □

O Lema 3.11 (Monotonicidade $confId$ SegReconfigure), a seguir, garante que é possível utilizar o atributo $confId$ dos certificados de segmento para verificar a antiguidade do mesmo. Esse Lema é citado como parte da demonstração do teorema relativo à tentativa de invocação de um segmento inativo, apresentado mais abaixo.

Lema 3.11 (Monotonicidade $confId$ SegReconfigure)**.** *Para quaisquer segmentos S e S' válidos, se $S \prec S'$, então $S'.confId > S.confId$.*

Demonstração. A prova se dá por construção, pela forma como o valor de $confId$ é gerado para os novos certificados de segmento. Na reconfiguração simples (Algoritmo 8), o $confId$ é simplesmente incrementado (linha 2). O mesmo ocorre na divisão de segmentos (Algoritmo 9), porém são dois segmentos gerados (linha 6). Na união de segmentos (Algoritmo 10), o novo $confId$ é o sucessor do maior $confId$ dentre os segmentos unidos (linha 4). □

O Lema 3.12 (Acordo $TOReconfigure$ SegReconfigure), a seguir, garante que nós corretos sempre implantam a mesma visão ao final de reconfigurações, isto é, nós corretos concordam quanto à composição da máquina de estados dos segmentos resultantes de uma reconfiguração.

Lema 3.12 (Acordo $TOReconfigure$ SegReconfigure)**.** *Se um nó correto p executa $TOReconfigure(P)$ para um conjunto P de certificados de nó, então $C_p \in P$ e todo nó correto i, tal que $C_i \in P$, também termina por executar $TOReconfigure(P)$, para o mesmo conjunto P.*

Demonstração. A primitiva $TOReconfigure$ somente é chamada pelos nós quando ocorre uma reconfiguração ou quando entram em algum segmento (pela operação *SegJoin*). Precisamos demonstrar que em qualquer reconfiguração, todos os nós corretos que participam dos segmentos gerados executam $TOReconfigure$ com o mesmo conjunto. Por construção, seja qual for o tipo de reconfiguração, em qualquer nó correto, a primitiva $TOReconfigure$ é invocada passando o conjunto de membros indicado no certificado de segmento recém gerado ou recebido (Algoritmo 4, linhas 9 e 12; Algoritmo 8, linhas 5 e 10; Algoritmo 9, linhas 20, 22 e 24; Algoritmo 10, linhas 21 e 29; e Algoritmo 11, linhas 11 e 17). Pela prova do Lema 3.8 (Consistência SegReconfigure), demonstramos que ao final da reconfiguração todos os nós corretos do mesmo segmento gerado (inclusive nós recém adicionados) possuem o mesmo certificado, o que implica que a chamada $TOReconfigure$ é realizada com os mesmos conjuntos de membros. □

O Teorema 3.13 (Consistência do Estado), a seguir, corresponde à Propriedade 3.10 (Consistência do Estado) do sistema e está relacionado ao funcionamento correto da operação *SegReconfigure*, em especial a transferência de estado dos segmentos antigos aos segmentos novos.

Teorema 3.13 (Consistência do Estado)**.** *Sejam S_i, $i \in [1,n]$, os segmentos envolvidos em uma reconfiguração que gera os segmentos S'_j, $j \in [1,m]$: se $appState$ é a união dos estados armazenados nos segmentos S_i imediatamente antes da reconfiguração iniciar e $appState'$ é a união dos estados armazenados em S'_j imediatamente após o término da reconfiguração, então $appState = appState'$.*

Demonstração. Precisamos demonstrar que em toda reconfiguração o estado da aplicação nos segmentos novos é igual ao estado nos segmentos antigos. Pelo Lema 3.3 (Acordo SegReconfigure), todos os nós corretos dos segmentos S_i concordam quanto ao tipo de reconfiguração. Portanto, há três casos a se considerar:

(a) na reconfiguração simples (Algoritmo 8), não ocorre alteração dos limites do intervalo de chaves do segmento. Por construção, o algoritmo

não altera o estado nos nós do segmento antigo e envia o mesmo estado aos nós adicionados (linhas 6 e 8). Dessa forma, o estado no segmento antigo antigo é idêntico ao estado no segmento novo;

(b) na divisão de segmento (Algoritmo 9), o estado é dividido para cada segmento gerado a partir da divisão (linhas 11 e 11). Por construção, as partes do estado obtidas são enviadas aos nós adicionados (linhas 14 e 17) e armazenados pelos nós do segmento antigo (linhas 20 e 22). Pelo Lema 3.4 (Manutenção da Cobertura SegReconfigure), os intervalos de chaves dos segmentos gerados são adjacentes e cobrem as mesmas chaves do segmento antigo. Como os estados são divididos com base nos certificados de segmento gerados, a união dos estados dos segmentos resultantes da divisão é igual ao estado do segmento antigo;

(c) na união de segmentos (Algoritmos 10 e 11), os estados da aplicação nos segmentos antigos é unido no segmento resultante. Por construção, o estado completo de cada segmento antigo é obtido (Alg. 10, linha 22, Alg. 11, linha 8) e transferido para o segmento adjacente (Alg. 10, linha 9, Alg. 11, linha 23). Em cada segmento antigo, as partes do estado são unidas (Alg. 10, linha 24, Alg. 11, linha 12) e a união resultante é enviada aos nós adicionados na reconfiguração (Alg. 10, linhas 25 e 27, Alg. 11, linhas 13 e 15). Dessa forma, fica demonstrado que o estado do segmento resultante da união é a união dos estados dos segmentos antigos.

□

O funcionamento das operações *SegFind* e *SegRequest* está relacionado ao funcionamento das camadas de *overlay* e suporte a replicação, respectivamente. Vale notar que o overlay apresentado na seção 3.1.1 garante entrega das mensagens com alta probabilidade, no entanto pode ocorrer a perda de uma mensagem caso as rotas utilizadas não sejam suficientemente disjuntas por conta da estrutura do *overlay*. Para simplificar o algoritmo, assumimos que a probabilidade de entrega é 1, no entanto na Seção 3.3 discutimos um mecanismo simples para lidar com esse problema. O teorema 3.14 (SegFind), à seguir, corresponde à Propriedade 3.3 (SegFind) do sistema.

Teorema 3.14 (SegFind)**.** *Se um nó correto p executa SegFind*(k,k')*, esta operação acaba por retornar (por meio de notificações SegFindOk*(S)*) um conjunto* $\mathbb{S}$ *de certificados de segmento válidos e que cobrem totalmente o intervalo de chaves buscado, isto é:*

$$(\forall S \in \mathbb{S} : Valid(S)) \wedge [k,k'] \subseteq \bigcup_{S \in \mathbb{S}} K(S)$$

Demonstração. O conjunto $\mathbb{S}$ contém os certificados notificados com *SegFindOk*. Como o teste da linha 7 (Algoritmo 1) envolve a chamada da notificação (linha 11), temos que somente certificados de segmento que passam pela verificação $Valid(S)$ são retornados, o que demonstra a primeira parte do teorema.

Para provar a segunda parte, com relação à cobertura das chaves buscadas, primeiramente verificamos que, segundo a linha 4, a mensagem de busca certamente chegará nos nós próximos à primeira chave do intervalo (k), visto que é a chave para qual a mensagem é endereçada. De acordo com o Lema 3.5 (Cobertura Total SegReconfigure), sempre existe um segmento responsável por k. Pela escolha dos parâmetros corretos do overlay (principalmente o tamanho do grupo de entrega) é possível garantir que pelo menos $2f+1$ nós desse segmento receberão a mensagem. Destes $2f+1$ nós, pelo menos $f+1$ são corretos e, por sua vez, segundo a linha 17, enviam o certificado ao cliente. Segundo o Lema 3.7 (Validade de Certificados SegReconfigure), todo nó correto possui um certificado de segmento válido e pelo Lema 3.8 (Consistência SegReconfigure) todos os nós corretos do mesmo segmento possuem o mesmo certificado de segmento. No lado do cliente, todo segmento válido recebido por $f+1$ nós é notificado (linha 11), logo, é garantida a notificação do segmento que cobre a primeira chave do intervalo de busca. Para os demais segmentos, basta verificar que os nós corretos que respondem à busca, em seguida (linha 18) testam se seu certificado de segmento cobre o fim da busca, caso contrário a requisição de busca é repassada para os segmentos subsequentes, utilizando novamente o overlay (linha 19), a fim de completar o intervalo $[k,k']$. A sequência se dá de maneira idêntica ao primeiro segmento. □

O Teorema 3.15 (Reconfigurações Finitas SegFind), a seguir, corresponde à Propriedade 3.4 (Reconfigurações Finitas SegFind) do sistema.

Teorema 3.15 (Reconfigurações Finitas SegFind)**.** *Existe um instante t tal que toda chamada $SegFind(k,k')$ executada por um nó correto p em $t' \geq t$ retorna um conjunto de certificados que permanecem ativos indefinidamente, ou seja:*

$$\forall t'' \geq t', S \in \mathbb{S} : Active(t'', S)$$

Demonstração. Conforme enunciado no modelo de sistema (seção 3.1), assumimos que o número de entradas e saídas de nós do sistema é finito. Por meio dessa premissa, o Lema 3.10 (Reconfigurações Finitas SegReconfigure) garante que o número de reconfigurações de segmentos é finito. Seja t o instante do término da última reconfiguração. A partir de t, todo segmento ativo permanece ativo indefinidamente. Pelo Algoritmo 1 (linha 17), temos que certificados corretos enviam certificados de segmentos atuais, ou seja, ativos

no instante da busca. Logo, para toda busca inciada em $t' \geq t$, os certificados retornados são de segmentos que permanecem ativos indefinidamente. □

O Teorema 3.16 (SegRequest), a seguir, corresponde às Propriedades 3.5 (Validade SegRequest), 3.6 (Acordo SegRequest) e 3.7 (Ordenação SegRequest) do sistema.

Teorema 3.16 (SegRequest)**.** *As operações SegRequest e SegDeliver apresentam as Propriedades 3.5 (Validade SegRequest), 3.6 (Acordo SegRequest) e 3.7 (Ordenação SegRequest).*

Demonstração. Pelo Algoritmo 2 (linha 5) temos que a chamada $SegRequest(S, req)$ implica em uma chamada $TOMulticast$ que envia req. Da mesma forma, a chamada $SegDeliver$ é resultado de uma notificação $TODeliver$ (linha 19). Dessa forma, as propriedades acima são garantidas pelas propriedades do algoritmo de RME utilizado na camada de suporte à replicação. □

O Teorema 3.17 (Terminação SegRequest), a seguir, corresponde à Propriedade 3.8 (Terminação SegRequest) do sistema.

Teorema 3.17 (Terminação SegRequest)**.** *Seja p um nó correto. Existe um instante t tal que, se S_q é um segmento obtido por uma chamada $SegFind(k,k')$ executada em $t' \geq t$, então uma requisição $SegRequest(S_q, req)$ executada por p acaba por ser entregue por meio de $SegDeliver(C_p, req)$ em todo nó correto de S_q.*

Demonstração. Segundo o Teorema 3.15 (Reconfigurações Finitas SegFind), existe um instante t a partir do qual toda chamada $SegFind$ retorna somente segmentos ativos. Se um segmento S_q permanece ativo, então uma chamada $SegRequest$ a S_q, segundo o Algoritmo 2, provoca uma chamada $TOMulticast$ (linha 5) que certamente será entregue com $TODeliver$, uma vez que a máquina de estados correspondente não é mais reconfigurada. Além disso, a mesma requisição leva o valor de $confId$ de S_q, que passa na verificação do Algoritmo 2 (linha 18). Dessa forma fica garantida a entrega, com $SegDeliver$ (linha 19), de toda requisição realizada após o instante t. □

O Teorema 3.18 (Segmento Inativo SegRequest), a seguir, corresponde à Propriedade 3.9 (Segmento Inativo SegRequest) do sistema.

Teorema 3.18 (Segmento Inativo SegRequest)**.** *Seja p um nó com certificado C_p. Se p executa $SegRequest(S_q, req)$ no instante t e S_q não está ativo nesse instante ($\neg Active(t, S_q)$), então nenhum nó correto de S_q executa $SegDeliver(C_p, req)$.*

Demonstração. Pelo Algoritmo 2 (linha 3), o valor de *knownId* é igual ao identificador de configuração do certificado S_q invocado. Pelo Lema 3.11 (Monotonicidade $confId$ SegReconfigure), sempre que um segmento reconfigura, o identificador de configuração do novo segmento é superior ao segmento antigo. Dessa forma, se um cliente realiza uma invocação a um segmento inativo, o valor de *knownId* utilizado é inferior ao $confId$ do segmento ativo. Essa característica, juntamente com o teste do Algoritmo 2 (linha 18) garante que nós corretos não entregam requisições enviadas para segmentos antigos (inativos). □

As operações *SegJoin* e *SegLeave* são baseadas nas operações *SegFind* e *SegRequest*, compostas de forma que a entrada e saída sejam executadas, com ordenação total, nos segmentos que correspondem corretamente ao identificador do nó tentando entrar ou sair. A prova das propriedades de *SegJoin* e *SegLeave*, portanto, dependem das propriedades provadas para *SegFind* e *SegRequest*.

O Lema 3.19 (Garantia Entrada SegReconfigure) garante que se um pedido de entrada foi registrado, o nó acabará por fazer parte de algum segmento. É uma das características cruciais de *SegJoin*.

Lema 3.19 (Garantia Entrada SegReconfigure)**.** *Se em algum nó correto de um segmento ativo S_q temos que $\langle +, C_p \rangle \in changes$, então p será membro de algum segmento resultante da reconfiguração de S_q.*

Demonstração. Pelo Lema 3.1 (Mesmo *changes* SegReconfigure), sabemos que $\langle +, C_p \rangle \in changes$ em todo nó correto de S_q. Pelos Lemas 3.2 (Início SegReconfigure) e 3.9 (Terminação SegReconfigure), sabemos também que uma reconfiguração inicia e termina em S_q. Por construção, no Algoritmo 6 (linha 15) estabelece que $C_p \in newMembers$. Sejam quais for os segmentos resultantes da reconfiguração, todos os nós contidos em *newMembers* passam a participar de algum desses segmentos. Isso pode ser visto no Algoritmo 8 (linha 5), no Algoritmo 9 (linhas 2, 3, 9 e 10) e no Algoritmo 11 (linhas 3 e 7). Pelos Lemas 3.12 (Acordo $TOReconfigure$ SegReconfigure) e 3.8 (Consistência SegReconfigure) temos que C_p configura a ME da mesma forma que os demais nós do novo segmento e executa com o mesmo estado da aplicação e mesmo certificado de segmento. □

O Teorema 3.20 (SegJoin), a seguir, corresponde à Propriedade 3.1 (SegJoin) do sistema.

Teorema 3.20 (SegJoin)**.** *Seja p um nó correto com certificado C_p, se p executa $SegJoin(C_p)$ em um instante t, então há um instante $t' \geq t$ no qual existe*

um segmento S_p ativo que contém p como membro, ou seja:

$$\exists S_p : Active(t', S_p) \wedge C_p.id \in K(S_p) \wedge C_p \in S_p.members$$

Demonstração. O Teorema 3.14 (SegFind) garante que a busca do Algoritmo 4 (linha 5) retorna um segmento responsável pelo identificador $C_p.id$. Além disso, o laço de repetição das linhas 4 a 8, juntamente com os Teoremas 3.17 (Terminação SegRequest) e 3.18 (Segmento Inativo SegRequest), garantem que em algum momento a requisição é enviada a um segmento ativo. Por consequência, o pedido de entrada termina por ser entregue em todos os nós corretos de um segmento ativo responsável pelo identificador de p. Essa entrega resulta no registro de p no conjunto *changes* dos nós corretos (linha 15) e, conforme os Lemas 3.2 (Início SegReconfigure) e 3.19 (Garantia Entrada SegReconfigure), uma reconfiguração termina por ocorrer e incluir p no novo segmento. □

De forma similar ao Lema 3.19 (Garantia Entrada SegReconfigure) com relação à entrada de nós, o Lema 3.21 (Garantia Saída SegReconfigure) demonstra que se um pedido de saída é registrado nos nós corretos do segmento, então o nó acabará por não fazer parte de nenhum segmento.

Lema 3.21 (Garantia Saída SegReconfigure)**.** *Se em algum nó correto de um segmento ativo S_q temos que $\langle -, C_p \rangle \in$ changes, então p não será membro de nenhum segmento resultante da reconfiguração de S_q.*

Demonstração. A prova é bastante similar ao Lema 3.19 (Garantia Entrada SegReconfigure), com a diferença de que no Algoritmo 8 (linha 15) fica evidente que $C_q \notin newMembers$. □

O Teorema 3.22 (SegLeave), a seguir, corresponde à Propriedade 3.2 (SegLeave) do sistema.

Teorema 3.22 (SegLeave)**.** *Sejam p um nó correto com certificado C_p e S_p um segmento válido tal que $C_p \in S_p.members$ e $Active(t, S_p)$ para um instante t. Se p executa $SegLeave(C_p)$ em t, então há um instante $t' \geq t$ no qual S_p deixa de ser ativo e nenhum segmento derivado da reconfiguração do mesmo contém p como membro, isto é:*

$$\neg Active(t', S_p) \wedge \forall S' : S_p \prec S' \rightarrow C_p \notin S'.members$$

Demonstração. De maneira similar à prova do Teorema 3.20 (SegJoin), no Algoritmo 5 também apresenta um laço onde é realizada uma invocação ao segmento do próprio nó p. Apesar de ser o próprio segmento de p, e a busca não ser necessária, ainda podem ocorrer reconfigurações concorrentemente à

operação *SegLeave*, tornando o segmento utilizado na requisição obsoleto. O laço utilizado, juntamente com os Teoremas 3.17 (Terminação SegRequest) e 3.18 (Segmento Inativo SegRequest), garante que haverá algum instante no qual o pedido de saída será recebido por todos os nós do segmento. Essa entrega implica no registro do pedido de saída no conjunto *changes* (linha 10) e, segundo os Lemas 3.2 (Início SegReconfigure) e 3.21 (Garantia Saída SegReconfigure), acabará por ocorrer uma reconfiguração que excluirá p dos novos segmentos gerados. □

3.3 DISCUSSÃO SOBRE A CAMADA DE SEGMENTAÇÃO

A camada de segmentação tem a função de permitir o uso eficiente de algoritmos de suporte à RME (Replicação Máquina de Estado) em ambientes de larga escala. Normalmente esses algoritmos possuem custo quadrático em função do número de participantes, o que torna pouco escalável e bastante custosa sua aplicação direta em redes P2P. Com a solução proposta neste trabalho, é possível prover confiabilidade e tolerância a intrusões para aplicações executando sobre redes P2P de grande porte, uma vez que que as técnicas de RME são aplicadas a um número de nós independente do crescimento do sistema. Apesar disso, a infraestrutura proposta apresenta custos adicionais próprios, principalmente com a necessidade de tratar entradas e saídas de nós, bem como a consequente reconfiguração das MEs.

A operação *SegFind* consiste de uma simples busca no *overlay* por um intervalo de chaves. O Algoritmo 1 descreve uma busca recursiva, na qual segmentos adicionais são buscados apenas depois que o segmento anterior é encontrado. Para grandes intervalos de chaves, é possível dividir os mesmos e realizar buscas em paralelo nos correspondentes subintervalos, porém um número grande de buscas paralelas pode levar a redundância, isto é, múltiplas buscas podem chegar ao mesmo segmento. A terminação da operação está ligada à garantia de entrega de mensagens provida pela camada de *overlay*. Como o *overlay* utilizado é probabilista, existe uma pequena chance da operação *SegFind* ficar bloqueada aguardando respostas porque o *overlay* falhou. Uma solução simples seria usar um temporizador que levaria à repetição da busca.

Outro aspecto importante da operação *SegFind* é a verificação, por meio do histórico de segmentos, da validade dos certificados de segmento recebidos na busca. Essa verificação pode implicar em um alto custo, uma vez que o histórico é um conjunto de segmentos que aumenta à medida que o sistema evolui. Uma otimização que diminui o custo de processamento consiste em manter em cada nó um cache de certificados válidos. Toda vez

que um certificado é validado, por meio da verificação das assinaturas, este é adicionado ao cache. Validações subsequentes do mesmo certificado não precisariam verificar as assinaturas, uma vez que este estaria presente no cache. Essa otimização tem um efeito importante, pois quanto mais antigo é um segmento, maior a probabilidade de ser compartilhado por vários históricos de segmentos mais atuais. Para reduzir o custo de transmissão, ao realizar uma busca, um nó pode enviar na requisição certificados de segmento contidos no cache local que interseccionam com o intervalo de chaves procurado. Caso os certificados enviados façam parte do histórico dos segmentos requisitados, os nós que responderem a requisição podem podar os históricos de certificados que atendem a demanda. Ou seja, os históricos não precisam ser enviados completos para as validações. São excluídos dos históricos todos os certificados anteriores aos certificados enviados na requisição de busca.

A operação *SegRequest* apresenta apenas o custo de uma invocação da RME, uma vez que o certificado de segmento contém os endereços dos nós correspondentes. Pode ocorrer, no entanto, que o certificado usado no momento da invocação esteja ultrapassado por conta de uma reconfiguração no segmento invocado. Nesse caso, os nós não responderão à requisição, e faz-se o uso de um temporizador para retirar o cliente da espera. Pode ocorrer de o temporizador estourar por causa de um atraso na rede e não por conta da reconfiguração do segmento e a repetição da invocação provocará uma execução dupla da requisição. Cabe à aplicação buscar novamente o certificado com *SegFind*, bem como garantir a idempotência das operações, por exemplo, com uso de *nonces*.

A repetição da requisição, por si só, não garante que esta será entregue nos nós corretos. É possível, graças ao assincronismo do sistema, que o tempo entre o recebimento do resultado da busca, no cliente, e o recebimento da requisição, nos nós servidores, seja sempre superior ao tempo entre as reconfigurações nos segmentos buscados. Neste caso, a requisição sempre ocorre com certificados de segmento obsoletos. Apesar de ser possível escolher um valor para o tempo entre as reconfigurações que diminua bastante a probabilidade dessa situação, para dar total garantia é preciso assumir um certo limite no dinamismo do sistema. É preciso assumir que apenas um número finito de reconfigurações (portanto, de entradas e saídas) pode concorrer com qualquer operação da aplicação. No entanto, assim como em outros trabalhos da literatura (AGUILERA et al., 2009), assumimos uma premissa mais simples, porém mais forte, a fim de simplificar as demonstrações das propriedades do sistema. Essa premissa é a de que o número total de reconfigurações (portanto, de entradas e saídas de nós) do sistema é finito.

A operação *SegJoin* faz uso das operações *SegFind* e *SegRequest* em um laço de repetição similar ao que seria usado na camada de aplicação.

Dessa forma, também depende da premissa descrita acima para terminar. A operação *SegLeave* é direcionada diretamente ao próprio segmento do nó cliente, ou seja, não é possível que o certificado de segmento seja antigo em nós corretos. Uma limitação dessa operação é que nós corretos precisam aguardar a próxima reconfiguração para saírem do sistema. Isso é necessário para que as requisições à ME, bem como a assinatura do próximo certificado e a transferência do estado da aplicação possam sempre ocorrer corretamente.

A operação *SegReconfigure* é a que apresenta o maior custo de toda a camada de segmentação por conta da necessidade de união e divisão de segmentos. No caso de uma reconfiguração simples (sem divisão nem união) há uma rodada de disseminação de assinaturas e depois a transferência de estado para os segmentos novos. No caso de divisão de segmento são duas assinaturas, porém o custo de mensagens é o mesmo e a transferência de estado é igualmente simples. A união, no entanto, é muito mais custosa, uma vez que envolve a invocação de outro segmento e a transferência de estado ocorre também entre os membros dos segmentos antigos antes desse estado ser enviado aos novos membros.

Os algoritmos de reconfiguração propostos na seção 3.2.3 apresentam algumas limitações com relação a cenários extremos com muitas entradas ou muitas saídas de nós simultâneas em regiões próximas do *overlay*. Se entre duas reconfigurações, o número de entradas de nó em um mesmo segmento for muito grande, pode ocorrer de a divisão resultar em dois segmentos ainda com número muito alto de nós. O Algoritmo 9, por simplicidade, assume que essa situação não ocorrerá, porém não é difícil alterar o mesmo algoritmo de forma que mais de dois segmentos sejam resultado da divisão: se o número de nós $n = \#newMembers$ (linha 1) for tal que $m \times n_{MIN} \leq n < (m+1) \times n_{MIN}$, para algum inteiro $m \geq 2$, divide-se o conjunto *newMembers* em m subconjuntos de nós com identificador contíguo com pelo menos n_{MIN} nós cada; o cálculo dos demais atributos dos segmentos, bem como a transferência do estado, ocorrem de maneira similar ao algoritmo atual, apenas com um número maior de segmentos. Essa alteração não aumenta o número de rodadas na execução do algoritmo, elevando o custo de maneira relevante apenas com relação ao número de assinaturas geradas.

Assim como muitos nós podem entrar em um mesmo segmento, muitos nós podem sair de segmentos vizinhos em um pequeno intervalo de tempo, de forma que a união de dois segmentos seja insuficiente para atingir o número de membros necessário para a execução dos algoritmos de RME. Mais uma vez, o Algoritmo 10 não lida com essa situação, porém é possível tornar o algoritmo recursivo, de forma que os próximos segmentos do *overlay* são unidos até que o número mínimo de nós seja atingido. Nesse caso, no entanto, diferente da divisão, são necessárias mais rodadas de comunicação pois mais

segmentos precisam ser contactados e reconfigurados.

Uma outra situação, também não tratada nos algoritmos por questão de simplicidade, é a seguinte: ao mesmo tempo que muitos nós saem de um segmento S, muitos nós entram no segmento sucessor S', de maneira que S faz um pedido de união com S', porém o conjunto de membros resultantes da união é maior que n_{MAX}. Nesse caso, a união deve incluir juntamente a lógica da divisão, de forma que mais de um segmento são resultantes da união. Por fim, em certos padrões de entradas e saídas extremos, poderia ocorrer combinação dessas situações, por exemplo, dois segmentos sequencias com tamanho reduzido acabam por tentar se unir com um terceiro com grande número de entradas de maneira que resultam da união três segmentos.

O alto custo da união de segmentos resulta, em grande parte, da comunicação e coordenação de duas máquinas de estados independentes. Para lidar com esse problema, neste trabalho utilizamos uma estratégia simples, acumulando pedidos de união em um subconjunto de $f+1$ nós, denominados de nós acumuladores, que realiza uma invocação ordenada ao segmento sucessor no *overlay* passando os pedidos acumulados como prova da validade da requisição de união. O uso de $f+1$ nós acumuladores garante que pelo menos um realiza a invocação ao segmento sucessor, porém também implica que pode haver mais de uma invocação para o mesmo pedido de união. Não há risco de reexecução da união, no entanto, uma vez que ao ordenar uma das requisições, a união é realizada e a RME é reconfigurada, descartando requisições enviadas para o segmento antigo.

A Tabela 2 apresenta aproximações (assintóticas) dos custos de mensagens e de rodadas relativos às operações da camada de segmentação em situações típicas, sem levar em conta as otimizações descritas nesta seção. Os valores são derivados dos algoritmos das Seções 3.2.1, 3.2.2 e 3.2.3 e assumem que $N_{Seg} = O(f)$ é o número médio de nós por segmento, K_{Seg} é o tamanho médio do intervalo de chaves dos segmentos, o custo da invocação na RME é $O(N_{Seg}^2)$ mensagens e demanda 5 rodadas (incluindo a invocação e a resposta ao cliente) (CASTRO; LISKOV, 2002), o custo esperado de um envio usando o *overlay* é $O(f \log N + f^2)$ mensagens e $\log N + 3$ rodadas, onde N é o número de nós no sistema (CASTRO et al., 2002).

Apesar de, na união de segmentos, a busca realizada para encontrar o segmento sucessor utilize o *overlay*, esta tende a terminar em poucos passos, uma vez que os segmentos são vizinhos. Dessa forma, a parcela logarítmica do custo da união é um pouco pessimista. Outra questão importante com relação aos custos é que o tempo de execução das operações *SegJoin* e *SegLeave* não é refletido pelo número de rodadas de comunicação, uma vez que essas operações envolvem uma espera pelo temporizador de reconfiguração ou por um pedido de união.

Tabela 2: Custos típicos aproximados das operações da camada de segmentação

Operação	**Mensagens**	**Rodadas**
SegFind	$O(f \log N + N_{Seg}^2)$	$\log N + 3 + \lceil \frac{k'-k}{K_{Seg}} \rceil$
SegRequest	$O(N_{Seg}^2)$	5
SegJoin	$O(f \log N + N_{Seg}^2)$	$\log N + 8 + \lceil \frac{k'-k}{K_{Seg}} \rceil$
SegLeave	$O(N_{Seg}^2)$	5
SegReconfigure	$O(N_{Seg}^2)$	8 (reconfiguração simples e divisão) ou $\log N + 14$ (união)

3.4 TRABALHOS RELACIONADOS

Rosebud (RODRIGUES; LISKOV, 2003) é um sistema de armazenamento distribuído que apresenta características similares a um *overlay* P2P estruturado. Para garantir disponibilidade e consistência diante de nós maliciosos, dados são replicados em $3f+1$ nós e quóruns de $2f+1$ nós são usados para realizar leitura e escrita. O sistema permite entrada e saída de nós por meio de um esquema de reconfiguração. Diferentemente da proposta deste trabalho, a reconfiguração é controlada por um subconjunto de nós do sistema, chamado de serviço de reconfiguração. A visão completa do sistema é mantida pelo serviço de reconfiguração e certificados contendo essa visão são disseminados a todos os nós a cada reconfiguração. No nosso trabalho, evitamos a necessidade de conhecimento completo do sistema, com o intuito de aliviar problemas de escala e também para mantermos a nossa proposta mais de acordo com a filosofia P2P que enfatiza a inexistência de gerenciamentos globais.

Castro et al. (2002) apresentam a proposta de um *overlay* P2P que garante alta probabilidade na entrega de mensagens mesmo quando uma parcela dos nós do sistema é maliciosa. A garantia de entrega é uma propriedade importante em redes P2P e permite a construção de soluções mais completas para prover tolerância a falhas e intrusões em redes P2P, como a que apresentamos neste trabalho. No próprio artigo, Castro et al. (2002) descrevem brevemente uma solução para leitura e escrita consistente de objetos mutáveis em uma rede P2P usando o roteamento confiável juntamente com técnicas de RME. Há certa similaridade com a solução proposta neste trabalho, porém aqui estendemos a replicação para suportar quaisquer operações, e tratamos de questões como entrada e saída de nós, além de união e divisão de segmen-

tos.

Bhattacharjee, Rodrigues e Kouznetsov (2007) definem uma solução de roteamento P2P tolerante a intrusões com base na divisão do sistema em segmentos dinâmicos que podem sofrer divisões ou uniões à medida que o sistema evolui. Porém, estes segmentos são em geral maiores dos que aqueles adotados neste trabalho e não estão associados a RMEs. Cada segmento possui um subconjunto de nós, o comitê, que participam de uma RME e têm a função de decidir sobre as reconfigurações do segmento, disseminando os certificados de novos segmentos. A confiabilidade das informações emitidas pelo comitê é garantida pelo uso de criptografia de limiar assimétrica (SCHULTZ; LISKOV; LISKOV, 2008) que permite a recriação do segredo compartilhado em caso de reconfiguração do comitê. Neste trabalho evitamos a criação de uma hierarquia, pois isso adicionaria complexidade, além de tornar o comitê um alvo de potenciais ataques e um possível gargalo nas reconfigurações. Além disso, adotamos o mecanismo de encadeamento de certificados de segmento como meio de verificação dos mesmos para evitar os altos custos de reconfiguração da criptografia de limiar (obtenção de novas chaves parciais em cada atualização dos segmentos).

PeerCube (ANCEAUME et al., 2008) é um *overlay* P2P estruturado do tipo DHT que visa minimizar as penalidades de desempenho causadas por *churn* e impedir que nós maliciosos subvertam o sistema. As principais propriedades do *overlay* são: a aplicação de uma estratégia de agrupamento para permitir operações baseadas em quóruns; uso de um algoritmo de inserção aleatória para diminuir a probabilidade de nós maliciosos corromperem grupos; estruturação dos grupos em um hipercubo que garante o funcionamento correto mesmo que alguns grupos sejam corrompidos. Os grupos do *PeerCube* se assemelham, em parte, aos segmentos propostos neste trabalho. Entradas e saídas de nós são tratadas localmente nos grupos e todos os membros do grupo são responsáveis pelos mesmos conjuntos de dados. Os grupos também utilizam protocolos de consenso para garantir consistência das visões e podem ser divididos e unidos a fim de manter o número de nós por grupo dentro de certos limites e manter o desempenho aceitável. Uma diferenã entre grupos e segmentos é que apenas um subconjunto dos nós do grupo, chamado de núcleo, é responsável por executar todos os protocolos, sendo que os demais, chamados suplentes, são réplicas passivas que simplesmente tentam manter suas cópias dos dados atualizadas. Qualquer nó que entre em um grupo se torna membro suplente e a saída de membros suplentes é simplesmente notificada aos nós do núcleo, que geram e disseminam a nova visão. Quando um nó do núcleo deseja sair, um novo núcleo é escolhido aleatoriamente entre todos os nós do grupo, incluindo suplentes. Essa separação permite tolerar *churn* dos suplentes, tornando custosa somente a saída de nós

do núcleo. A escolha aleatória do novo núcleo dificulta que um conjunto malicioso tome conta do grupo. O *PeerCube* oferece uma interface do tipo DHT, ou seja, objetos são associados a chaves pela operação *put* e podem ser obtidos pela operação *lookup*. Requisições são enviadas ao nó de destino usando caminhos disjuntos passando pelos grupos do hipercubo. Apesar de fazer uso de consenso bizantino, o *PeerCube* oferece funcionalidade similar ao *overlay* de Castro et al. (2002), porém com um foco maior em tolerar *churn*.

3.5 CONCLUSÃO DO CAPÍTULO

Este capítulo apresentou a proposta de infraestrutura tolerante a intrusões para construção de aplicações de memória compartilhada em redes P2P. As premissas do sistema, bem como as funcionalidades básicas das camadas inferiores do sistema foram descritas na Seção 3.1. As premissas do sistema não são diferentes das usualmente encontradas em outros sistemas presentes na literatura, e soluções existentes foram indicadas para cumprir os requisitos das camadas inferiores, dessa forma não prejudicando a viabilidade da proposta.

A Seção 3.2 apresentou a proposta propriamente dita. Iniciamos pela descrição geral da camada de segmentação, seguida pela formalização das operações desta camada e das propriedades que estas operações devem cumprir. Na sequência, as Seções 3.2.1, 3.2.2 e 3.2.3 detalharam os algoritmos relacionados às operações anteriormente descritas, acompanhados de explicações textuais dos passos desses algoritmos. A Seção 3.2.4 traz diversos lemas e teoremas associados às propriedades do sistema, seguidos de provas razoavelmente formais que determinam que os algoritmos propostos cumprem tais propriedades.

Além das provas do funcionamento correto dos algoritmos, a Seção 3.3 apresenta uma discussão sobre as limitações da infraestrutura e dos algoritmos propostos, bem como uma análise dos custos associados. Diversas otimizações são levantadas na mesma seção. Por fim, a Seção 3.4 compara a proposta deste trabalho com outros trabalhos similares da literatura.

4 ESPAÇO DE TUPLAS PAR A PAR TOLERANTE A INTRUSÕES

No Capítulo 3, apresentado anteriormente, definimos uma infraestrutura tolerante a intrusões para construção de aplicações de memória compartilhada sobre redes P2P. Um conjunto de operações com propriedades específicas para esse sistema são enunciadas e acompanhadas de algoritmos, cuja conformidade com as propriedades está provada. Agora, neste capítulo, tentamos demonstrar que as operações escolhidas são suficientes para construir aplicações de memória complexas e expressivas. Para isso, apresentamos uma construção de um **espaço de tuplas** tolerante a intrusões utilizando as operações da camada de segmentação.

Um Espaço de Tuplas (GELERNTER, 1985) (ET) é um mecanismo de coordenação de processos que apresenta desacoplamento espacial, isto é, processos não precisam conhecer os endereços um do outro para se comunicar; e desacoplamento temporal, isto é, processos não precisam estar ativos simultaneamente para efetivar a comunicação. Essas propriedades são garantidas pelo uso de uma memória associativa na qual as mensagens persistem e podem ser acessadas por buscas baseadas no conteúdo. Essas mensagens persistentes e endereçáveis por conteúdo são denominadas **tuplas** e a memória que as armazena é denominada de **espaço de tuplas**.

As características de desacoplamento espacial e temporal de um espaço de tuplas são especialmente interessantes para a comunicação em sistemas abertos e dinâmicos, como as redes P2P, nos quais processos não possuem conhecimento completo sobre os demais participantes do sistema. Nós podem se comunicar armazenando tuplas em um ET global que garante a persistência dessas informações, mesmo após a saída do nó que produziu a tupla. As tuplas ficam disponíveis a qualquer nó que realize uma busca. A definição e as propriedades dos espaços de tuplas em geral são apresentadas na Seção 4.1.

Para que o ET proposto possa fazer uso da infraestrutura descrita no Capítulo 3, é necessário cumprir com as exigências impostas pela camada de segmentação. A principal dessas exigências é com relação à indexação do estado da aplicação, que deve ser feita de forma a permitir a divisão e união de partes do estado de acordo com as divisões e uniões dos segmentos do sistema. De maneira geral, um espaço de tuplas é um multiconjunto de tuplas, isto é, uma coleção de tuplas que permite repetição de elementos idênticos. Indexamos o ET atribuindo uma chave a cada tupla e dividimos o mesmo em espaços menores contendo parte das tuplas com chaves numericamente próximas. Cada segmento S, conforme definido na camada de segmentação,

armazena as tuplas com chaves dentro do intervalo $K(S)$. Para atribuição das chaves, utilizamos um esquema de indexação multidimensional baseado em **Curvas de Preenchimento de Espaço** (*Space Filling Curves* ou SFCs em inglês), em especial a **Curva de Hilbert** (LAWDER; KING, 2000, 2001). A Curva de Hilbert apresenta a propriedade de ter boa localidade, isto é, tuplas com conteúdo próximo tendem a possuir índices próximos, o que facilita a realização de buscas por conteúdo. A Curva de Hilbert é apresentada na Seção 4.2. Os detalhes do esquema de indexação são descritos na Seção 4.3.

O ET proposto neste capítulo, por ser baseado na infraestrutura descrita no Capítulo 3, apresenta a propriedade de garantir a consistência das informações mesmo na presença de nós maliciosos. Isso assegura o funcionamento correto do ET, porém não provê características de segurança como privacidade de tuplas e controle de acesso, de modo que um nó maliciosos pode prejudicar o funcionamento da aplicação que utiliza o ET (p. ex. excluindo uma tupla de maneira indevida). Existem maneiras de garantir essas propriedades de segurança adicionais (BESSANI et al., 2008), porém isso está fora do escopo deste trabalho.

O restante do capítulo está dividido da seguinte forma: a Seção 4.1 apresenta o conceito e as propriedades dos espaços de tuplas; a Seção 4.2 descreve em linhas gerais as curvas de preenchimento de espaço e a curva de Hilbert, bem como o mecanismo de indexação de tuplas utilizado; a Seção 4.3 apresenta a construção do espaço de tuplas propriamente dito, utilizando as primitivas apresentadas no Capítulo 3; a Seção 4.5 aborda outras implementações de espaços de tuplas dinâmicos e compara as características dessas implementações com a proposta deste capítulo; a Seção 4.6 conclui o presente capítulo.

4.1 ESPAÇOS DE TUPLAS

Um **Espaço de Tuplas** (ET) é uma abstração de memória compartilhada entre processos na qual objetos chamados de **tuplas** são armazenados e recuperados (GELERNTER, 1985). Tuplas são persistentes, de forma que uma tupla pode permanecer no espaço mesmo após o processo que a inseriu ter terminado. Tuplas podem representar, entre outras coisas, resultados de computações, variáveis compartilhadas ou mensagens entre processos. O espaço de tuplas garante certas propriedades sobre as tuplas e sua manipulação, de forma que processos podem coordenar sua execução por meio das operações providas pelo ET.

Originalmente, a ideia dos espaços de tuplas foi concebida como abstração de comunicação de uma linguagem de programação concorrente deno-

minada Linda (GELERNTER, 1985). Posteriormente o conceito foi isolado e portado para diversas plataformas computacionais como forma de coordenação de processos em geral. Neste trabalho usamos uma versão simplificada da definição de espaço de tuplas voltada para a coordenação de processos em sistemas abertos.

Uma **tupla** é uma sequência de campos que podem assumir valores específicos, como 1, *João* e *TRUE*. São exemplos de tuplas: $\langle IDADE, João, 25\rangle$, $\langle 1,2\rangle$ e $\langle TRUE\rangle$. Para realização do endereçamento por conteúdo, tuplas são buscadas a partir de um **molde**, que nada mais é que uma tupla que pode ter zero ou mais dos seus campos indefinidos. A indefinição de um campo do molde é representada por $*$. São exemplos de moldes: $\langle IDADE, João, *\rangle$, $\langle 1, *\rangle$ e $\langle *\rangle$. Existe uma relação entre tuplas e moldes, denominada de **relação de casamento** (*matching*, em inglês), e denotada por $\leq_m$, na qual uma tupla t **casa** com um molde $\bar{t}$ (denota-se a relação por $t \leq_m \bar{t}$), se e somente se t e $\bar{t}$ contêm o mesmo número de campos e todos os campos definidos de $\bar{t}$ contém o mesmo valor que o campo correspondente de t. Caso contrário, diz-se que t **não casa** com $\bar{t}$ ($t \not\leq_m \bar{t}$). Alguns exemplos de casamentos são:

$$\langle IDADE, João, 25\rangle \leq_m \langle IDADE, João, 25\rangle,$$

$$\langle IDADE, João, 25\rangle \leq_m \langle IDADE, João, *\rangle,$$

$$\langle IDADE, João, 25\rangle \leq_m \langle IDADE, *, *\rangle,$$

$$\langle IDADE, João, 25\rangle \leq_m \langle *, João, *\rangle.$$

Por outro lado:

$$\langle IDADE, João, 25\rangle \not\leq_m \langle João, *\rangle,$$

$$\langle IDADE, João, 25\rangle \not\leq_m \langle João, 25\rangle,$$

$$\langle IDADE, João, 25\rangle \not\leq_m \langle IDADE, João, 15\rangle.$$

Na definição original da linguagem Linda (GELERNTER, 1985), campos de tuplas e moldes são tipificados, de maneira que é possível especificar um campo de molde que somente aceite valores de um tipo específico. Por exemplo, a tupla $\langle IDADE, 25\rangle$ casa com o molde $\langle IDADE, *Inteiro\rangle$, mas não com o molde $\langle IDADE, *Texto\rangle$. Diferente da concepção original, utili-

zamos uma definição simplificada onde campos não são tipificados. Dessa forma, não é possível realizar o casamento de tuplas por comparação de tipos diretamente, porém essa mesma funcionalidade pode ser obtida por meio da representação de tipos como valores nos campos das tuplas. Por exemplo:

$$\langle Texto, IDADE, Número, 25\rangle \leq_m \langle Texto, IDADE, Número, *\rangle,$$

porém,

$$\langle Texto, IDADE, Número, 25\rangle \not\leq_m \langle Texto, João, Texto, *\rangle.$$

Processos interagem com o espaço de tuplas por meio de operações para inserção (*out*), busca (*rd*) e remoção (*in*) de tuplas. Tradicionalmente, a semântica dessas operações segue as definições originais da linguagem Linda (GELERNTER, 1985): $out(t)$ é a operação que insere a tupla t no ET; a operação $rd(\bar{t})$ busca no ET uma tupla que case com o molde $\bar{t}$; e a operação $in(\bar{t})$ busca e remove do ET uma tupla que case com o molde $\bar{t}$[1]. Quando as operações *rd* e *in* são executadas, pode haver mais de uma tupla no ET que case com o molde passado. Nesse caso, uma das tuplas é escolhida de maneira não determinista para ser retornada e removida. Por outro lado, caso não exista nenhuma tupla no ET que case com o molde, as operações *rd* e *in* bloqueiam até que uma tupla adequada seja inserida. Quando uma tupla é inserida no ET, esta pode casar com os moldes especificados por mais de um processo bloqueado. Nessa situação, um desses processos é escolhido de maneira não determinista para ser liberado e retornar com a tupla. O espaço apresenta também versões não bloqueantes das operações *rd* e *in*, respectivamente *rdp* e *inp*, que retornam um valor especial $\perp$ caso nenhuma tupla do espaço case com o molde.

A abstração de espaço de tuplas permite um modelo de coordenação denominado de **comunicação generativa** (GELERNTER, 1985). Nesse modelo, processos podem se comunicar sem conhecer os endereços ou identificadores um do outro, característica denominada de desacoplamento espacial. Além disso, a comunicação generativa também apresenta a característica de desacoplamento temporal, pois processos não precisam estar ativos ao mesmo tempo para efetuar uma comunicação. No espaço de tuplas, um processo pode

[1] A linguagem Linda define ainda uma primitiva *eval* que dispara um novo processo e armazena o resultado do mesmo como uma tupla. Essa operação faz mais sentido em um sistema fechado, como um *cluster* ou um *grid*, contexto para o qual Linda foi projetada. A nossa proposta se trata de um espaço de tuplas sobre um sistema aberto onde processos correspondem a nós autônomos, logo decidimos por não suportar a primitiva *eval*.

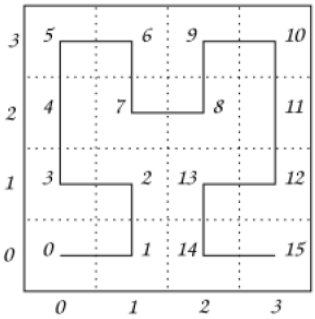

Figura 5: Desenho da curva de preenchimento de espaço bidimensional

inserir uma tupla e outro processo pode retirar essa tupla mesmo após a saída do processo que gerou a tupla. Outro termo utilizado para se referir a esse modelo é **coordenação dirigida por dados** (*data-driven coordination*, em inglês) (BUSI; GORRIERI; ZAVATTARO, 2000).

4.2 CURVAS DE PREENCHIMENTO DE ESPAÇO

Uma curva de preenchimento de espaço (*Space Filling Curve*, SFC) é um mapeamento bidirecional entre pontos de um hipercubo de D dimensões e pontos de uma linha, ou seja, é uma curva que passa por todos os pontos de um hipercubo D-dimensional exatamente uma vez (LAWDER; KING, 2000). SFCs foram um tópico de interesse dos matemáticos no fim do século XIX. Atualmente são usadas como mecanismo para criação de índices multidimensionais em bases de dados. A Figura 5 mostra um exemplo simples de uma SFC sobre um quadrado duas dimensões e coordenadas entre $(0,0)$ e $(3,3)$. Cada ponto do espaço é representado por um quadrado e a curva passa exatamente uma vez no centro de cada um desses quadrados.

Uma SFC pode ser usada como mecanismo de indexação de tuplas em um ET. Podemos tratar as tuplas como elementos de um espaço multidimensional, cada campo da tupla sendo uma coordenada, e podemos usar uma SFC para calcular um índice (ou chave) associado à tupla. Usando o exemplo da Figura 5, podemos indexar tuplas de dois campos com valores numéricos entre 0 e 3: a tupla $\langle 0,0\rangle$ tem índice 0, a tupla $\langle 1,0\rangle$ tem índice 1, a tupla $\langle 1,1\rangle$ tem índice 2, e assim sucessivamente. Esse exemplo é bastante simplificado e não permite a indexação de tuplas em situações mais complexas. Uma generalização desse princípio que permite a indexação de quaisquer tuplas é

apresentada na Seção 4.3.

Dado um mesmo espaço multidimensional, existem diversas formas de se indexar os pontos desse espaço, ou seja, existe mais de uma SFC que pode ser aplicada, e cada indexação apresenta propriedades distintas. A curva de Hilbert é um tipo de SFC que apresenta propriedades de localidade, no sentido de que pontos adjacentes na curva unidimensional são também adjacentes no espaço multidimensional (LAWDER; KING, 2000). Essa localidade facilita as buscas por conteúdo necessárias em um ET, uma vez que tuplas que casem com o mesmo molde tem chance maior de terem índices próximos. Usando ainda o exemplo da Figura 5 (que, de fato, ilustra uma curva de Hilbert), podemos ver que uma busca usando o molde $\langle 1, * \rangle$ precisa localizar tuplas com índices nos intervalos $[1,2]$ e $[6,7]$.

De maneira geral, dado um espaço multidimensional de D dimensões, uma curva de Hilbert de ordem k divide o espaço em 2^{Dk} subespaços iguais e trata o centro desses subespaços como os pontos pelos quais a curva deve passar. Dessa forma, uma curva de ordem k divide cada eixo do espaço em k níveis e possui 2^{Dk} pontos. A Figura 6 ilustra as três primeiras ordens de uma curva de Hilbert em duas dimensões. A curva de primeira ordem (a) divide o quadrado em quatro partes e percorre os quadrantes em sentido horário. A curva de segunda ordem (b) é construída a partir da curva de primeira ordem pela substituição de cada quadrante por uma curva de primeira ordem rotacionada. A curva de terceira ordem (c) é obtida de maneira similar. Em geral, a curva de ordem $k+1$ é construída a partir da curva de ordem k dividindo cada ponto desta curva em um subespaço de 2^D pontos e traçando uma curva de ordem 1 rotacionada de forma a se conectar com os subespaços vizinhos. Mais detalhes sobre o procedimento de cálculo podem ser encontrados em (LAWDER; KING, 2000).

A curva completa é obtida aplicando a definição recursiva infinitamente, porém para a construção de índices são usadas as curvas de ordem finita. A ordem da curva está relacionada à quantidade de informação (número de bits) necessária para representar as coordenadas dos pontos do espaço e dos pontos da curva unidimensional. Em geral, dado um espaço D-dimensional e uma curva de k-ésima ordem, cada coordenada dos pontos do espaço precisam de k bits e um ponto qualquer (tanto no espaço multidimensional quanto na curva unidimensional) precisa de $D \times k$ bits. Assumimos que, dado um ponto x no espaço multidimensional, $Hilbert(x)$ representa o ponto na curva de Hilbert correspondente a x.

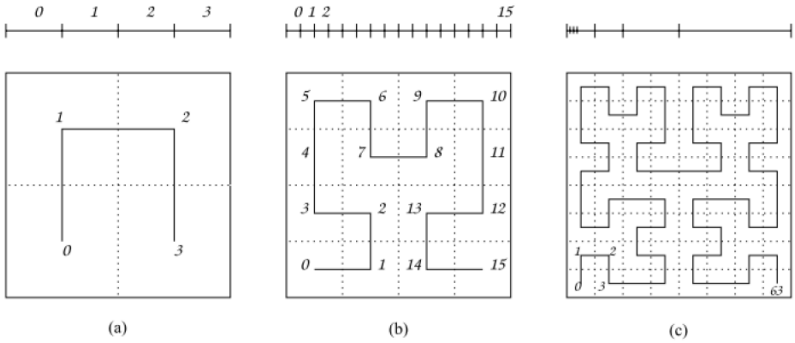

Figura 6: Desenho da curva de Hilbert bidimensional nas três primeiras ordens

4.3 ESPAÇO DE TUPLAS TOLERANTE A INTRUSÕES

A camada de espaço de tuplas (ET), construída sobre a camada de segmentação, implementa as operações *out*, *rd*, *rdp*, *in* e *inp* com a mesma semântica descrita na Seção 4.1. A indexação do espaço de tuplas, conforme necessário para a utilização da camada de segmentação, é realizada pela atribuição de chaves a tuplas individuais e de conjuntos de chaves a moldes. De maneira geral, se uma tupla t possui chave k, então qualquer molde $\bar{t}$, tal que $t \leq_m \bar{t}$, deve possuir um conjunto de chaves $\bar{K}$ tal que $k \in \bar{K}$. Dessa maneira, se a tupla t é armazenada no segmento responsável pela chave k, então uma busca cobrindo as chaves $\bar{K}$ termina por encontrar t. Usando essa estratégia, não é necessário que as chaves associadas às tuplas sejam únicas, isto é, tuplas diferentes podem possuir a mesma chave. Basta que a chave associada à tupla aponte sempre para o segmento que armazena a tupla. Isso é possível por conta da generalidade das operações providas pela camada de segmentação. Se uma interface do tipo DHT com operações *put* e *get* fosse provida, tuplas com chaves iguais seriam tratadas como colisões e metadados adicionais seriam necessários para decidir qual tupla está sendo buscada.

Uma maneira trivial de satisfazer essa restrição é atribuir a mesma chave a todas as tuplas e o mesmo conjunto contendo essa chave a todos os moldes, no entanto essa solução concentra todas as tuplas em um mesmo segmento. É interessante que a atribuição de chaves às tuplas aumente as chances de uma distribuição uniforme ao longo dos segmentos do sistema para tornar menos provável uma sobrecarga em segmentos específicos. Outra solução trivial é fazer com que os conjuntos de chaves associados aos moldes

cubram todo o espaço de chaves, porém o desempenho das buscas nesse caso degrada linearmente com o crescimento do número de segmentos. É interessante que a abrangência da busca reflita a abrangência do molde utilizado, ou seja, quanto mais campos indefinidos, mais tuplas podem casar com o molde e mais segmentos precisam ser buscados.

A Curva de Hilbert é utilizada para atribuir chaves a tuplas e conjuntos de chaves a moldes, de forma a possibilitar que as tuplas inseridas no ET sejam posteriormente encontradas. Seja D um parâmetro global do sistema que indica o número máximo de dimensões de uma tupla. A **chave** relacionada a uma tupla $t = \langle t_1, \ldots, t_l \rangle$, onde $l \leq D$, é calculada da seguinte forma: a tupla é normalizada para $D+1$ dimensões na forma $t' = \langle l, t_1, \ldots, t_l, \perp_{l+1}, \ldots, \perp_D \rangle$. Na sequência, é calculado o *hash* (ex. SHA-1) de cada campo $t''_i = hash(t'_i)$, $\forall i \in [1, D+1]$; a tupla $t'' = \langle t''_1, \ldots, t''_{D+1} \rangle$ é tratada como um ponto no espaço $D+1$-dimensional e é mapeada em um índice pela função $k = Hilbert(t'')$. Esse procedimento é representado pela função $TupleKey(t)$. Para inserir a tupla t, calcula-se a chave $k = TupleKey(t)$ e insere-se a tupla no segmento responsável por k usando a camada de segmentação.

Note que é possível que tuplas distintas possuam a mesma chave, portanto $k \in \bar{K}$ não necessariamente implica que $t \leq_m \bar{t}$. Colisões de chaves de tuplas podem resultar de colisões dos *hashes* das coordenadas ou de truncamentos realizados pela função $Hilbert$, uma vez que esta recebe D parâmetros com 128 bits cada (se for usado SHA-1) e retorna um número do mesmo tamanho dos identificadores do *overlay*. Se o número de bits do identificador do *overlay* é menor que D multiplicado pelo número de bits da função *hash* utilizada, então alguma informação é perdida pela função $Hilbert$ de maneira que colisões podem ocorrer. No entanto, como a camada de segmentação permite a execução de requisições arbitrárias, podemos lidar com colisões enviando o próprio molde ao segmento responsável pelas chaves cobertas na busca. Dessa forma, as tuplas são encontradas localmente usando o molde e não somente a chave, logo apenas tuplas que efetivamente casam com o molde são retornadas. Dessa forma, fica claro que a chave calculada para as tuplas não tem o papel de identificador, mas sim de estabelecer um local inequívoco para a tupla e permitir a busca eficiente. Como espaços de tuplas são baseados em busca por conteúdo, nenhuma semântica é perdida pela existência de colisões.

Para encontrar uma tupla a partir de um molde $\bar{t}$, procede-se de maneira similar ao procedimento para inserir uma tupla. Primeiro o molde é normalizado para $D+1$ dimensões, depois é calculado o *hash* de cada campo, porém em vez de calcular uma única chave com a função $Hilbert$, é necessário encontrar o conjunto de todas as chaves na curva unidimensional que correspondem às possíveis tuplas que casam com $\bar{t}$. Esse procedimento é

representado pela função $TemplateKeys(\bar{t})$. Por não ser o escopo principal deste trabalho, não abordamos em detalhes o procedimento de computação das chaves usando a curva de Hilbert. Mais informações podem ser obtidas a partir de outros trabalhos da literatura que também utilizam o mesmo mecanismo de indexação para realizar buscas multidimensionais e por intervalos (LI; PARASHAR, 2005; LEE et al., 2005; SHEN et al., 2008).

Na camada de espaço de tuplas, cada nó possui um espaço de tuplas local, denominado de ts. Para que o algoritmo de replicação garanta a consistência, é necessário que o espaço de tuplas seja idêntico em todas as réplicas e que retorne o mesmo resultado quando invocado com os mesmos parâmetros. Isso não é garantido pela semântica básica de um espaço de tuplas e implementações diferentes podem retornar resultados diferentes. Para evitar esse problema e garantir o determinismo das réplicas, pode-se assumir o uso de uma única implementação por todos os nós corretos do sistema ou pode-se aumentar a semântica do espaço de tuplas local com propriedades de ordenação sobre as tuplas. O importante é assegurar que, quando um molde $\bar{t}$ casa com mais de uma tupla contida em ts, réplicas distintas de um mesmo segmento que realizem uma busca usando $\bar{t}$ retornem a mesma tupla consistentemente.

- ts: é uma implementação local de espaços de tuplas. Essa implementação provê os métodos $ts.out(t)$, $ts.rdp(\bar{t})$ e $ts.inp(\bar{t})$ com a semântica usual de espaços de tuplas. Além disso, essa estrutura contém um atributo $ts.limits$ que representa um intervalo tal que toda tupla t com $TupleKey(t) \notin ts.limits$ é descartada. A operação de união de dois espaços de tuplas locais ($ts' \cup ts''$), com intervalos de limite consecutivos, é dada pela união de todas as tuplas contidas nos dois operandos e pela união dos intervalos $limits$. Note que $ts.limits$ corresponde ao intervalo de chaves do segmento conforme definido na camada de segmentação;

- *pending*: é um conjunto com entradas da forma $\langle C_i, \bar{t} \rangle$ que indicam que o nó i está aguardando uma tupla que case com o molde $\bar{t}$.

O Algoritmo 12 descreve como o espaço de tuplas responde às chamadas da camada de segmentação para obter e alterar o estado. Na chamada *SegGetAppState* são retornadas as tuplas dentro do intervalo especificado (linha 2) e as buscas pendentes nas quais as chaves do molde tem intersecção com o intervalo (linha 3). Na chamada *SegSetAppState* são recebidos um intervalo de chaves e uma lista de estados parciais que precisam ser unidos. Os limites de ts são alterados para o intervalo passado (linha 7), as tuplas de todos os estados parciais são unidas e armazenadas em ts (linha 8) e as buscas pendentes também são reunidas (linha 9).

Algoritmo 12 Obtenção e alteração do estado local do Espaço de Tuplas

```
    /* Upcall da camada de segmentação para obter estado */
 1: upon SegGetAppState(start, end) do
        /* Tuplas com chave dentro do intervalo */
 2:     T ← {t ∈ ts : TupleKey(t) ∈ [start, end)}
        /* Pendencias que interseccionam com intervalo */
 3:     P ← {⟨C_i, t̄⟩ ∈ pending : TemplateKeys(t̄) ∪ [start, end) ≠ ∅}
 4:     return ⟨T, P⟩
 5: end upon

    /* Upcall da camada de segmentação para alterar estado */
 6: upon SegSetAppState(start, end, ⟨⟨T^1, P^1⟩, ..., ⟨T^n, P^n⟩⟩) do
        /* Altera limites do espaço de tuplas local */
 7:     ts.limits ← [start, end)
        /* Guarda união das tuplas dos subestados */
 8:     ts ← ⋃_{i=1}^{n} T_i
        /* Guarda união das pendências dos subestados */
 9:     pending ← ⋃_{i=1}^{n} P_i
10: end upon
```

4.3.1 Inserção de Tupla

A inserção de tupla consiste em calcular a chave relacionada à tupla a ser inserida, buscar o segmento responsável por essa chave e enviar a tupla para inserção. Essa busca e inserção devem ser repetidas até que a resposta correta seja recebida. O uso de um *nonce* garante que a mesma requisição não é executada mais de uma vez. No servidor, ao receber uma requisição de inserção, o nó insere a tupla no espaço local ts e procura alguma busca pendente $\langle C_i, \bar{t} \rangle$ na qual a tupla inserida case com o molde $\bar{t}$. Caso alguma pendência correspondente seja encontrada, o nó C_i, registrado como iniciador da busca, é notificado para que possa prosseguir com a leitura ou exclusão. Note que a exclusão em si não é executada nesse ponto, ou seja, um nó registrado ao tentar uma excluir uma tupla, ao receber a notificação, precisa ainda realizar uma requisição de exclusão. Mais detalhes sobre a exclusão de tuplas serão apresentados na Seção 4.3.4.

A operação $out(t)$ (Algoritmo 13) inicia com o cálculo da chave da tupla t por meio da função $TupleKey$ (linha 2). Depois disso, um valor único (*nonce*) é gerado. Na sequência, um laço de repetição é iniciado para buscar o segmento responsável pela tupla (linhas 6 e 7) e invocar o segmento passando a requisição para inserir a tupla (linha 8). No lado dos nós servidores, quando

Algoritmo 13 Inserção de tuplas (*out*)

```
 1: operation out(t)
    /* Código do cliente p */
        /* Gera chave da tupla */
 2:     k ← TupleKey(t)
        /* Gera nonce para requisição */
 3:     nonce ← GenerateNonce()
        /* Busca e invoca segmento responsável pela chave da tupla */
 4:     resp ←⊥
 5:     while resp =⊥ do
 6:         SegFind(k,k)
 7:         wait for SegFindOk(S_q)
 8:         resp ← SegRequest(S_q, ⟨OUT, nonce, t⟩)
 9:     end while

    /* Código do servidor q */
10:     upon SegDeliver(C_p, ⟨OUT, nonce, t⟩) do
            /* Calcula chave da tupla e verifica se está nos limites do seg-
    mento */
11:         if TupleKey(t) ∈ ts.limits then
                /* Insere tupla no espaço local */
12:             ts.out(t)
                /* Responde para cliente */
13:             SegResponse(C_p, ⟨OUT_OK⟩)
                /* Notifica clientes pendentes interessados na nova tupla */
14:             for all ⟨C_i, t̄⟩ ∈ pending : t ≤_m t̄ do
15:                 SegNotify(C_i, ⟨READ_OK, t⟩)
16:                 pending ← pending \ {⟨C_i, t̄⟩}
17:             end for
18:         end if
19:     end upon
20: end operation
```

uma requisição para inserir uma tupla é recebida (linha 10), a chave da tupla é verificada (linha 11) e caso a tupla pertença realmente ao segmento esta é inserida no espaço de tuplas local (linha 12). Na sequência, é verificado se a tupla inserida casa com alguma busca pendente registrada no segmento (linha 14) e para cada caso em que a tupla casar com o molde uma notificação é enviada ao nó que efetuou a busca em questão (linha 15). Além disso, a pendência é removida (linha 16).

4.3.2 Busca de Tupla

As operações de leitura e exclusão de tuplas apresentam uma característica similar: calculam as chaves relacionadas ao molde de busca e varrem todos os segmentos responsáveis por essas chaves a fim de descobrir quais segmentos possuem tuplas que casem com um molde. Após encontrar os segmentos relevantes, as operações de leitura simplesmente retornam as tuplas encontradas, enquanto as operações de exclusão invocam o segmento para remover a tupla. Dada essa similaridade, decidimos escrever um procedimento genérico, que é utilizado pelas operações de leitura e exclusão, e que realiza a busca nos segmentos e notifica as tuplas que forem encontradas. Esse procedimento foi idealizado para ser executado em uma *thread* separada para que a busca de tuplas possa ocorrer de maneira concorrente.

A busca de uma tupla a partir de um molde consiste em calcular as chaves do molde, buscar todos os segmentos responsáveis por chaves contidas no conjunto de chaves e consultar os segmentos para ler uma tupla que case com o molde buscado. Se uma tupla adequada for encontrada durante a consulta, esta é notificada para a *thread* principal da operação. Por outro lado, se nenhuma tupla for encontrada, então o algoritmo é diferente para as operações bloqueantes e não bloqueantes: no primeiro caso, o algoritmo bloqueia para receber notificações de segmentos sempre que uma tupla que casa com o molde for encontrada; no caso não bloqueante a busca termina imediatamente. No lado do servidor, os procedimentos também são bastante similares: ao receber um pedido de leitura, o nó busca seu espaço de tuplas local usando rdp e retorna o resultado, seja este uma tupla ou $\perp$, para o cliente. Se a leitura for bloqueante e uma tupla adequada não estiver presente (resultado $\perp$), então o servidor, além de responder, também registra o pedido na lista de pendências para que possa notificar o cliente posteriormente.

Como pode ser notado na descrição acima, a busca de tupla é bastante similar nos casos bloqueante e não bloqueante, portanto decidimos fazer um algoritmo geral $FindTuple$ (Algoritmos 14 e 15) que engloba as duas funcionalidades e recebe um parâmetro *block* que indica o caso específico. As

operações específicas simplesmente invocam *FindTuple* passando o molde e o parâmetro *block* adequado. A cada tupla t encontrada no segmento S, o procedimento faz uma notificação por meio da chamada $FindTupleOk(S,t)$. Essa notificação é tratada de maneira diferente pelas operações de leitura e de exclusão.

A ideia geral da busca de tuplas é bastante simples, porém há um ponto que adiciona complexidade ao algoritmo: a possibilidade de uma consulta a um segmento não ser executada efetivamente e retornar $\perp$. Para lidar com esses casos, é necessário construir o algoritmo com várias "passadas" pelas chaves remanescentes, cujos segmentos responsáveis ainda não puderam ser consultados, até que todo o conjunto seja efetivamente coberto.

O procedimento *FindTuple*, no lado do cliente (Algoritmo 14), inicia pelo cálculo do conjunto de chaves associado ao molde da busca (linha 2). Depois, o algoritmo entra em um laço de repetição (linhas 3 a 21) que se repete até que todas as chaves sejam efetivamente buscadas. Na primeira passada do laço, as chaves buscadas são exatamente as chaves do molde recém calculadas. À medida que segmentos forem consultados, o conjunto de chaves remanescentes é reduzido até que não reste nenhuma chave a ser buscada e o laço termina.

No interior do laço de repetição, dois conjuntos são criados para controlar as chaves que foram buscadas e os segmentos consultados na passada atual, respectivamente $foundKeys$ (linha 4) e *doneKeys* (linha 5). Na sequência, o conjunto de chaves remanescente é dividido em intervalos contíguos e para cada intervalo de chaves, *SegFind* é invocado a fim de encontrar os segmentos responsáveis (linha 7).

Depois de realizar *SegFind* em todas as chaves, um outro laço de repetição (linhas 9 a 19) realiza o recebimento dos certificados de segmentos buscados na etapa anterior e a consulta dos segmentos correspondentes. Para cada segmento encontrado (linha 10), as chaves correspondentes são marcadas como encontradas (linha 11) e o segmento é invocado para buscar tuplas que casem com o molde (linha 12). Além do molde, a requisição contém também o parâmetro *block*, que indica para os nós do segmento se o cliente deve ser registrado como pendente caso a tupla não seja encontrada.

Depois de invocado o segmento, a resposta recebida é testada para saber se a invocação foi realizada de fato (linha 13). Se a resposta foi efetivamente recebida, as chaves do segmento são marcadas como consultadas (linha 14) e o valor recebido é testado (linha 15) para saber se uma tupla que casa com o molde foi encontrada. Se for o caso, o algoritmo notifica a *thread* principal sobre a tupla encontrada.

Depois que todos os segmentos foram invocados, ou seja, depois que o conjunto $foundKeys$ passa a conter todas as chaves remanescentes, o teste

Algoritmo 14 Busca de tuplas (*FindTuple*) (1 de 2)

```
 1: procedure FindTuple(t̄, block)
    /* Código do cliente p */
        /* Calcula intervalos de chaves do molde */
 2:     remainingKeys ← TemplateKeys(t̄)
        /* Repete até que todas as chaves do molde sejam cobertas */
 3:     while remainingKeys ≠ ∅ do
            /* Chaves que tiveram segmento encontrado */
 4:         foundKeys ← ∅
            /* Chaves que tiveram segmento consultado com sucesso */
 5:         doneKeys ← ∅
            /* Busca segmentos de todas as chaves pendentes */
 6:         for all [k,k'] ∈ remainingKeys do
 7:             SegFind(k,k')
 8:         end for
            /* Repete até que todas as chaves pendentes tenham segmento encontrado */
 9:         while remainingKeys ⊈ foundKeys do
                /* Segmento encontrado */
10:             wait for SegFindOk(S_i)
                /* Marca chaves do segmento encontrado */
11:             foundKeys ← foundKeys ∪ K(S_i)
                /* Consulta segmento */
12:             resp ← SegRequest(S_i, ⟨READ, t̄, block⟩)
13:             if resp = ⟨READ_OK, t⟩ then
                    /* Se consultou com sucesso, marca chaves do segmento */
14:                 doneKeys ← doneKeys ∪ K(S_i)
                    /* Se encontrou tupla, realiza notificação */
15:                 if t ≠ ⊥ then
16:                     FindTupleOk(S_i, t)
17:                 end if
18:             end if
19:         end while
            /* Não encontrou tupla nessa passada, remove chaves consultadas */
20:         remainingKeys ← remainingKeys \ doneKeys
21:     end while
        /* Todas as chaves do molde foram consultadas e nenhuma tupla encontrada. */
22:     if block then
            /* Busca bloqueante, aguarda notificação com tupla */
23:         wait for SegNotifyDeliver(S_i, ⟨READ_OK, t⟩)
            /* Retorna tupla notificada */
24:         FindTupleOk(S_i, t)
25:     else
            /* Busca não-bloqueante, termina sem aguardar notificação */
26:         return
27:     end if
28: end procedure
```

Algoritmo 15 Busca de tuplas (*FindTuple*) (2 de 2)

```
    /* Código do servidor q */
1:  upon SegDeliver(C_p, ⟨READ, t̄, block⟩) do
        /* Busca tupla localmente */
2:      t ← ts.rdp(t̄)
        /* Verifica necessidade de registrar pendência */
3:      if block ∧ t = ⊥ then
            /* Cliente marcado como pendente */
4:          pending ← pending ∪ {⟨C_p, t̄⟩}
5:      end if
        /* Envia resposta contendo tupla ou ⊥ */
6:      SegResponse(C_p, ⟨READ_OK, t⟩)
7:  end upon
```

da linha 9 falha e o laço interno termina. Depois disso, todas as chaves de segmentos que foram efetivamente consultados são removidas das chaves remanescentes (linha 20). Se todas as chaves foram consultadas, então o laço de repetição externo termina. A partir desse ponto, o cliente age de maneira diferente dependendo do parâmetro *block*: se $block = TRUE$, então os segmentos consultados terão sido informados que o cliente deseja ser registrado como pendente (linha 12), o cliente aguarda alguma notificação vinda de um segmento (linha 23) e repassa qualquer tupla informada (linha 24); se $block = FALSE$, então o algoritmo termina.

No lado do servidor (Algoritmo 15), qualquer segmento que recebe uma requisição para leitura de tupla realiza uma busca no espaço de tuplas local (linha 2) e verifica, caso a tupla não exista, se o cliente está realizando uma busca bloqueante (linha 3). Em caso afirmativo, o cliente é registrado nas pendências (linha 4) e será notificado quando uma tupla adequada for inserida, conforme a operação *out* (Algoritmo 13, linha 15). Finalmente, a requisição é respondida com o resultado da busca local (linha 6).

4.3.3 Leitura de Tupla

As operações leitura de tupla, *rd* e *rdp*, utilizam o procedimento de busca de tupla descrito na Seção 4.3.2 e simplesmente retornam a primeira tupla encontrada. Dessa forma, o código principal (Algoritmo 16) das operações é bem simples e consiste apenas em iniciar a *thread* de busca, aguardar uma notificação e retornar a tupla encontrada. Em todo caso, a operação termina ou quando uma tupla é retornada, ou quando o procedimento de busca

termina. As condições de término do procedimento de busca são diferentes para as operações bloqueantes e não bloqueantes, conforme descrito na Seção 4.3.2.

Algoritmo 16 Leitura de tuplas bloqueante (*rd*) e não bloqueante (*rdp*)

```
1:  operation rd(t̄)
    /* Código do cliente */
        /* Inicia procedimento de busca de tupla */
2:      FindTuple(t̄, TRUE)
        /* Aguarda notificação de tupla encontrada */
3:      wait for FindTupleOk(S_i, t)
        /* Retorna a tupla */
4:      return t
5:  end operation

6:  operation rdp(t̄)
    /* Código do cliente */
        /* Inicia procedimento de busca de tupla */
7:      FindTuple(t̄, FALSE)
        /* Recebe notificação de tupla encontrada ou término da busca */
8:      upon FindTupleOk(S_i, t) do
            /* Retorna a tupla */
9:          return t
10:     end upon
11: end operation
```

4.3.4 Exclusão de Tupla

O procedimento de exclusão de tupla, efetivado pelas operações *in* e *inp*, apresenta estrutura similar ao de leitura, o espaço de chaves que corresponde ao molde é buscado para encontrar tuplas que casem, porém a diferença é que sempre que uma tupla é encontrada em um segmento, uma nova invocação é realizada para tentar excluir a tupla. Essa tentativa de exclusão é passível de falha, uma vez que outro cliente pode remover a mesma tupla antes. Essa estratégia de separar a operação em duas fases, de busca e de exclusão, permite que a fase de busca seja executada em paralelo para intervalos de chaves distintos, melhorando o desempenho para buscas grandes.

É importante que as invocações para exclusão sejam realizadas em série, uma vez que somente uma tupla deve ser excluída em cada chamada de

in ou *inp*. Além disso, caso uma requisição de exclusão falhe (com uma resposta $\perp$), esta precisa ser repetida até que um resultado seja obtido indicando se a tupla foi ou não removida, pois se a requisição foi executada e a resposta simplesmente atrasou, a operação deve parar e retornar a tupla removida, evitando de remover outra tupla.

As operações de exclusão bloqueante e não bloqueante, isto é, *in* e *inp*, apresentam procedimentos bastante similares. Uma diferença é com relação ao tipo de busca, ou seja, o parâmetro *block* do procedimento *FindTuple*, que é *TRUE* para *in* e *FALSE* para *inp*. A outra diferença é com relação às tentativas de remover uma tupla encontrada: na operação bloqueante, quando o cliente tenta remover uma tupla que não está mais presente no espaço de tuplas, este deve ser registrado novamente como pendente no segmento; na operação não bloqueante isso não é necessário. Assim, sendo pequenas as diferenças, resolvemos construir um procedimento único que recebe um parâmetro *block*, de maneira similar ao procedimento *FindTuple* (Seção 4.3.2).

O procedimento *RemoveTuple* (Algoritmo 17) incia chamando o procedimento *FindTuple* passando o molde e o parâmetro *block* (linha 2). Na sequência, ocorre o tratamento das tuplas que são encontradas durante a busca (linha 3). Primeiro um *nonce* é gerado (linha 4) a fim de evitar remover mais de uma vez a mesma tupla e garantir idempotência. Depois o segmento responsável pela tupla encontrada é invocado para remover a tupla (linha 5). Essa requisição inclui a tupla a ser removida e o parâmetro *block*, que determina se o cliente deve ser registrado como pendente caso a tupla não exista mais no espaço. Conforme explicado anteriormente, se a requisição falhar, esta deve ser repetida até que uma resposta seja efetivamente recebida. Isso é realizado por um laço de repetição que busca novamente o segmento responsável pela tupla (linha 7) e envia novamente a requisição de exclusão para o segmento encontrado (linha 9). Quando a resposta é recebida, se esta contiver a tupla e não o valor $\perp$, então a tupla é retornada e o procedimento termina.

No lado do servidor, os nós do segmento invocado para excluir uma tupla agem de maneira bastante similar ao caso da busca de tupla (Seção 4.3.2). A diferença é que *inp*, em vez de *rdp*, é chamado no espaço de tuplas local, e uma tupla específica é buscada, em vez de um molde. Assim como na busca, se a tupla não é encontrada e se $block = TRUE$, o cliente é registrado como pendente.

As operações *in* e *inp* (Algoritmo 18) são implementadas simplesmente chamando *RemoveTuple* passando o molde e o valor do parâmetro *block* adequado.

Algoritmo 17 Exclusão de tuplas (bloqueante ou não)

```
 1: procedure RemoveTuple(t̄, block)
    /* Código do cliente p */
        /* Inicia procedimento de busca de tuplas */
        /* Quando (se) a busca termina, RemoveTuple também termina */
 2:     FindTuple(t̄, block)
        /* Tupla encontrada */
 3:     upon FindTupleOk(S_i, t) do
            /* nonce evita que mesmo molde remova mais de uma tupla */
 4:         nonce ← GenerateNonce()
            /* Envia requisição para segmento remover tupla */
 5:         resp ← SegRequest(S_i, ⟨REMOVE, t, block, nonce⟩)
            /* Repete até que resposta seja recebida */
 6:         while resp = ⊥ do
                /* Busca novamente segmento responsável pela tupla */
 7:             SegFind(TupleKey(t))
 8:             wait for SegFindOk(S_j)
                /* Repete invocação com mesmo nonce */
 9:             resp ← SegRequest(S_j, ⟨REMOVE, t, block, nonce⟩)
10:         end while
11:         if resp = ⟨REMOVE_OK, t⟩ ∧ t ≠ ⊥ then
                /* Retorna tupla que foi removida */
12:             return t
13:         end if
14:     end upon
15: end procedure

    /* Código do servidor q */
16: upon SegDeliver(C_p, ⟨REMOVE, t, block, nonce⟩) do
        /* Busca e exclui tupla localmente */
17:     t ← ts.inp(t)
        /* Verifica necessidade de registrar pendência */
18:     if block ∧ t = ⊥ then
            /* Cliente marcado como pendente */
19:         pending ← pending ∪ {⟨C_p, t̄⟩}
20:     end if
        /* Envia resposta contendo a tupla ou ⊥ */
21:     SegResponse(C_p, ⟨REMOVE_OK, t⟩)
22: end upon
```

Algoritmo 18 Exclusão de tuplas bloqueante (*in*) e não bloqueante (*inp*)

```
1: operation in(t̄)
   /* Código do cliente p */
       /* Inicia procedimento de remover tuplas bloqueante */
2:     return RemoveTuple(t̄, TRUE)
3: end operation

4: operation inp(t̄)
   /* Código do cliente p */
       /* Inicia procedimento de remover tuplas não bloqueante */
5:     return RemoveTuple(t̄, FALSE)
6: end operation
```

4.4 DISCUSSÃO SOBRE O ESPAÇO DE TUPLAS

Os algoritmos apresentados na Seção 4.3 implementam um espaço de tuplas sobre um *overlay* P2P e possuem características de tolerância a intrusões suportadas pela camada de segmentação proposta no Capítulo 3. Os algoritmos fazem uso das primitivas da camada de segmentação para prover uma abstração de comunicação e coordenação de alto nível para redes P2P. Graças às propriedades das operações da camada de Segmentação, o espaço de tuplas apresenta garantias de consistência mesmo com a presença de certo número de nós maliciosos.

A operação *out*, descrita na Seção 4.3.1, consiste simplesmente em calcular a chave da tupla usando a curva de Hilbert, depois buscar e invocar o segmento responsável pela chave calculada. Conforme as propriedades das operações *SegFind* e *SegRequest*, a busca e invocação podem precisar ser repetidas caso o segmento buscado reconfigure antes da invocação ou caso a resposta demore a chegar no nó cliente. Para o segundo caso, o algoritmo utiliza um *nonce* que evita a repetição da inserção da tupla, o que violaria a semântica da operação *out*.

As operações *rd* e *rdp*, descritas na Seção 4.3.3, consistem em uma varredura do conjunto de chaves representado pelo molde buscado, de forma que cada segmento responsável por chaves deste conjunto é consultado sobre a existência de uma tupla adequada. De acordo com as propriedades das curvas de preenchimento de espaço, quanto mais geral é o molde utilizado, maior é o conjunto de chaves coberto e maior é o número de segmentos consultados. Buscas por moldes mais gerais, portanto, implicam em um maior número de trocas de mensagens, porém a consulta a diferentes segmentos é realizada em paralelo, reduzindo o tempo de resposta geral das operações *rd* e *rdp*. Assim

como na inserção de tupla, é preciso repetir a busca e consulta a segmentos sempre que o temporizador estourar em uma invocação. No caso da operação bloqueante *rd*, a busca, além de consultar os segmentos responsáveis pelas chaves do molde, também aguarda notificações posteriores caso nenhuma tupla seja encontrada, de maneira a garantir a semântica da operação.

As operações *in* e *inp*, descritas na Seção 4.3.4, iniciam de maneira similar às operações de leitura de tuplas, isto é, realizam uma varredura em paralelo nos segmentos responsáveis pelas chaves do molde a fim de encontrar tuplas que casem com o mesmo. No caso de *in*, assim como *rd*, a operação também aguarda o recebimento de notificações posteriores. No entanto, após encontrar uma tupla, o algoritmo precisa invocar novamente o segmento responsável para remover a tupla encontrada. Pode ocorrer de algum outro nó conseguir remover a tupla antes, e nesse caso a busca por tuplas continua. Caso mais de uma tupla seja encontrada concomitantemente, é importante que as invocações de exclusão sejam realizadas em série. Isso é para evitar que uma única operação *in* ou *inp* termine na exclusão de mais de uma tupla, o que consiste uma violação da semântica. Como a exclusão da tupla é uma invocação destrutiva, então um *nonce* também é utilizado como na inserção de tupla.

A operação *out*, além de inserir a tupla, também realiza a verificação das buscas pendentes e notifica todos os nós bloqueados com moldes que casem com a tupla inserida. Todo nó notificado é excluído do conjunto de pendências. Isso não é problema para nós bloqueados em operações *rd*, pois a simples notificação de uma tupla basta para encerrar o bloqueio. Por outro lado, no caso de nós bloqueados em operações *in*, o nó precisa ainda tentar remover a tupla para terminar a operação e caso a mesma tupla acabe sendo excluída antes, o nó precisa continuar bloqueado. Por esse motivo a invocação para excluir a tupla pode fazer com que o nó entre novamente no conjunto de pendências. Uma característica interessante desse mecanismo de notificação é que se múltiplos nós bloqueados aceitam uma mesma tupla, a finalização de todas as leituras é garantida, mesmo que haja uma ou mais exclusões pendentes. Isso não viola a semântica das operações e até reduz a probabilidade de certas condições de *starvation* causadas pelo não determinismo na escolha do nó que acessa a tupla inserida.

Uma propriedade semântica importante do espaço de tuplas é que uma tupla não pode ser lida ou excluída sem que tenha sido antes inserida. Em outras palavras, uma operação de leitura ou exclusão não pode retornar uma tupla t se não tiver ocorrido uma operação $out(t)$. Nesta solução, essa propriedade é garantida (em nós corretos) mesmo na presença de certo número de nós maliciosos. Enquanto os limites de faltas da camada de segmentação forem mantidos, nenhum nó consegue corromper o estado global do espaço de

tuplas. Apesar disso, como não tratamos de questões de segurança de acesso ao espaço de tuplas, qualquer nó malicioso pode invocar $out(t)$ a qualquer momento, mesmo que isso não esteja de acordo com a aplicação executando sobre o espaço de tuplas. De maneira similar, uma tupla removida não pode mais ser lida ou removida, o que também é garantido pelas propriedades de consistência asseguradas pela camada de segmentação, porém um nó malicioso pode remover tuplas quaisquer usando moldes arbitrários. Uma forma de proteger as aplicações contra as ações de nós maliciosos é implementar funcionalidades de controle de acesso às tuplas (BESSANI et al., 2008).

4.5 TRABALHOS RELACIONADOS

Linda (GELERNTER, 1985) é uma linguagem de coordenação de processos distribuídos que introduziu o conceito de espaços de tuplas e comunicação generativa. Gelernter (1985) define esquemas eficientes para armazenamento e localização de tuplas no espaço de tuplas global baseados na identificação de padrões e situações frequentes na manipulação do ET. No entanto, as otimizações apresentadas dependem de um mecanismo centralizado de compilação dos processos (descritos em Linda), o que não é possível em um sistema distribuído aberto. Além disso, o ET em Linda, por si só, não possui garantia de consistência na presença de nós maliciosos ou intrusões.

Diversas implementações de espaços de tuplas foram desenvolvidas e incluídas em plataformas de coordenação, como JavaSpaces[2], parte da plataforma Jini, e IBM TSpaces[3]. Essas soluções, no entanto, armazenam as tuplas em um servidor centralizado, o que apresenta limitações de escala ou obriga a aplicação a dividir a coordenação em múltiplos espaços de tuplas lógicos, o que nem sempre é viável. Além disso, um servidor centralizado é um ponto único de falha e deve ser totalmente confiável.

Existem diversas soluções que aplicam técnicas de replicação para garantir disponibilidade e confiabilidade ao espaço de tuplas (XU; LISKOV, 1989; HANSEN; CANNON, 1994; BAKKEN; SCHLICHTING, 1995; BESSANI et al., 2008), em especial, Bessani et al. (2008) definem um espaço de tuplas que agrega tolerância a faltas bizantinas e propriedades de segurança como confidencialidade e controle de acesso. Apesar de aumentar a confiabilidade do espaço de tuplas, essas soluções ainda centralizam as tuplas em um conjunto fixo de processos, seguindo um modelo cliente/servidor, o que apresenta problemas de escala.

Lime (PICCO; MURPHY; ROMAN, 1999) é uma implementação de

[2]http://www.jini.org/wiki/JavaSpaces_Specification
[3]http://www.almaden.ibm.com/cs/TSpaces/

espaços de tuplas para redes móveis. Redes móveis apresentam diversas características em comum com redes P2P, como dinamismo e ausência de gerência global. Lime inclui em cada nó da rede um espaço de tuplas local com o qual o nó interage diretamente. Uma visão distribuída do espaço de tuplas é construída a partir da detecção de outros nós vizinhos na rede. Os espaços de tuplas de nós vizinhos são agrupados e o nó passa a acessar tuplas localizadas nos vizinhos por meio desse espaço de tuplas aumentado. Apesar da distribuição, o sistema não tolera saídas de nós ou falhas, uma vez que as tuplas do espaço de tuplas local não são replicadas.

Comet (LI; PARASHAR, 2005) é uma implementação de espaços de tuplas sobre uma DHT *Chord*. Assim como neste trabalho, *Comet* utiliza curvas de Hilbert para distribuir as tuplas pelos nós do *overlay*. No entanto, os autores apenas citam, como trabalho futuro, o uso de replicação para tolerar apenas faltas de parada (*crash*, em inglês). A ideia de utilizar curvas de Hilbert para indexar tuplas foi inspirada nesse trabalho. Comet suporta especificação de moldes mais complexos, envolvendo limites nos valores não especificados.

4.6 CONCLUSÃO DO CAPÍTULO

Este Capítulo apresentou uma construção de espaço de tuplas sobre a infraestrutura de segmentação proposta no Capítulo 3. O objetivo foi avaliar a expressividade das operações providas pela camada de segmentação enquanto suporte para construção de aplicações de memória compartilhada de larga escala e tolerantes a intrusões. A semântica de espaço de tuplas adotada, incluindo as operações suportadas, foram descritas na Seção 4.1. O espaço de tuplas foi construído conforme a semântica comum (GELERNTER, 1985), e tanto as operações bloqueantes quanto não bloqueantes foram implementadas.

O mecanismo de indexação de tuplas adotado foi apresentado na Seção 4.2. A ideia de utilizar curvas de preenchimento de espaço (LAWDER; KING, 2000) foi retirada de outros trabalhos da literatura e a escolha se deu devido à simplicidade desse mecanismo e a suas características de facilitar buscas por conteúdo.

A Seção 4.3 apresenta a construção do espaço de tuplas propriamente dito. Os algoritmos relativos às operações do ET são descritos nas Seções 4.3.1, 4.3.2, 4.3.3 e 4.3.4. Esses algoritmos não tiveram sua correção provada formalmente, porém uma discussão detalhada do seu funcionamento e suas limitações resultou na Seção 4.4. Por fim, a Seção 4.5 compara o espaços de tuplas apresentado nas Seções anteriores com outros trabalhos similares na

literatura.

5 CONCLUSÃO

Este Capítulo conclui a dissertação. A Seção 5.1 apresenta uma revisão do trabalho. A Seção 5.2 evidencia as suas contribuições e compara as mesmas com os objetivos enumerados na Seção 1.2. Por fim, a Seção 5.3 lista algumas possibilidades de trabalhos futuros que preenchem lacunas, estendem ou complementam funcionalidades.

5.1 REVISÃO

Este trabalho tratou da elaboração e avaliação de uma infraestrutura para a construção de aplicações de memória distribuída compartilhada de larga escala. Essa infraestrutura é baseada em um *overlay* P2P estruturado, e garante a consistência das informações e estruturas de dados mesmo se uma parte dos processos componentes apresentar comportamento malicioso. A ideia central consiste em dividir o *overlay* P2P em grupos menores a fim de tornar eficiente a aplicação de protocolos de Replicação Máquina de Estados (RME), que por sua vez garantem tolerância a intrusões.

Primeiramente, a solução foi delineada por meio da especificação de um modelo de sistema distribuído dinâmico (Seção 3.1), a partir do qual foi estabelecida uma arquitetura em camadas, tal que cada camada é formada por um conjunto de operações com propriedades específicas. A camada mais inferior é a camada de rede, que fornece as bases para o sistema distribuído, com operações para trocas de mensagens entre processos sobre canais ponto a ponto confiáveis. Acima da rede, duas camadas independentes funcionalidades distintas: a camada de *overlay* (Seção 3.1.1) implementa um *overlay* P2P com características de tolerância a intrusões que permite a entrega de mensagens a um grupo de nós responsáveis por uma chave; a camada de suporte à replicação (Seção 3.1.2) implementa um protocolo de Replicação Máquina de Estados reconfigurável a partir de um conjunto de nós.

Acima dessas duas camadas, foi construída a camada de segmentação (Seção 3.2), que divide o *overlay* em segmentos compostos por nós contíguos e aplica a Replicação Máquina de Estados reconfigurável a esses nós. De maneira geral, as operações do *overlay* são utilizadas para encontrar segmentos específicos e o protocolo de RME é utilizado para submeter requisições a esses segmentos. Além disso, a camada de segmentação é responsável por realizar o gerenciamento dos grupos de nós e permitir a entrada e saída de nós em segmentos, promovendo a divisão de segmentos grandes e a união de segmentos pequenos de maneira a garantir a execução correta e eficiente dos

protocolos de RME.

A fim de avaliar a expressividade das operações providas pela camada de segmentação, um espaço de tuplas foi elaborado tendo como base a infraestrutura proposta. As operações do espaço de tuplas foram construídas com a semântica usual, incluindo as propriedades de desacoplamento espacial e temporal. As tuplas são indexadas por meio do uso de curvas de preenchimento de espaço do tipo curvas de Hilbert, que apresentam boa localidade. As propriedades da camada de segmentação garantem a consistência do espaço de tuplas desenvolvido.

5.2 REVISÃO DOS OBJETIVOS E CONTRIBUIÇÕES

Os objetivos da dissertação estão enumerados na Seção 1.2. A seguir, esses objetivos são re enumerados e as partes da dissertação que apresentam contribuições relacionadas ao cumprimento desses objetivos.

- **estudar modelos de sistemas distribuídos dinâmicos e redes P2P, bem como abstrações de memória compartilhada, a fim de definir precisamente o problema e elaborar o modelo de sistema adotado**;
  O modelo de sistema distribuído escolhido teve como base o estudo de outros sistemas distribuídos dinâmicos e está descrito na Seção 3.1. Esse modelo possui um conjunto de premissas compatível com outros trabalhos encontrados na literatura.

- **propor uma infraestrutura para construção de aplicações conforme o objetivo geral descrito acima e definir as propriedades garantidas pela infraestrutura**;
  A Seção 3.2 apresenta a camada de segmentação, que fornece um conjunto de operações para a construção de aplicações de memória compartilhada. Essas operações são acompanhadas de propriedades que definem precisamente a semântica da segmentação.

- **especificar o funcionamento da infraestrutura por meio de algoritmos distribuídos, definidos em um nível de detalhe que permita demonstrar a adequação às propriedades enumeradas**;
  As Subseções 3.2.1, 3.2.2 e 3.2.3 contêm algoritmos que implementam as operações da camada de segmentação. A ideia central e o funcionamento de cada algoritmo são descritos textualmente a fim de facilitar o entendimento dos mesmos.

- **demonstrar por meio de provas de teoremas e lemas que os algoritmos elaborados se adequam às propriedades da infraestrutura**

proposta;
A Seção 3.2.4 contém um conjunto de teoremas e lemas que demonstram que os algoritmos elaborados cumprem as propriedades da camada de segmentação. As provas elaboradas são bastante abrangentes e consistentes, de forma que promovem a confiança nos algoritmos.

- **analisar a complexidade e os custos dos algoritmos elaborados**;
  A Seção 3.3 apresenta uma discussão ampla das limitações e dos custos dos algoritmos da camada de segmentação. Nenhuma implementação dos algoritmos foi realizada, portanto não foi possível coletar dados experimentais acerca dos custos da infraestrutura. Isso está previsto como um importante trabalho a ser desenvolvido futuramente (ver Seção 5.3).

- **realizar um estudo de caso por meio da construção de uma aplicação de memória compartilhada expressiva sobre a infraestrutura proposta**;
  O Capítulo 4 apresenta a elaboração e análise de um espaço de tuplas construído sobre a camada de segmentação. Assim como na camada de segmentação, o espaço de tuplas foi definido na forma de operações e propriedades e algoritmos que obedecem essas propriedades foram descritos. Uma análise informal dos algoritmos foi realizada, mas não com a mesma profundidade e rigor da camada de segmentação.

5.3 PERSPECTIVAS FUTURAS

As contribuições dessa dissertação são importantes no sentido de criar uma base para uma série de estudos futuros sobre tolerância a intrusões em sistemas distribuídos dinâmicos e P2P. Os algoritmos, as análises e provas representam uma verificação conceitual e uma abordagem inicial que pode ser expandida, generalizada e aprimorada. A formalização realizada constitui uma contribuição importante na medida em que dá certa segurança de que bons resultados serão obtidos em estudos futuros.

Um primeiro trabalho futuro importante consiste em realizar uma implementação dos algoritmos descritos nessa dissertação, tanto da camada de segmentação quanto do espaço de tuplas. Apesar da ampla formalização realizada neste trabalho, uma implementação executável é necessária para que se tenha noção de aspectos mais práticos relacionados à execução do sistema. Acreditamos que o nível de formalização utilizado nos algoritmos e nas análises deixa bastante claro a viabilidade de uma implementação correta da infraestrutura. A avaliação do desempenho da infraestrutura em um sistema real é de fundamental importância para permitir elaborar novas otimi-

zações e aprimoramentos, e essa análise é possibilitada imediatamente após a implementação dos algoritmos.

Uma ideia que surgiu durante o desenvolvimento do trabalho mas não foi devidamente explorada, é o uso de múltiplos algoritmos de consenso diferentes para execução da Replicação Máquina de Estados, de acordo com o número de nós no segmento entre duas reconfigurações. Por exemplo, se após uma reconfiguração, o número de nós do segmento passa a ser $5f + 1$, é possível utilizar algoritmos de consenso que executam em dois passos de comunicação em situações favoráveis (MARTIN; ALVISI, 2006b). Se o número chega a $7f + 1$ é possível utilizar algoritmos de consenso de um passo (SONG; RENESSE, 2008). Essa otimização pode aliviar problemas de escala resultantes do uso de protocolos de consenso e, em algumas situações, adiar a necessidade de dividir segmentos.

REFERÊNCIAS BIBLIOGRÁFICAS

ABERER, K.; CUDRÉ-MAUROUX, P.; DATTA, A.; DESPOTOVIC, Z.; HAUSWIRTH, M.; PUNCEVA, M.; SCHMIDT, R. P-grid: a self-organizing structured p2p system. *SIGMOD Rec.*, ACM, New York, NY, USA, v. 32, n. 3, p. 29–33, set. 2003. ISSN 0163-5808. <http://doi.acm.org/10.1145/945721.945729>.

AGUILERA, M. K. A pleasant stroll through the land of infinitely many creatures. *SIGACT News*, ACM, New York, NY, USA, v. 35, n. 2, p. 36–59, jun. 2004. ISSN 0163-5700. <http://doi.acm.org/10.1145/992287.992298>.

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*. New York, NY, USA: ACM, 2009. (PODC '09), p. 17–25. ISBN 978-1-60558-396-9. <http://doi.acm.org/10.1145/1582716.1582726>.

ALCHIERI, E.; BESSANI, A.; GREVE, F.; FRAGA, J. da S. Memória compartilhada em sistemas bizantinos dinâmicos. In: *Anais do XIII Workshop de Teste e Tolerância a Falhas (WTF 2012)*. Ouro Preto - MG: [s.n.], 2012. <http://sbrc2012.dcc.ufmg.br/app/pdfs/p-05/wtf/WTF-ST2-2.pdf>.

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: [s.n.], 2008. p. 169 – 182. <http://www.lbd.dcc.ufmg.br/bdbcomp/servlet/Trabalho?id=9742>.

ANCEAUME, E.; LUDINARD, R.; RAVOAJA, A.; BRASILEIRO, F. Peercube: A hypercube-based p2p overlay robust against collusion and churn. In: *Self-Adaptive and Self-Organizing Systems, 2008. SASO '08. Second IEEE International Conference on*. [s.n.], 2008. p. 15 –24. <http://dx.doi.org/10.1109/SASO.2008.44>.

ATTIYA, C.; DOLEV, D.; GIL, J. Asynchronous byzantine consensus. In: *Proceedings of the third annual ACM symposium on Principles of distributed computing*. New York, NY, USA: ACM, 1984. (PODC '84), p. 119–133. ISBN 0-89791-143-1. <http://doi.acm.org/10.1145/800222.806740>.

ATTIYA, H.; WELCH, J. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. 2. ed. [S.l.]: John Wiley & Sons, 2004. (Wiley Series on Parallel and Distributed Computing). ISBN 0471453242.

BAKKEN, D.; SCHLICHTING, R. Supporting fault-tolerant parallel programming in linda. *Parallel and Distributed Systems, IEEE Transactions on*, v. 6, n. 3, p. 287 –302, mar 1995. ISSN 1045-9219. <http://dx.doi.org/10.1109/71.372777>.

BALAKRISHNAN, H.; KAASHOEK, M. F.; KARGER, D.; MORRIS, R.; STOICA, I. Looking up data in p2p systems. *Commun. ACM*, ACM, New York, NY, USA, v. 46, n. 2, p. 43–48, fev. 2003. ISSN 0001-0782. <http://doi.acm.org/10.1145/606272.606299>.

BALDONI, R.; BERTIER, M.; RAYNAL, M.; TUCCI-PIERGIOVANNI, S. Looking for a definition of dynamic distributed systems. In: MALYSHKIN, V. (Ed.). *Parallel Computing Technologies*. Berlin: Springer, 2007, (Lecture Notes in Computer Science, v. 4671). p. 1–14. ISBN 978-3-540-73939-5. <http://dx.doi.org/10.1007/978-3-540-73940-1_1>.

BESSANI, A. N.; ALCHIERI, E. P.; CORREIA, M.; FRAGA, J. S. Depspace: a byzantine fault-tolerant coordination service. *SIGOPS Oper. Syst. Rev.*, ACM, New York, NY, USA, v. 42, n. 4, p. 163–176, abr. 2008. ISSN 0163-5980. <http://doi.acm.org/10.1145/1357010.1352610>.

BHATTACHARJEE, B.; RODRIGUES, R.; KOUZNETSOV, P. Secure lookup without (constrained) flooding. In: CORREIA, M.; NEVES, N. F. (Ed.). *Proceedings of the Workshop on Recent Advances on Intrusion-Tolerant Systems (WRAITS'07)*. [s.n.], 2007. p. 13 – 17. <http://wraits07.di.fc.ul.pt/8.pdf>.

BÖGER, D. S.; FRAGA, J.; ALCHIERI, E.; WANGHAM, M. Segmentação de overlays p2p como suporte para memórias tolerantes a intrusões. In: *XI Simpósio Brasileiro Segurança da Informação e Sistemas Computacionais*. Brasília - DF: SBC, 2011. p. 155 – 168. <http://www.lbd.dcc.ufmg.br/colecoes/sbseg/2011/0011.pdf>.

BÖGER, D. S.; FRAGA, J.; ALCHIERI, E.; WANGHAM, M. Intrusion-tolerant shared memory through a p2p overlay segmentation. In: *Advanced Information Networking and Applications (AINA), 2012 IEEE 26th International Conference on*. Fukuoka, JP: IEEE, 2012. p. 779 –786. ISSN 1550-445X. <http://dx.doi.org/10.1109/AINA.2012.104>.

BUSI, N.; GORRIERI, R.; ZAVATTARO, G. On the expressiveness of linda coordination primitives. *Information and Computation*, v. 156, n. 1 - 2, p. 90 – 121, 2000. ISSN 0890-5401. <http://www.sciencedirect.com/science/article/pii/S0890540199928237>.

CASTRO, M.; DRUSCHEL, P.; GANESH, A.; ROWSTRON, A.; WALLACH, D. S. Secure routing for structured peer-to-peer overlay networks. *SIGOPS Oper. Syst. Rev.*, ACM, New York, NY, USA, v. 36, n. SI, p. 299–314, dez. 2002. ISSN 0163-5980. <http://doi.acm.org/10.1145/844128.844156>.

CASTRO, M.; LISKOV, B. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, ACM, New York, NY, USA, v. 20, n. 4, p. 398–461, nov. 2002. ISSN 0734-2071. <http://doi.acm.org/10.1145/571637.571640>.

CAVIN, D.; SASSON, Y.; SCHIPER, A. Consensus with unknown participants or fundamental self-organization. In: NIKOLAIDIS, I.; BARBEAU, M.; KRANAKIS, E. (Ed.). *Ad-Hoc, Mobile, and Wireless Networks*. Springer Berlin / Heidelberg, 2004, (Lecture Notes in Computer Science, v. 3158). p. 630–630. ISBN 978-3-540-22543-0. <http://dx.doi.org/10.1007/978-3-540-28634-9_11>.

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

CLARKE, I.; SANDBERG, O.; WILEY, B.; HONG, T. Freenet: A distributed anonymous information storage and retrieval system. In: FEDERRATH, H. (Ed.). *Designing Privacy Enhancing Technologies*. Springer Berlin / Heidelberg, 2001, (Lecture Notes in Computer Science, v. 2009). p. 46–66. ISBN 978-3-540-41724-8. <http://dx.doi.org/10.1007/3-540-44702-4_4>.

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, p. 82–96, January 2006. <http://comjnl.oxfordjournals.org/content/49/1/82.abstract>.

DÉFAGO, X.; SCHIPER, A.; URBÁN, P. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 36, n. 4, p. 372–421, dez. 2004. ISSN 0360-0300. <http://doi.acm.org/10.1145/1041680.1041682>.

DWORK, C.; LYNCH, N.; STOCKMEYER, L. Consensus in the presence of partial synchrony. *J. ACM*, ACM, New York, NY, USA, v. 35, n. 2, p. 288–323, abr. 1988. ISSN 0004-5411. <http://doi.acm.org/10.1145/42282.42283>.

FISCHER, M. J.; LYNCH, N. A.; PATERSON, M. S. Impossibility of distributed consensus with one faulty process. In: *Proceedings of the 2nd ACM SIGACT-SIGMOD symposium on Principles of database systems*. New York, NY, USA: ACM, 1983. (PODS '83), p. 1–7. ISBN 0-89791-097-4. <http://doi.acm.org/10.1145/588058.588060>.

GELERNTER, D. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, ACM, New York, NY, USA, v. 7, n. 1, p. 80–112, jan. 1985. ISSN 0164-0925. <http://doi.acm.org/10.1145/2363.2433>.

GIFFORD, D. K. Weighted voting for replicated data. In: *Proceedings of the seventh ACM symposium on Operating systems principles*. New York, NY, USA: ACM, 1979. (SOSP '79), p. 150–162. ISBN 0-89791-009-5. <http://doi.acm.org/10.1145/800215.806583>.

GODFREY, P. B.; SHENKER, S.; STOICA, I. Minimizing churn in distributed systems. *SIGCOMM Comput. Commun. Rev.*, ACM, New York, NY, USA, v. 36, n. 4, p. 147–158, ago. 2006. ISSN 0146-4833. <http://doi.acm.org/10.1145/1151659.1159931>.

GREVE, F.; TIXEUIL, S. Knowledge connectivity vs. synchrony requirements for fault-tolerant agreement in unknown networks. In: *Dependable Systems and Networks, 2007. DSN '07. 37th Annual IEEE/IFIP International Conference on*. [s.n.], 2007. p. 82 –91. <http://dx.doi.org/10.1109/DSN.2007.61>.

GREVE, F.; TIXEUIL, S. Conditions for the solvability of fault-tolerant consensus in asynchronous unknown networks: invited paper. In: *Proceedings of the Third International Workshop on Reliability, Availability, and Security*. New York, NY, USA: ACM, 2010. (WRAS '10), p. 1:1–1:6. ISBN 978-1-4503-0642-3. <http://doi.acm.org/10.1145/1953563.1953564>.

HADZILACOS, V.; TOUEG, S. *A Modular Approach to Fault-Tolerant Broadcasts and Related Problems*. [S.l.], maio 1994. <http://www.cs.toronto.edu/ vassos/research/publications/HT94/paper.ps.gz>.

HANSEN, R.; CANNON, S. An efficient fault-tolerant tuple space. In: *Fault-Tolerant Parallel and Distributed Systems, 1994., Proceedings of IEEE Workshop on*. [s.n.], 1994. p. 220 –225. <http://dx.doi.org/10.1109/FTPDS.1994.494493>.

LAMPORT, L. The part-time parliament. *ACM Trans. Comput. Syst.*, ACM, New York, NY, USA, v. 16, n. 2, p. 133–169, maio 1998. ISSN 0734-2071. <http://doi.acm.org/10.1145/279227.279229>.

LAMPORT, L. Byzantizing paxos by refinement. In: *Proceedings of the 25th international conference on Distributed computing*. Berlin, Heidelberg: Springer-Verlag, 2011. (DISC'11), p. 211–224. ISBN 978-3-642-24099-7. <http://dl.acm.org/citation.cfm?id=2075029.2075058>.

LAMPORT, L.; MALKHI, D.; ZHOU, L. Reconfiguring a state machine. *SIGACT News*, ACM, New York, NY, USA, v. 41, n. 1, p. 63–73, mar. 2010. ISSN 0163-5700. <http://doi.acm.org/10.1145/1753171.1753191>.

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

LAMPSON, B. The abcd's of paxos. In: *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*. New York, NY, USA: ACM, 2001. (PODC '01), p. 13–. ISBN 1-58113-383-9. <http://doi.acm.org/10.1145/383962.383969>.

LAWDER, J.; KING, P. Using space-filling curves for multi-dimensional indexing. In: LINGS, B.; JEFFERY, K. (Ed.). *Advances in Databases*. Springer Berlin / Heidelberg, 2000, (Lecture Notes in Computer Science, v. 1832). p. 20–35. ISBN 978-3-540-67743-7. <http://dx.doi.org/10.1007/3-540-45033-5_3>.

LAWDER, J. K.; KING, P. J. H. Querying multi-dimensional data indexed using the hilbert space-filling curve. *SIGMOD Rec.*, ACM, New York, NY, USA, v. 30, n. 1, p. 19–24, mar. 2001. ISSN 0163-5808. <http://doi.acm.org/10.1145/373626.373678>.

LEE, J.; LEE, H.; KANG, S.; CHOE, S.; SONG, J. Ciss: An efficient object clustering framework for dht-based peer-to-peer applications. In: NG, W.; OOI, B.-C.; OUKSEL, A.; SARTORI, C. (Ed.). *Databases, Information Systems, and Peer-to-Peer Computing*. Springer Berlin Heidelberg, 2005, (Lecture Notes in Computer Science, v. 3367). p. 215–229. ISBN 978-3-540-25233-7. <http://dx.doi.org/10.1007/978-3-540-31838-5_15>.

LI, X.; WU, J. Searching techniques in peer-to-peer networks. In: ____. *Handbook of Theoretical and Algorithmic Aspects of Ad Hoc, Sensor, and Peer-to-Peer Networks*. Auerbach, 2006. cap. Chapter, p. 613 – 642. <http://www.kiv.zcu.cz/ ledvina/DHT/p2psurvey.pdf>.

LI, Z.; PARASHAR, M. Comet: a scalable coordination space for decentralized distributed environments. In: *Hot Topics in Peer-to-Peer*

Systems, 2005. HOT-P2P 2005. Second International Workshop on. [s.n.], 2005. p. 104 – 111. <http://dx.doi.org/10.1109/HOT-P2P.2005.7>.

LV, Q.; CAO, P.; COHEN, E.; LI, K.; SHENKER, S. Search and replication in unstructured peer-to-peer networks. In: *Proceedings of the 16th international conference on Supercomputing*. New York, NY, USA: ACM, 2002. (ICS '02), p. 84–95. ISBN 1-58113-483-5. <http://doi.acm.org/10.1145/514191.514206>.

LYNCH, N.; SHVARTSMAN, A. Rambo: A reconfigurable atomic memory service for dynamic networks. In: MALKHI, D. (Ed.). *Distributed Computing*. Springer Berlin / Heidelberg, 2002, (Lecture Notes in Computer Science, v. 2508). p. 173–190. ISBN 978-3-540-00073-0. <http://dx.doi.org/10.1007/3-540-36108-1_12>.

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

MALKHI, D.; REITER, M. Byzantine quorum systems. *Distributed Computing*, Springer Berlin / Heidelberg, v. 11, p. 203–213, 1998. ISSN 0178-2770. <http://dx.doi.org/10.1007/s004460050050>.

MARTIN, J.-P.; ALVISI, L. Fast byzantine consensus. *Dependable and Secure Computing, IEEE Transactions on*, v. 3, n. 3, p. 202 –215, july-sept. 2006. ISSN 1545-5971. <http://dx.doi.org/10.1109/TDSC.2006.35>.

MARTIN, J.-P.; ALVISI, L. Fast byzantine consensus. *Dependable and Secure Computing, IEEE Transactions on*, v. 3, n. 3, p. 202 –215, july-sept. 2006. ISSN 1545-5971. <http://dx.doi.org/10.1109/TDSC.2006.35>.

MOSTEFAOUI, A.; RAYNAL, M.; TRAVERS, C.; PATTERSON, S.; AGRAWAL, D.; ABBADI, A. From static distributed systems to dynamic systems. In: *Reliable Distributed Systems, 2005. SRDS 2005. 24th IEEE Symposium on*. IEEE, 2005. p. 109 – 118. <http://dx.doi.org/10.1109/RELDIS.2005.19>.

PICCO, G.; MURPHY, A.; ROMAN, G.-C. Lime: Linda meets mobility. In: *Software Engineering, 1999. Proceedings of the 1999 International Conference on*. [S.l.: s.n.], 1999. p. 368 –377. ISSN 0270-5257.

PIERGIOVANNI, S. T. *Concurrent Connectivity Maintenance with Infinitely Many Processes*. Tese (Doutorado) — MIDLAB - Università di Roma "La Sapienza", 2005. <https://www.dis.uniroma1.it/ midlab/articoli/STthesis.pdf>.

RODRIGUES, R.; LISKOV, B. *Rosebud: A scalable Byzantine-fault-tolerant storage architecture*. [S.l.], 2003. TR-2003-035. <http://hdl.handle.net/1721.1/30440>.

ROWSTRON, A.; DRUSCHEL, P. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: GUERRAOUI, R. (Ed.). *Middleware 2001*. Berlin: Springer, 2001, (Lecture Notes in Computer Science, v. 2218). p. 329–350. ISBN 978-3-540-42800-8. <http://dx.doi.org/10.1007/3-540-45518-3_18>.

SCHNEIDER, F. B. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 22, n. 4, p. 299–319, dez. 1990. ISSN 0360-0300. <http://doi.acm.org/10.1145/98163.98167>.

SCHULTZ, D. A.; LISKOV, B.; LISKOV, M. Mobile proactive secret sharing. In: *Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*. New York, NY, USA: ACM, 2008. (PODC '08), p. 458–458. ISBN 978-1-59593-989-0. <http://doi.acm.org/10.1145/1400751.1400856>.

SHEN, D.; SHAO, Y.; NIE, T.; KOU, Y.; WANG, Z.; YU, G. Hilbertchord: A p2p framework for service resources management. In: WU, S.; YANG, L.; XU, T. (Ed.). *Advances in Grid and Pervasive Computing*. Springer Berlin Heidelberg, 2008, (Lecture Notes in Computer Science, v. 5036). p. 331–342. ISBN 978-3-540-68081-9. <http://dx.doi.org/10.1007/978-3-540-68083-3_33>.

SIT, E.; MORRIS, R. Security considerations for peer-to-peer distributed hash tables. In: DRUSCHEL, P.; KAASHOEK, F.; ROWSTRON, A. (Ed.). *Peer-to-Peer Systems*. Springer Berlin / Heidelberg, 2002, (Lecture Notes in Computer Science, v. 2429). p. 261–269. ISBN 978-3-540-44179-3. <http://dx.doi.org/10.1007/3-540-45748-8_25>.

SONG, Y.; RENESSE, R. van. Bosco: One-step byzantine asynchronous consensus. In: TAUBENFELD, G. (Ed.). *Distributed Computing*. Springer Berlin / Heidelberg, 2008, (Lecture Notes in Computer Science, v. 5218). p. 438–450. ISBN 978-3-540-87778-3. <http://dx.doi.org/10.1007/978-3-540-87779-0_30>.

STEINMETZ, R.; WEHRLE, K. (Ed.). *Peer-to-Peer Systems and Applications*. Berlin: Springer, 2005. (Lecture Notes in Computer Science, v. 3485). ISSN 0302-9743. ISBN 978-3-540-29192-3.

STOICA, I.; MORRIS, R.; KARGER, D.; KAASHOEK, M. F.; BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.*, ACM, New York, NY, USA, v. 31, n. 4, p. 149–160, ago. 2001. ISSN 0146-4833. <http://doi.acm.org/10.1145/964723.383071>.

VU, Q. H.; LUPU, M.; OOI, B. C. *Peer-to-Peer Computing: Principles and Applications*. 1st. ed. [S.l.]: Springer, 2010. ISBN 3642035132, 9783642035135.

WALLACH, D. A survey of peer-to-peer security issues. In: OKADA, M.; PIERCE, B.; SCEDROV, A.; TOKUDA, H.; YONEZAWA, A. (Ed.). *Software Security - Theories and Systems*. Springer Berlin / Heidelberg, 2003, (Lecture Notes in Computer Science, v. 2609). p. 253–258. ISBN 978-3-540-00708-1. <http://dx.doi.org/10.1007/3-540-36532-X_4>.

XU, A.; LISKOV, B. A design for a fault-tolerant, distributed implementation of linda. In: *Fault-Tolerant Computing, 1989. FTCS-19. Digest of Papers., Nineteenth International Symposium on*. [s.n.], 1989. p. 199 –206. <http://dx.doi.org/10.1109/FTCS.1989.105566>.