LUIZ ANTONIO RODRIGUES

UMA SOLUÇÃO AUTONÔMICA PARA K-EXCLUSÃO MÚTUA EM SISTEMAS DISTRIBUÍDOS

Tese apresentada como requisito parcial à obtenção do grau de Doutor pelo Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.

Orientador: Prof. Dr. Elias P. Duarte Jr.

CURITIBA

2014

R696 Rodrigues, Luiz Antonio
Uma solução autonômica para K-exclusão mútua em sistemas distribuídos / Luiz Antonio Rodrigues. – Curitiba, 2014.
106f. : il. color. ; 30 cm.

Tese - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-graduação em Informática, 2014.

Orientador: Elias P. Duarte Jr.
Bibliografia: p. 93-101.

1. Sistemas Distribuídos. 2. Exclusão Mútua. 3. Difusão. 4. VCube. 5 Desempenho. I. Universidade Federal do Paraná. II. Duarte Jr., Elias P. III. Título.

CDD: 005.4476

Ministério da Educação
Universidade Federal do Paraná
Programa de Pós-Graduação em Informática

PARECER

Nós, abaixo assinados, membros da Banca Examinadora da defesa do aluno de Doutorado em Ciência da Computação, Luiz Antonio Rodrigues, avaliamos a tese de doutorado intitulada *"Uma Solução Autonômica para a k-Exclusão Mútua em Sistemas Distribuídos"*, cuja defesa pública foi realizada no dia 12 de agosto de 2014, às 13:30 horas, no Departamento de Informática do Setor de Ciências Exatas da Universidade Federal do Paraná. Após avaliação, decidimos pela:

(X) **aprovação** do candidato. ( ) **reprovação** do candidato.

Curitiba, 12 de agosto de 2014.

Prof. Dr. Elias Procópio Duarte Júnior
DINF/UFPR – Orientador

Profa. Dra. Luciana Bezerra Arantes
U. Pierre et Marie Curie(Paris VI)/França – Membro externo

Prof. Dr. Fernando Luís Dotti
PUC/RS – Membro Externo

Prof. Dr. Carlos Alberto Maziero
PPGINF/UTFPR – Membro Interno

Profa. Dra. Andrea Weber
DINF/UFPR – Membro Interno

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DO PARANÁ
Setor de Ciências Exatas
Coordenação de Pós-Graduação em Informática

AGRADECIMENTOS

A elaboração, desenvolvimento e conclusão de uma tese envolve tantos colaboradores diretos e indiretos que é arriscado tentar enumerá-los sem correr o risco de ser negligente. Assim, já começo me desculpando por possíveis omissões.

Agradeço imensamente ao meu professor e orientador Elias Duarte Jr. pelo inestimável trabalho de orientação e pelas inúmeras dicas, conselhos, apoio, publicações, entre muitas outras contribuições. Agradeço também a professora Luciana Arantes, especialmente pela oportunidade e orientação durante o estágio sanduíche no Lip6 - *Laboratoire d'Informatique de Paris 6*, Paris, França. *Merci beaucoup !!!*

Obrigado a minha família e amigos por compreender a minha ausência em tantos aniversários, festas de família, dia dos pais, da mães, especialmente durante o ano que passei no exterior. Obrigado a minha querida tia e amiga Rosa Maria pelo apoio desde a graduação, mestrado, universidade e doutorado. Agradeço também ao amigo Clodis e aos demais companheiros de aventuras, pelas discussões produtivas, mas principalmente por aquelas não tão produtivas: viagens e trilhas, indispensáveis para manter o corpo e a mente sãos.

Obrigado aos colegas do PPG-Inf, em especial aos companheiros do LaRSis, Guilherme, Ailton, Ricardo, Crist†ane, Rubens, Jéfer, Edmar, "Cowboy", Rogério, Edson, pelas discussões, debates, colaborações, brincadeiras, almoços, jantares, cafés.. Obrigado também aos inúmeros companheiros do LIP6 e *Maison du Brésil* na França. Muito obrigado pelas conversas, jantares, pique-niques, viagens, enfim, por todo o apoio recebido. *Merci mes amis! Vous me maquez beaucoup!.*

Um obrigado especial aos funcionários do PPG-Inf, especialmente Jucélia e Rafael, pelo apoio em todas as demandas burocráticas do processo, sem esquecer é claro, de todos os professores do DInf com os quais tive oportunidade de interagir nas aulas, nos corredores, nos eventos, nos cafés e, por que não, nos bares.

Meu muito obrigado às professoras Andrea Weber e Luciana Arantes, e professores Fernando Dotti e Carlos Maziero, membros da banca, pela aceitação, participação e inúmeras contribuições.

Agradeço ainda às agências de fomento brasileiras Fundação Araucária, Capes e CNPq, e ao INRIA/Lip6, França, pelo aporte financeiro nesta empreitada.

"Na hora ingrata de escrever,
como optar entre as variedades de insólito?
E que dizer, que não seja invalidado
pelo acontecimento de logo mais,
ou de agora mesmo?"

– Carlos Drummond de Andrade

RESUMO

Uma das grandes vantagens dos sistemas distribuídos é o compartilhamento de recursos. No entanto, diversos processos podem solicitar o acesso a um recurso compartilhado de forma concorrente e, em certos casos, é necessário garantir que um único processo obtenha permissão de acesso ao recurso em cada instante de tempo. Para tanto, são utilizados os algoritmos de exclusão mútua. Nas soluções que utilizam pedidos de permissão, cada processo deve solicitar aos demais a permissão para utilizar o recurso. A permissão deve ser obtida de todos os processos ou de um subconjunto deles, como definido pelas soluções com quóruns. Uma extensão do problema da exclusão mútua é a k-exclusão mútua. Nesta categoria, ao invés de um, existem k cópias idênticas do recurso compartilhado. O objetivo é garantir que, no máximo, k processos obtenham acesso aos recursos de cada vez. As soluções de k-exclusão mútua existentes são basicamente adaptações dos algoritmos de 1-exclusão mútua. Entretanto, a maior parte destas soluções não aborda a questão da ocorrência de falhas no sistema. Nesta tese é proposta uma solução autonômica de k-exclusão mútua distribuída que opera corretamente para até $n-1$ processos falhos, sendo n o total de processos no sistema. O algoritmo de k-exclusão mútua é baseado em algoritmos hierárquicos de difusão (*broadcast*), também propostos nesta tese. O objetivo de desenvolver estes algoritmos é otimizar a propagação das mensagens de requisição de recursos do algoritmo de exclusão mútua. Dois algoritmos de difusão foram propostos, um para difusão de melhor-esforço e outro para difusão confiável. Estes algoritmos são baseados em uma outra solução também proposta neste trabalho: um algoritmo autonômico e hierárquico para a construção e manutenção de árvores geradoras (*spanning trees*). As árvores são construídas de forma totalmente distribuída e adaptativa sobre uma topologia de hipercubo virtual, denominada VCube. A estratégia proposta é eficiente e escalável, além de tolerar até $n-1$ falhas de processo. As soluções propostas são também autonômicas no sentido que se adaptam automaticamente frente à ocorrência de falhas, reorganizando os elementos corretos do sistema. Uma segunda abordagem foi proposta para o problema da exclusão mútua, um algoritmo de quóruns, também construído sobre a topologia VCube. A carga e o tamanho dos quóruns são balanceados, mesmo após a ocorrência de falhas. Todos os algoritmos propostos são descritos, especificados e foram implementados através de simulação. São apresentadas provas de correção e resultados experimentais para todas as propostas.

Palavras-chave: Sistemas Distribuídos. Exclusão Mútua. Difusão. VCube. Desempenho.

ABSTRACT

One of the key purposes of distributed systems is to allow resources to be shared. However, several processes can request access to a shared resource concurrently and in some cases it is necessary to ensure that only a single process has permission to access the resource per instant of time. Mutual exclusion algorithms are used for this purpose. In permission-based solutions, each process must request permission to others before accessing the resource. The permission must be obtained for all processes or a subset of them, as is the case when quorum-based solutions are employed. An extension of the mutual exclusion problem is k-mutual exclusion. In this case, instead of one, there are k identical copies of the shared resource. The main issue is to ensure that at most k processes get access to resources at a time instant. Current k-mutual exclusion algorithms are basically adaptations of algorithms for mutual exclusion of a single resource. However, most of these solutions do not address the question of the occurrence of faults in the system. In this thesis an autonomic solution for distributed k-mutual exclusion is proposed that works correctly even if up to $n-1$ processes are faulty, assuming that the system consists of n processes. The k-mutual exclusion algorithm is based on hierarchical reliable broadcast algorithms also proposed in this thesis. The purpose for developing these algorithms is to optimize the propagation of request messages used by the mutual exclusion algorithm. Two hierarchical broadcast algorithms were proposed, one for best-effort broadcast and another for reliable broadcast. These broadcast algorithms are based on yet another building block that was proposed in this thesis: an autonomic hierarchical algorithm for building and maintaining spanning trees. The spanning trees are constructed in a fully distributed and adaptive way on a virtual hypercube-like topology, called VCube. The proposed approach is efficient and scalable, besides tolerating the fault of up to $n-1$ processes. The proposed solutions are also autonomic in the sense that they adapt themselves automatically after the occurrence of faults by reorganizing the correct processes remaining in the system. A second approach was also proposed for the mutual exclusion problem, a quorum-based algorithm, also built on the VCube topology. The load and size of the quorums are kept balanced, even after faults. All proposed algorithms are described, specified and have been implemented by simulation. Proofs of correctness and experimental results for all proposals are presented.

Keywords: Distributed Systems. Mutual Exclusion. Broadcast. VCube. Performance.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

CAPÍTULO 1

INTRODUÇÃO

Organizações e indivíduos dependem cada vez mais da computação distribuída para a realização das mais diversas tarefas. Exemplos disso são os sistemas para comércio eletrônico, governo eletrônico, educação a distância, entre várias outras aplicações empresariais e da Internet. Na sua definição clássica, um sistema distribuído consiste de um conjunto finito Π de $n > 1$ processos independentes $\{p_0, ..., p_{n-1}\}$ em execução sobre uma rede de computadores, como a Internet, que se comunicam usando troca de mensagens, colaborando para a realização de alguma tarefa (Coulouris et al., 2011; Raynal, 2013).

Uma das vantagens dos sistemas distribuídos é o compartilhamento de recursos (dispositivos, programas, dados, entre outros). No entanto, cada processo pode solicitar o acesso a um recurso compartilhado de forma arbitrária. Em alguns casos, este acesso precisa ser exclusivo. Assim, uma questão relevante é como organizar a concorrência pelos recursos garantindo duas propriedades principais: a segurança (*safety*), que garante que no máximo um solicitante obtenha o recurso de cada vez; e a propriedade de progressão (*liveness*), na qual todos os interessados em um recurso consigam obtê-lo em um tempo finito. A solução para este problema é chamada de *exclusão mútua* (Lamport, 1978; Ricart e Agrawala, 1981; Raynal e Beeson, 1986; Luo et al., 2013).

Existem duas abordagens clássicas para implementar a exclusão mútua em sistemas distribuídos. A primeira é por solicitação de permissão e a segunda é através de passagem de *token* (Raynal, 1991). Na solicitação de permissão cada processo que deseja fazer uso do recurso deve solicitar a todos os demais a permissão para utilizá-lo (Ricart e Agrawala, 1981; Sanders, 1987). Para o caso em que um único recurso é compartilhado, a solução trivial é enviar uma mensagem de solicitação a cada um dos outros $n - 1$ processos do sistema e aguardar as respostas (Bertsekas et al., 1991). Com passagem de *token*, somente o processo que detém o *token* pode acessar o recurso compartilhado (Le Lann, 1977; Suzuki e Kasami, 1985; Raymond, 1989b; Naimi et al., 1996; Bertier et al., 2004). O *token* pode circular entre os processos seguindo uma organização lógica em anel, por exemplo.

Uma variação da exclusão mútua é a k-exclusão mútua, na qual k recursos são compartilhados entre os n processos (Raymond, 1989a; Bulgannawar e Vaidya, 1995; Bouillaguet et al., 2008). Os algoritmos desta categoria também são baseados em pedido de permissão e passagem de *token*. Nas soluções com pedido de permissão, o processo solicitante precisa aguardar, no mínimo, $n - k$ permissões. Com a utilização de *tokens*, apenas os processos que possuem um dos k *tokens* podem fazer uso dos recursos.

Um fator relevante que tem impacto direto na escalabilidade de um algoritmo de ex-

clusão mútua distribuído é o mecanismo de disseminação de mensagens. Uma abordagem simples, empregada na maioria das propostas, é utilizar *broadcast* para enviar as mensagens de requisição. Em redes nas quais este mecanismo não está disponível, como a Internet, uma forma de emulá-lo é enviar uma mensagem ponto-a-ponto para cada outro processo. No entanto, essa solução pode sobrecarregar o emissor quando muitos processos estão envolvidos. Uma forma de contornar este problema é utilizar soluções hierárquicas, como as árvores (Avresky, 1999). Em função das propriedades logarítmicas, a utilização de árvores distribui a carga de envio de mensagens entre todos os processos do sistema.

Um segundo problema que merece atenção em uma solução distribuída de exclusão mútua é a possibilidade de ocorrência de falhas. No modelo com pedido de permissão, por exemplo, o solicitante precisa ter informações sobre o estado dos processos para não ficar aguardando indefinidamente por respostas daqueles falhos (Bouillaguet et al., 2008). O mesmo acontece quando se usa passagem de permissão. No caso de falha do processo que possui o *token*, o sistema precisa identificar o problema e gerar um novo *token*. Considerando a quantidade de elementos envolvidos e a complexidade cada vez mais crescente dos sistemas distribuídos, é imprescindível que as soluções propostas sejam capazes de se recuperar de falhas de modo automático, adaptando-se frente a condições adversas (Kephart e Chess, 2003).

Neste contexto, foi proposta uma solução autonômica tolerante a falhas de k-exclusão mútua distribuída baseada no modelo com pedidos de permissão. O algoritmo proposto possibilita a obtenção de recursos de forma eficiente, mesmo na presença de até $n - 1$ processos falhos. Um mecanismo auxiliar de monitoramento, chamado VCube (Ruoso, 2013), é utilizado para detectar as falhas e adaptar o sistema. No VCube, os processos do sistema são organizados em uma topologia virtual em hipercubo, apresentando diversas propriedades logarítmicas. Uma segunda abordagem para solucionar o problema da exclusão mútua utilizando sistemas de quóruns está em desenvolvimento e um algoritmo para construção de quóruns com base no VCube é apresentado. Resultados de simulação demonstram a estabilidade no tamanho e carga dos quóruns em cenários com e sem falhas.

Para propagar as mensagens de requisição de recursos do algoritmo de exclusão mútua, dois mecanismos hierárquicos de difusão de mensagens (*broadcast*) foram criados: um para difusão de melhor-esforço e outro para difusão confiável. Os dois mecanismos são baseados em um algoritmo distribuído de árvore geradora mínima, também proposto neste trabalho. O algoritmo de árvore faz uso da topologia mantida pelo VCube para construir, a partir de um nodo fonte qualquer (raiz), uma árvore geradora mínima que contém todos os nodos considerados corretos pelo VCube. O algoritmo é totalmente distribuído e autonômico no sentido que a árvore é construída e mantida automaticamente após a ocorrência de falhas, sem acarretar, entretanto, custos adicionais com troca de mensagens e/ou paralisação do sistema. Além das provas, testes de simulação confirmam a eficiência dos algoritmos comparados a outras alternativas.

O restante do texto está organizado nos seguintes capítulos. O Capítulo 2 apresenta alguns dos principais modelos de sistema distribuído presentes na literatura, destacando suas principais características em termos dos atributos de sincronismo e modelo de falhas. O Capítulo 3 apresenta o problema da exclusão mútua e os principais algoritmos baseados em pedidos de permissão e passagem de *token*. No Capítulo 4 é apresentada a solução de árvore geradora mínima distribuída e autonômica, que é a base para todas as demais soluções propostas neste trabalho. O Capítulo 5 apresenta duas soluções de *broadcast* construídas utilizando a árvore geradora proposta. O Capítulo 6 apresenta a solução de k-exclusão mútua que faz uso da árvore geradora e da solução de *broadcast* propostas no trabalho. Por fim, o Capítulo 7 contém a conclusão, um resumo das principais contribuições e uma lista de trabalhos futuros.

CAPÍTULO 2

MODELOS DE SISTEMAS DISTRIBUÍDOS

Um *modelo* pode ser definido como uma coleção de atributos e um conjunto de regras que define como estes atributos interagem (Mullender, 1993). A partir do modelo, é possível estudar o comportamento do sistema, seja por meio de análise teórica ou de simulação. Um bom modelo é completo o bastante para representar fielmente o comportamento do sistema da forma mais simples possível, isto é, com o menor número de parâmetros necessários. Neste capítulo, modelos de sistemas distribuídos são analisados sob o ponto de vista de características referentes à topologia do sistema, ao sincronismo, às falhas que podem ocorrer e ao dinamismo na composição do sistema.

2.1 Definições Básicas

Um sistema distribuído consiste de um conjunto finito Π de $n > 1$ processos independentes $\{p_1, ..., p_n\}$ que se comunicam usando troca de mensagens, colaborando para a realização de alguma tarefa (Raynal, 2013). Sem perda de generalidade, considera-se que cada processo é executado em um processador (nodo) individual. A execução de um processo é caracterizada pela execução sequencial de ações, que abrangem eventos internos e eventos de envio/recebimento de mensagens. As ações são executadas de forma atômica e em tempo finito.

A comunicação entre processos é realizada, em geral, de duas formas: ponto-a-ponto e *broadcast* (Hadzilacos e Toueg, 1994). Existem duas primitivas associadas aos enlaces: `send` utilizada para o envio de mensagens e `receive`, utilizada para o recebimento de mensagens. Em redes ponto-a-ponto, cada enlace conecta um par de processos e cada processo pode enviar uma mensagem para um único destinatário de cada vez. Em redes *broadcast*, um canal compartilhado conecta todos os processos. Com isso, cada processo pode enviar uma mensagem simultaneamente a todos os outros processos.

Em um sistema baseado em troca de mensagens existe um atraso entre o envio da mensagem por um processo e o recebimento dela no destinatário. Neste trajeto, a mensagem normalmente é armazenada em *buffers* de entrada e saída, que podem ser finitos ou infinitos (Lamport e Lynch, 1990). Se o *buffer* é finito, um conjunto limitado de mensagens pode ser enviado através do enlace até que o *buffer* esteja cheio. Se o *buffer* tem capacidade infinita, pode existir uma quantidade arbitrária de mensagens em trânsito e um processo sempre pode enviar uma nova mensagem.

Quando o *buffer* do enlace permite que mais uma mensagem seja enviada, é possível que mensagens sejam recebidas fora de ordem, isto é, em uma ordem diferente da que

foram enviadas (Birman e Joseph, 1987; Hadzilacos e Toueg, 1994). Um modelo que utiliza canais de comunicação FIFO (*First In, First Out*) garante que as mensagens enviadas por um emissor serão entregues no receptor na ordem em que foram enviadas. Isto é, se um processo envia uma mensagem m_1 antes de uma mensagem m_2, o processo destinatário deverá receber m_1 antes de receber m_2.

Se considerado o sistema distribuído como um todo, a ordenação FIFO garante que as mensagens enviadas por um mesmo emissor serão entregues a todos os destinatários na ordem em que foram enviadas. Quando mais de um emissor está envolvido e as mensagens enviadas por eles possuem uma relação de dependência, a ordenação causal pode ser utilizada. A ordenação causal baseia-se na relação de precedência ($\rightarrow$) de Lamport (Lamport, 1978), que pode ser lida informalmente como "aconteceu antes de". Um sistema neste modelo tem a seguinte propriedade (Kshemkalyani e Singhal, 2008): para quaisquer duas mensagens m_{ij} enviada pelo processo p_i para o processo p_j e m_{kj} enviada de p_k para p_j, se $send(m_{ij}) \rightarrow send(m_{kj})$, então $receive(m_{ij}) \rightarrow receive(m_{kj})$.

Quanto não existe relação de precedência, mensagens podem ser entregues em ordens diferentes em cada processo. Para prevenir este comportamento, a ordenação total pode ser empregada. Na ordenação total, todos os processos corretos recebem todas as mensagens na mesma ordem. Formalmente, se dois processos corretos p_i e p_j recebem duas mensagens m_1 e m_2, p_i pode receber m_1 antes de m_2 se e, somente se, p_j receber m_1 antes de m_2.

Na prática, os sistemas reais podem implementar a ordenação FIFO das mensagens utilizando números de sequência e as ordenações causal e total por meio de *timestamps*. Além disso, algoritmos distribuídos específicos devem estar implementados para garantir os diferentes níveis de ordenação. Com isso, mesmo que no nível do roteamento elas possam ser encaminhadas por enlaces diferentes e alcancem o destino em uma ordem arbitrária, a ordenação é garantida para os processos finais.

2.2 Topologia

A topologia descreve como os processos de um sistema distribuído podem enviar mensagens entre si (Lamport e Lynch, 1990). Uma topologia é descrita por um grafo $G = (V, E)$ no qual os vértices $V(G) = \Pi$ são os nodos e as arestas $E(G)$ são os enlaces entre os nodos. Um enlace corresponde à capacidade de dois nodos se comunicarem sem intermediários, podendo ser um enlace físico ou lógico. Sendo assim, uma aresta $(i, j) \in E(G)$ indica que o processo i pode se comunicar diretamente com o processo j. O grafo pode ser direcionado ou não-direcionado. Em um grafo não-direcionado, a existência de uma aresta entre dois processos indica que ambos podem se comunicar diretamente. Em alguns casos, a topologia é inicialmente conhecida, mas existem situações nas quais o conhecimento é limitado e, no mínimo, cada nodo conhece o conjunto de processos vizinhos. O sistema pode ser

totalmente conectado (*fully connected*), isto é, todos os processos podem se comunicar diretamente entre si, configurando um grafo não-direcionado completo. Os sistemas que não são totalmente conectados são ditos de topologia arbitrária.

2.3 Sincronismo

Um modelo de sistema distribuído considera, em geral, dois atributos temporais: a velocidade de execução dos processos Δ e o atraso de mensagens nos canais de comunicação Φ. De acordo com estes limites, os sistemas podem ser classificados em *síncronos* ou *assíncronos* (Cristian, 1991). Em um sistema síncrono, há limites conhecidos para estes atributos. Nos sistemas assíncronos estes limites não existem. Entretanto, a maioria dos sistemas reais não se encaixa nestes modelos. Uma sobrecarga momentânea na rede, por exemplo, pode fazer como que o tempo de transmissão das mensagens passe a ser maior que o esperado. Com o objetivo de definir modelos que melhor reflitam os sistemas reais, foram definidos os sistemas *parcialmente síncronos*. Esta seção apresenta os modelos temporais para sistemas distribuídos com foco nas definições e características principais de cada um deles.

2.3.1 Sistemas Síncronos

Em um sistema síncrono existem limites conhecidos para os tempos de processamento, envio de mensagens, taxa de variação de relógio (*drift*) e diferenças entre relógios locais (Veríssimo e Rodrigues, 2001). Hadzilacos e Toueg (1994) definem três propriedades que caracterizam esses sistemas:

1. Existe um limite superior de tempo utilizado por um processador para executar um passo de execução, isto é, dada uma constante $\tau \geq 0$, todo processo p executa uma instrução em, no máximo, τ unidades de tempo após iniciá-la. Desta forma, pode-se definir um limite Δ no qual um processador pode executar mais rápido que os demais (Turek e Shasha, 1992).

2. Cada processo p possui um relógio local C_p com limite da taxa de variação conhecida $\rho \geq 0$ tal que, para todo p e todo tempo $t > t'$,

$$\frac{1}{1+\rho} \leq \frac{C_p(t) - C_p(t')}{(t - t')} \leq (1+\rho) \tag{2.1}$$

3. Existe um limite conhecido $\Phi \geq 0$ para o atraso na entrega de mensagens, constituído pelo tempo de envio, transporte e recebimento das mensagens pela rede;

Fetzer e Cristian (1995) acrescentam que os relógios entre dois processos p e q quaisquer estão sincronizados se existe um parâmetro de erro máximo ε para todo tempo t tal que:

$$| C_p(t) - C_q(t) | \leq \varepsilon \tag{2.2}$$

Ao invés de caracterizar os limites temporais em termos de tempo real, é possível definilos em quantidade de passos executados pelos processos (Freiling et al., 2011). Neste caso, Δ e Φ têm os seguintes significados:

- Velocidade de processamento: para cada Δ passos realizados por um processo, todos os outros processos executam ao menos um passo;
- Atraso das mensagens: se uma mensagem é enviada no passo k, ela deve ser entregue em no máximo $k + \Phi$ passos do processo emissor.

Assim, o limite ω, que corresponde ao tempo de ida e volta de uma mensagem trocada entre dois processos p e q, pode ser definido como:

$$\omega = \Phi + s \cdot \Delta + \Delta \cdot \Phi \tag{2.3}$$

Inicialmente, são necessários Φ passos entre o envio da mensagem pelo processo p e a recepção pelo processo q. Em seguida, considerando que um processo utiliza s passos para receber a mensagem e enviar uma resposta, serão necessários $s \cdot \Delta$ passos para receber, processar e responder cada mensagem. Por fim, após Δ passos de processamento em q, a mensagem de resposta alcança p. Considerando que q pode ser muito mais lento que p e Δ é medido em q, a multiplicação de Δ por Φ define um limite superior para q executar Δ passos.

O fato de existirem limites na velocidade relativa dos processos permite que qualquer processo defina uma quantidade máxima de passos que um outro processo executou no sistema. Estas propriedades são equivalentes à existência de um relógio global, o que caracteriza estes sistemas como síncronos. Além disso, é possível e seguro fazer uso de *timeouts* para detectar falhas de processos ou de comunicação. Estas características permitem que os sistemas síncronos sejam utilizados, inclusive, para aplicações de tempo real (Freiling et al., 2011). É importante ressaltar que o fato dos relógios avançarem de forma sincronizada e constante não significa que os processos avançam igualmente na execução local de algoritmos.

A maior limitação em assumir um sistema síncrono é a dificuldade de garantir os limites temporais. Tais garantias normalmente exigem um controle rigoroso em diversos aspectos do sistema, inclusive na sua carga de processamento, o que demanda usualmente *hardware* e *software* apropriados. Em redes locais estas condições podem até ser satisfeitas, mas é muito difícil (para não dizer impossível) manter garantias tão rígidas em redes de larga

escala, como a Internet. Ao se definir limites muito pequenos, corre-se o risco de violá-los. Por outro lado, limites muito grandes impactam no desempenho do sistema. Por estas razões, é muito difícil e custoso implementar um sistema distribuído síncrono.

2.3.2 Sistemas Assíncronos

Sistemas distribuídos assíncronos são aqueles livres de restrições temporais, também definidos como *time-free* (Cristian e Fetzer, 1999; Veríssimo e Rodrigues, 2001). Neste modelo, os limites temporais (como velocidade de processamento e atraso de entrega das mensagens) não são conhecidos ou não existem, o que impossibilita a definição de qualquer asserção neste sentido. Assim, não é possível garantir que uma execução termine no tempo esperado (Krakowiak e Shrivastava, 1999).

Além disso, considera-se que os processos não têm acesso a qualquer tipo de relógio global, embora uma ordenação virtual dos eventos no sistema possa ser definida com o auxílio de relógios lógicos (Lamport, 1978). A passagem de tempo em um sistema assíncrono pode ser medida da seguinte maneira (Guerraoui e Rodrigues, 2006):

- Cada processo p mantém um contador inteiro local l_p, inicializado em 0;
- Para cada instrução executada por p, l_p é incrementado em uma unidade;
- Quando p envia uma mensagem, registra nela o valor do seu contador l_p. Esta marcação do evento e, denominada (*timestamp*), é denotada por $t(e)$;
- Quando um processo q recebe uma mensagem de p com *timestamp* l_p, incrementa seu contador $l_q = max(l_q, l_p) + 1$.

Desta forma, dados dois eventos quaisquer e_1 e e_2, pode se dizer que $e_1 \rightarrow e_2$ (e_1 precede e_2) se: (a) e_1 e e_2 ocorreram no mesmo processo e e_1 aconteceu antes de e_2; (b) e_1 é um envio de mensagem do processo p para o processo q e e_2 é a recepção desta mensagem; ou (c) existe um evento e' tal que $e_1 \rightarrow e'$ e $e' \rightarrow e_2$. Considerando-se o *timestamp*, $e_1 \rightarrow e_2 \Rightarrow t(e_1) < t(e_2)$.

Como exemplo, considere o cenário com três processos p_A, p_B e p_C da Tabela 2.1. Os eventos são representados por instruções locais (*instr*), envio de mensagens (*send*) e recebimento (*recv*). Inicialmente, como o contator incia em 0, cada processo atribui o valor 1 ao primeiro evento. O processo p_C também inicia o contador em 0, mas o atualiza já no primeiro evento de recebimento. Note que todo evento de recebimento exige este tratamento diferenciado no incremento do contador.

Feita a ordenação parcial, é possível que mais de um processo possua eventos com o mesmo *timestamp*. Uma solução para obter a ordenação total é utilizar o identificador do processo como referência. No exemplo da Tabela 2.1, considerando a ordem dos identificadores como p_A-p_B-p_C, a ordenação total é: $instr\ A_1 \rightarrow instr\ B_1 \rightarrow send(p_B) \rightarrow$

$instr\ B_2 \rightarrow recv(p_A) \rightarrow instr\ B_3 \rightarrow send(p_C) \rightarrow recv(p_B) \rightarrow instr\ C1 \rightarrow send(p_A) \rightarrow recv(p_C) \rightarrow instr\ C_2$.

Tabela 2.1: Exemplo de ordenação de eventos com relógios lógicos.

p_A	l_A	p_B	l_B	p_C	l_C
$instr\ A_1$	(1)	$instr\ B_1$	(1)	$recv(p_B)$	$max(0,5)+1$=(6)
$send(p_B)$	(2)	$instr\ B_2$	(2)	$instr\ C_1$	(7)
$recv(p_C)$	$max(2,8)+1$=(9)	$recv(p_A)$	$max(2,2)+1$=(3)	$send(p_A)$	(8)
		$instr\ B_3$	(4)	$instr\ C_2$	(9)
		$send(p_C)$	(5)		

Embora o uso de relógios lógicos seja útil para algumas aplicações, a ausência de limites temporais impossibilita a implementação de soluções determinísticas para problemas de acordo em sistemas assíncronos sujeitos a sequer uma falha. A grande dificuldade está na determinação do estado dos processos distribuídos, dado que não é possível distinguir entre um processo falho e um muito lento (Fischer, Lynch e Paterson, 1985). Dada esta impossibilidade, várias aplicações ficam inviabilizadas, como o gerenciamento de grupos (*group membership*) e a ordenação total de mensagens (*atomic broadcast*), entre outras.

2.3.3 Sistemas Parcialmente Síncronos

Um dos problemas mais importantes em sistemas distribuídos é o consenso (Pease et al., 1980; Dixit et al., 2009; Correia et al., 2011). No consenso, todos os processos corretos deve entrar em acordo sobre um determinado valor, mesmo na presença de falhas. Em sistemas síncronos, este problema tem solução trivial, devido à possibilidade de utilização de *timeouts*. Em sistemas assíncronos, Fischer, Lynch e Paterson (1985) mostraram que não existe solução para o consenso devido à impossibilidade de diferenciar um processo muito lento de um processo falho. Esta prova ficou conhecida como *impossibilidade FLP*, sigla derivada das iniciais dos sobrenomes dos autores.

Dada a dificuldade de se construir um sistema síncrono e a impossibilidade de consenso em sistemas assíncronos com possibilidade de falhas, passaram a ser propostas estratégias que consideram um modelo de sistema entre o síncrono e o assíncrono, chamados de modelos *parcialmente síncronos*. Dwork et al. (1988) consideram dois tipos de sistemas parcialmente síncronos. No primeiro, os limites dos atributos temporais existem, mas são desconhecidos. No segundo modelo, os limites são conhecidos, mas somente após um tempo de estabilização *GST* (*Global Stabilization Time*), que tem duração finita, mas desconhecida. Nestes casos, a solução para o consenso existe se a maioria dos processos é correta. Já em Chandra e Toueg (1996) é considerado um modelo parcialmente síncrono que engloba os dois anteriores, no qual os atributos temporais têm limites, mas os limites são desconhecidos e só valem após um tempo de estabilização também desconhecido.

O fato de existir o tempo GST não significa que os limites de velocidade de processamento Δ e atraso de mensagens Φ serão garantidos eternamente. Em uma situação real, essas garantias normalmente são válidas por um intervalo de tempo limitado L. Considere que GST se refere ao instante em que inicia o intervalo de estabilização. A grande questão é garantir que este intervalo L seja grande o suficiente para resolver o problema, ou seja, um intervalo definido entre GST e $GST + L$ é suficiente para tornar as soluções factíveis. Segundo Cristian e Fetzer (1999), em uma rede local os períodos de estabilidade são muito maiores que os de instabilidade. Prova disso são as soluções baseadas em *timeout*, que funcionam perfeitamente em boa parte do tempo. Com isso, períodos estáveis na ordem de minutos ou horas são mais que suficientes para finalizar, por exemplo, uma transação em um banco de dados distribuído (Freiling et al., 2011).

Além da definição do modelo parcialmente síncrono, o trabalho de Dwork et al. (1988) define uma abstração de sistema para permitir a solução de problemas de acordo. Nesta abstração, o processamento é dividido em rodadas (*rounds*) sincronizadas. Cada rodada r possui três etapas: envio, recepção e transição. No envio, cada processo envia uma mensagem para todos os outros. Na recepção, um conjunto de mensagens que haviam sido enviadas é recebido por todos os processos. Este conjunto não é necessariamente igual ao conjunto enviado, já que mensagens podem ser perdidas. Na terceira etapa, o processo atualiza o seu estado baseado no conjunto de mensagens que recebeu.

De acordo com a definição do intervalo de estabilização, nenhuma mensagem pode ser perdida enquanto o sistema estiver estável. Tendo em vista a organização do sistema em rodadas, pode-se definir que existe um GSR (*Global Stabilization Round*) tal que, nenhuma mensagem é perdida para toda rodada $r \geq GSR$. Seja C o conjunto de processos corretos, σ_p^r o conjunto de mensagens enviadas pelo processo p na rodada r e $\mu_p^r[q]$ o conjunto das mensagens recebidas pelo processo p do processo q na rodada r, esta definição pode ser expressa como (Pedone e Schiper, 2010):

$$\forall r \geq GSR : \forall p, q \in C : \mu_p^r[q] = \sigma_q^r \tag{2.4}$$

Em outras palavras, para toda rodada r a partir do GSR e considerando todos os pares de processos p, q corretos, o conjunto de mensagens recebidas por p de q é igual ao conjunto de mensagens enviadas por q, ou seja, nenhuma mensagem foi perdida.

Além desta definição clássica, dois outros modelos foram propostos para definir os sistemas parcialmente síncronos. No modelo *timed asynchronous*, Cristian e Fetzer (1999) consideram que o sistema não é *sempre* assíncrono, variando entre períodos de estabilidade e instabilidade (Guerraoui e Schiper, 1997). No modelo *quasi-synchronous*, proposto por Veríssimo e Almeida (1995), o sistema não é assíncrono em sua totalidade, isto é, existem limites para algumas propriedades, por exemplo, o relógio pode ser sincronizado ou o atraso das mensagens possui um limite conhecido.

O modelo *timed asynchronous* difere do modelo assíncrono pelo fato de que os processos têm acesso a relógios locais sincronizados de tal forma que, considerando uma constante de variação de relógio ρ, um relógio local possui um valor entre $-\rho$ e $+\rho$ em relação ao relógio correto. Além disso, a comunicação é feita por um serviço de datagrama não confiável, que permite perdas e atrasos de mensagens. Processos podem falhar por colapso (*crash*) e desempenho, podendo se recuperar. De acordo com seus autores, estas condições aproximam o modelo das redes reais e permitem que sistemas não síncronos possam implementar soluções como o consenso.

Já o modelo *quasi-synchronous* considera que os atributos temporais (velocidade dos processos, atraso de mensagens, taxa de variação de relógio, entre outros) existem, mas alguns ou todos eles estão tão distantes do comportamento "normal" que outros valores mais próximos dos limites estabelecidos devem ser utilizados para viabilizar a solução. Neste caso, surge a necessidade de determinar quando um atributo pode ser utilizado e quando deve ser descartado. Para tanto, Almeida e Veríssimo (1996) definem o conceito de detector de falhas de temporização (*timing failure detector*). Semelhante aos detectores de falhas tradicionais, sua funcionalidade é detectar quando uma asserção temporal foi violada.

Além destes modelos, Veríssimo (2006) sugere uma outra forma de contornar o problema da impossibilidade FLP por meio de *wormholes*. O autor faz uma analogia com a teoria sobre atalhos no espaço, que permitiriam viagens através de dimensões, permitindo alcançar lugares no universo mais rapidamente que o limite imposto pela velocidade da luz. A abordagem considera a sincronia do sistema nas dimensões temporal e espacial (Bona et al., 2012). Na linha do tempo, os componentes do sistema podem operar mais rápido ou mais devagar, modificando os limites temporais e alternando entre períodos sincronizados e não-sincronizados. Além disso, alguns componentes do sistema podem apresentar um maior grau de sincronismo entre si, estabelecendo um sincronismo espacial. A solução propõe o uso de um sistema híbrido, combinando subsistemas assíncronos e não-assíncronos. O *wormhole* seria um subsistema não-assíncrono (síncrono ou parcialmente síncrono) utilizado para alcançar a sincronia exigida pela aplicação.

O modelo Θ, proposto por Widder e Schmid (2009), considera que uma troca de mensagem consiste de: preparação local no emissor, transmissão através do enlace e computação local no receptor. Sendo assim, o atraso de transmissão δ_{pq} entre dois processos corretos p e q consiste de duas partes: uma parte fixa $d_{pq} > 0$ determinada pela velocidade dos processadores e pelas características de transmissão do enlace (distância, velocidade, etc.); e uma parte variável $\omega_{pq} \geq 0$, constituída principalmente pelos atrasos nas filas, compartilhamento de recursos, carga do sistema e ocorrência de falhas. O modelo define um limite superior τ^+ e um limite inferior τ^- para o atraso entre dois processos corretos, tal que a taxa $\Theta = \tau^+/\tau^-$ respeite um limite superior.

2.4 Modelo de Falhas

Em um modelo de sistema baseado em troca de mensagens existem dois tipos de falhas: falhas de comunicação e falhas de processo (Lamport e Lynch, 1990). Falhas de comunicação geralmente resultam em perda de mensagens, duplicação ou corrupção de dados. Uma falha de processo ocorre quando o comportamento do algoritmo em execução foge da especificação inicial.

Durante a comunicação, mensagens podem ser perdidas por falhas de *omissão* e *temporização/desempenho* (Mullender, 1993; Jalote, 1994; Birman, 1996). Falhas de omissão são caracterizadas pelo não envio e/ou não recebimento de mensagens que deveriam ter sido enviadas e/ou recebidas por algum processo. Podem ser causadas por falta de espaço nos *buffers* do sistema operacional e da interface de rede ou por erros de transmissão detectados no receptor. Falhas de temporização ocorrem quando uma propriedade temporal do sistema é violada, como uma superação do limite de variação do relógio ou quando uma mensagem é entregue com um atraso maior que o tolerado pelos processos. Nos modelos que consideram relógios, uma mensagem atrasada pode ser considerada como perdida, já que este tipo de mensagem é normalmente descartada. Dentro das falhas de enlace pode-se ainda considerar a possibilidade de falhas de *particionamento*, que impossibilitam a comunicação entre determinados pares de nodos. Este é o tipo de falha de comunicação mais difícil de tratar, pois as partições podem evoluir no processamento independentemente umas das outras.

Além das falhas de comunicação, processos podem falhar por *colapso* (*crash*). Durante o colapso, o processo não executa qualquer ação e não responde aos estímulos externos, isto é, não executa processamento, nem envia ou recebe mensagens. O modelo de falhas de colapso pode considerar a possibilidade de recuperação do processo. Se a recuperação é possível, o sistema é dito *crash-recovery*. Neste caso, os processos possuem dois tipos de armazenamento: estável e volátil. Durante a recuperação, apenas os dados armazenados em memória estável, como discos, são recuperados.

O modelo de falhas mais abrangente é aquele que considera falhas arbitrárias, também conhecidas como bizantinas, termo inspirado no problema dos generais bizantinos (Lamport et al., 1982). Falhas arbitrárias incluem, além das falhas de comunicação e processo descritas anteriormente, aquelas geradas devido a comportamento malicioso.

Em uma definição hierárquica, as falhas de colapso são especificações das falhas de omissão. Estas por sua vez, são englobadas pelas falhas de temporização e as falhas arbitrárias compreendem todas as anteriores (Jalote, 1994). Isto significa que um sistema projetado para assumir falhas arbitrárias é capaz de tolerar qualquer tipo de falha. Entretanto, devido ao alto custo de projetar sistemas com este requisito, a maior parte das aplicações utiliza modelos que contemplam basicamente as falhas de colapso, visto que falhas de omissão podem ser superadas com o uso de canais confiáveis, bem como com o

auxílio de eventos de notificação do sistema operacional e as falhas de temporização são abstraídas do modelo de sincronismo adotado.

2.5 Sistemas Distribuídos Dinâmicos

Ao contrário de um sistema estático, em um sistema distribuído dinâmico não existe topologia fixa e nem conhecimento prévio sobre a quantidade de elementos na rede, que variam de acordo com a entrada e saída de nodos (Mostefaoui et al., 2005; Augustine et al., 2012). Este modelo é especialmente interessante porque representa uma gama diferenciada de aplicações, como aquelas baseadas em P2P (*peer-to-peer*), MANETS (*Mobile Ad-hoc Networks*), VANETS (*Vehicular Ad-hoc Networks*), redes de sensores e, mais recentemente, computação em nuvem.

A dinamicidade das redes implica em uma modificação significativa das características dos sistemas distribuídos. A entrada e saída de nodos gera constantes mudanças na topologia da rede. A mobilidade, por exemplo, pode definir um grafo de comunicação não completo, podendo inclusive criar o particionamento temporário da rede. Nestes cenários, cada nodo tem um conhecimento parcial da topologia e do conjunto de nodos que compõem o sistema. Formalmente, dado um conjunto finito Π de $n > 1$ nodos no sistema, cada nodo i possui uma visão parcial $\Pi_i \subseteq \Pi$, mas desconhece Π e n.

Assim, o desenvolvimento de soluções distribuídas para sistemas dinâmicos é um desafio, pois a volatilidade dos elementos prejudica a percepção global do sistema (conjunto total de nodos, número máximo de falhas, conectividade e comunicação confiável, por exemplo) (Walter et al., 2001; Lin et al., 2004; Tamhane e Kumar, 2010; Greve et al., 2011b). Em geral, as soluções para sistemas dinâmicos utilizam algumas asserções, como a estabilização, ou seja, em algum momento as mudanças param de acontecer ou passam a acontecer com menos frequência, possibilitando que o sistema consiga se organizar entre um evento e outro (Kuhn et al., 2010). Por este motivo, este tipo de sistema é também conhecido como auto-organizável.

O próprio modelo de grafos para redes estáticas não consegue representar adequadamente a topologia de uma rede dinâmica. Por esta razão, novos conceitos de grafos dinâmicos têm sido propostos e adotados neste contexto (Casteigts et al., 2010, 2011; Greve et al., 2011a).

CAPÍTULO 3

EXCLUSÃO MÚTUA EM SISTEMAS DISTRIBUÍDOS

O compartilhamento de recursos em sistemas distribuídos frequentemente demanda soluções de controle de acesso aos recursos compartilhados. Uma questão relevante é como organizar o acesso concorrente garantindo duas propriedades principais: a segurança (*safety*), que garante que somente um solicitante obtenha o recurso de cada vez e a progressão (*liveness*), na qual todos os interessados em um recurso consigam obtê-lo em um tempo finito. A solução para este problema é chamada de *exclusão mútua* (Lamport, 1978; Sopena et al., 2006; Romano e Rodrigues, 2009).

Este capítulo apresenta o problema da exclusão mútua e as principais soluções relevantes existentes, incluindo a k-exclusão mútua, utilizada quando mais de uma cópia do recurso está disponível no sistema. O capítulo apresenta também as métricas comumente utilizadas para medir o desempenho dos algoritmos, que são discutidos neste contexto.

3.1 Definição do Problema

A exclusão mútua soluciona um problema fundamental em sistemas distribuídos que é garantir a integridade dos recursos compartilhados. Na sua definição básica, a exclusão mútua garante que um único processo tenha permissão de acesso a um recurso em cada instante de tempo (Lamport, 1978; Ricart e Agrawala, 1981; Raynal e Beeson, 1986; Kshemkalyani e Singhal, 2008).

Existem basicamente duas abordagens clássicas para implementar a exclusão mútua em sistemas distribuídos: pedido de permissão (*permission-based*) e passagem de *token* (*token-based*) (Raynal, 1991). Nas soluções que utilizam pedido de permissão, cada processo que deseja fazer uso do recurso deve solicitar a todos os demais a permissão para utilizá-lo (Ricart e Agrawala, 1981; Sanders, 1987). Para o caso em que um único recurso é compartilhado, a solução trivial é enviar uma mensagem de solicitação a cada um dos outros $n-1$ processos do sistema e aguardar todas as respostas (Bertsekas et al., 1991). Já nos algoritmos com passagem de *token*, somente o processo que detém o *token* pode acessar o recurso compartilhado (Le Lann, 1977; Suzuki e Kasami, 1985; Raymond, 1989b; Naimi et al., 1996).

Nos modelos com pedido de permissão ainda podem ser incluídos os algoritmos baseados em quóruns (*quorum-based*) (Maekawa, 1985). Os quóruns são conjuntos de processos formados de tal forma que, se dois processos p_i e p_j solicitarem o acesso ao recurso, ao menos um outro processo p_k qualquer receberá ambas as solicitações. Cabe a p_k garantir a segurança, dando permissão a apenas um dos solicitantes de cada vez.

Em relação a dinamicidade, Saxena e Rai (2003) classificam os algoritmos de exclusão mútua em estáticos ou dinâmicos. Um algoritmo é estático se não faz uso de informações de execuções anteriores e não mantém informações sobre o estado atual do sistema. Quanto à disseminação de mensagens entre processos, as soluções podem fazer uso de topologias lógicas específicas ou difusão (*broadcast*). Em uma topologia, como árvore ou anel, as mensagens são transmitidas de acordo com as arestas que fazem parte da topologia adotada. Já na disseminação baseada em difusão, todo processo pode enviar mensagens a todos os outros em paralelo.

Uma variação dos algoritmos de exclusão mútua são as soluções que consideram a existência de mais de uma cópia do recurso. Nestes casos, quando k recursos estão disponíveis, é preciso garantir que, no máximo, k processos os estejam utilizando em cada instante de tempo. Este problema é conhecido como k-exclusão mútua.

3.2 Medidas de Desempenho

O desempenho em algoritmos de exclusão mútua é avaliado, em geral, de acordo com as métricas descritas nessa seção (Kshemkalyani e Singhal, 2008). A primeira é a complexidade de mensagens (MC), isto é, o total de mensagens necessárias por cada processo para obter acesso ao recurso. Já o atraso de sincronização (SD) mede o intervalo de tempo entre o instante em que um processo libera o recurso e o próximo processo consegue obtê-lo. Tem-se ainda o tempo de resposta, que é definido pelo intervalo de tempo entre o envio da solicitação e a liberação do recurso após o seu uso por um determinado processo. Estas métricas, que correspondem a intervalos de tempo, estão representadas na Figura 3.1. Por fim, o *throughput* define a taxa em que o sistema executa solicitações. Sendo E o tempo médio de utilização do recurso pelos processos, o *throughput* é dado por $1/(SD + E)$.

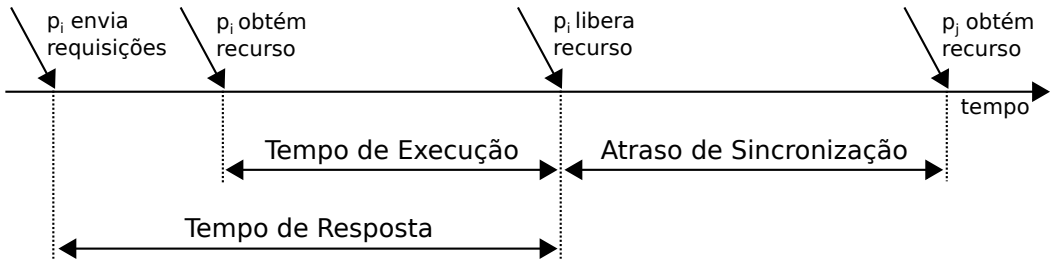


Figura 3.1: Métricas de desempenho para algoritmos de exclusão mútua.

Conforme observado por Fu et al. (2000), embora o total de mensagens seja importante em função da sobrecarga da rede, do ponto de vista do usuário do sistema, o tempo de resposta é prioritário. Isto é ainda mais evidente quando o algoritmo de exclusão mútua envolve um grande número de processos, o que implica em escalabilidade.

Saxena e Rai (2003) ainda acrescentam as métricas de tolerância a falhas e disponibilidade. A tolerância a falhas mede o número máximo de processos que podem falhar até que nenhum processo consiga mais acessar o recurso. A disponibilidade mede a probabilidade de um processo obter o recurso na presença de falhas.

Assim, um bom algoritmo de exclusão mútua é aquele que possui valores baixos para complexidade de mensagens e atraso de sincronização. Em termos qualitativos, espera-se que a solução apresente altas taxas de tolerância a falhas e disponibilidade.

A avaliação do desempenho ainda pode ser realizada utilizando variações de carga, conforme proposto por Kshemkalyani e Singhal (2008). Com *carga baixa*, em geral, não há concorrência nas solicitações. Com *carga alta*, vários processos enviam solicitações, sempre havendo solicitações pendentes. Em cenários com poucas solicitações simultâneas, o total de mensagens e a latência podem chegar a zero. Como exemplo, nas soluções baseadas em passagem de *token*, uma vez que o processo obtenha o *token*, poderá utilizá-lo infinitas vezes sem ter que requisitá-lo novamente.

3.3 Soluções com Pedido de Permissão

Nas soluções que utilizam pedido de permissão, cada processo que deseja fazer uso do recurso deve solicitar a todos os demais a permissão para utilizá-lo. Esta seção apresenta os principais algoritmos baseados em pedido de permissão.

3.3.1 Algoritmo de Lamport

A primeira solução para exclusão mútua distribuída baseada em pedido de permissão foi apresentada por Lamport (1978) e utiliza relógios lógicos para determinar a ordenação total das solicitações, conforme discutido na Seção 2.1. Quando um processo p_i deseja obter acesso ao recurso, envia uma mensagem REQUEST(i, ts_i) com o seu *timestamp* ts_i para todos os outros processos e armazena a sua solicitação em uma fila local. O envio é feito por *broadcast*. Um processo p_j que recebe uma mensagem de solicitação de p_i também armazena a mensagem na sua fila e retorna uma mensagem de resposta REPLY(j, ts_j) com o seu *timestamp* atualizado. Assim, p_i pode acessar o recurso quando o seu pedido é o primeiro da fila e ele já recebeu $n-1$ permissões com *timestamp* maior que ts_i. A Figura 3.2 ilustra um cenário com 3 processos em que os processos p_1 e p_2 solicitam recursos. Quando p_1 envia uma requisição, inclui o par (1,1) na sua fila, indicando que ele próprio está solicitando recurso. O processo p_2 executa o mesmo procedimento. Quando p_1 recebe a solicitação de p_2, inclui (2,1) na segunda posição da fila, já que tem prioridade de identificador, e responde com REPLY atualizado. O processo p_2 se comporta da mesma forma. Ao receber a resposta de p_3, p_1 verifica que é o primeiro da fila e que já tem todas as respostas necessárias, podendo utilizar o recurso.

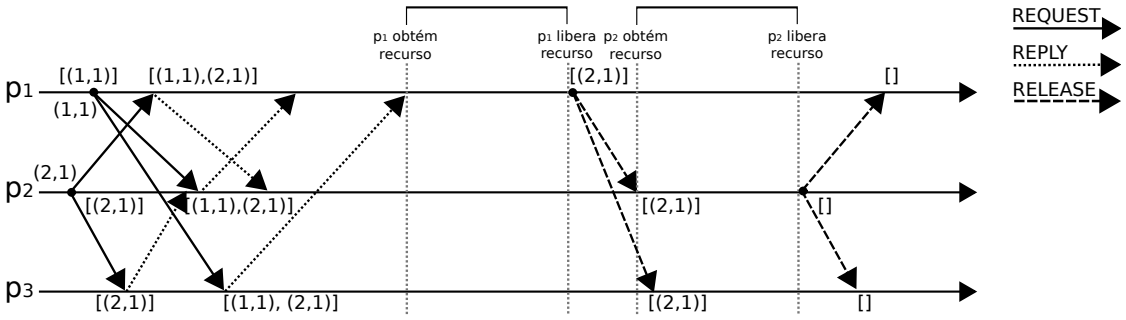


Figura 3.2: Execução do algoritmo de Lamport para $n = 3$. Cada solicitação é representada pelo par (processo, *timestamp*).

Quando o processo libera o recurso, envia uma mensagem de liberação RELEASE (também por *broadcast*) para que os demais processos retirem a solicitação das suas filas, dando oportunidade às demais solicitações pendentes. No exemplo da Figura 3.2, p_1 envia o RELEASE a p_2 e p_3 que removem a solicitação de p_1 de suas filas. Assim, a solicitação de p_2 passa a ser a primeira da sua fila e ele já pode fazer uso do recurso. Por fim, p_2 também envia o RELEASE e as filas são ajustadas, ficando vazias.

Cada fase de requisição, resposta e liberação gera $n-1$ mensagens, totalizando $3(n-1)$ mensagens por rodada. O atraso de sincronização é $1t$, sendo t o tempo necessário para a transmissão da mensagem de RELEASE.

3.3.2 Algoritmo de Ricart-Agrawala

O trabalho de Ricart e Agrawala (1981) aprimorou a solução de Lamport com um algoritmo que requer $2(n-1)$ mensagens. Ao invés de usar uma fila de solicitações local, se um processo p_i recebe uma mensagem de solicitação de um processo p_j, mas p_i está utilizando ou tentando utilizar o recurso e tem maior prioridade que p_j de acordo com o relógio lógico, p_i retém a resposta de permissão e adiciona a solicitação de p_j em uma lista de requisições pendentes. Assim, quando o processo p_i liberar o recurso, ele enviará todas as mensagens de resposta adiadas, permitindo que os demais processos interessados no recurso tenham a chance de obtê-lo, conforme a prioridade dada pelo relógio lógico.

Como exemplo, considere o cenário da Figura 3.3. As setas contínuas representam as mensagens de REQUEST e as pontilhadas as de REPLY. Os processos p_1 e p_2 solicitam o recurso de forma concorrente. Quando p_2 recebe a solicitação de p_1 responde imediatamente, pois ambos possuem o mesmo valor de relógio e o de menor identificador tem prioridade. Pela mesma razão, quando p_1 recebe a solicitação de p_2 ele retém a resposta. Ao receber as permissões de p_2 e p_3, p_1 obtém acesso ao recurso. Ao liberá-lo, envia para p_2 a resposta retida, permitindo que o mesmo acesse o recurso.

Esta solução garante que, para cada par de processos solicitando o recurso, aquele com maior prioridade adiará a resposta para o de menor prioridade. Assim, apenas o processo

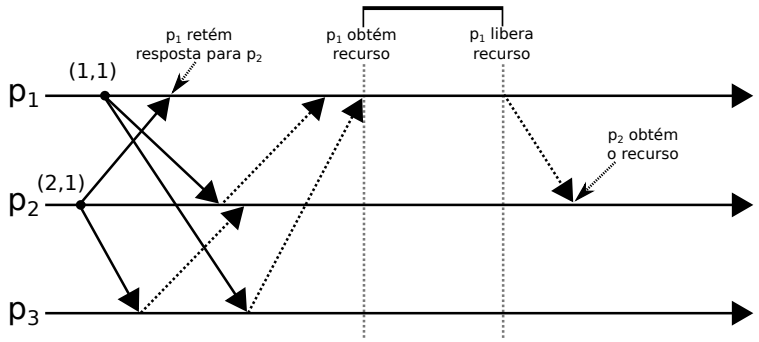


Figura 3.3: Execução do algoritmo de Ricart-Agrawala para $n = 3$.

de maior prioridade conseguirá todas as $n - 1$ respostas de permissão necessárias para acessar o recurso de forma exclusiva, garantindo a exclusão mútua.

3.3.3 Algoritmo de Singhal

Os algoritmos de Lamport e Ricart-Agrawala utilizam uma abordagem estática para solicitar a permissão, isto é, o processo de solicitação é sempre o mesmo, independente do estado do sistema. Singhal (1992), pelo contrário, propôs uma solução de exclusão mútua adaptativa que visa explorar a dinamicidade do sistema e otimizar o desempenho. O algoritmo de Singhal baseia-se no seguinte cenário: se alguns processos solicitam recursos com maior frequência e os outros com pouca ou nenhuma frequência, aqueles que solicitam com mais frequência não precisam solicitar a permissão aos que solicitam com pouca frequência sempre que quiserem utilizar o recurso. Para tanto, o algoritmo faz uso de dois conjuntos: o conjunto R_i (*request set*), que contém os processos dos quais o processo p_i precisa obter a permissão e o conjunto I_i (*inform set*), com os processos para os quais p_i deve enviar a permissão depois que terminar de utilizar o recurso.

Inicialmente, seguindo a prioridade dos relógios lógicos, R_i contém todos os processos com identificador menor que o de p_i e I_i está vazio. Quando um processo p_i está tentando obter o recurso e recebe um pedido de permissão de um processo p_j com maior prioridade ele envia o REPLY imediatamente para p_j e, se $p_j \notin R_i$, envia REQUEST para p_j e adiciona p_j em R_i. Por outro lado, se p_i tem prioridade sobre p_j, p_i apenas adiciona p_j em I_i, garantindo que p_j será notificado assim que p_i liberar o recurso. O mesmo acontece se p_i receber o pedido de p_j enquanto estiver utilizando o recurso. Por fim, se p_i receber REQUEST de p_j e p_i não está solicitando e nem utilizando o recurso, p_i adiciona p_j em R_i e envia imediatamente o REPLY para p_j.

Quando p_i deseja acessar o recurso, envia REQUEST para todos os processos em R_i e aguarda até receber REPLY de todos. Para cada REPLY recebido de p_j, p_i remove p_j de R_i. Assim, considerando uma ordenação p_1 a p_n da esquerda para a direita, a cardinalidade de R_i diminui da direita para a esquerda, criando um padrão escada (em

inglês, *staircase pattern*). Como exemplo, considere os 5 processos no estado inicial da Figura 3.4. Cada coluna representa o conjunto R_i de cada processo p_i. Na figura mais à esquerda, o processo p_5 solicita o recurso enviando REQUEST para todos os processos em R_5. Cada processo p_j, ao receber REQUEST de p_5 e não estando utilizando, nem solicitando recurso, envia imediatamente REPLY para p_5 e o inclui no conjunto R_j. Para cada REPLY recebido, p_5 se desloca uma posição para a esquerda até alcançar a primeira posição e poder acessar o recurso. O mesmo acontece quando p_3 é o solicitante. Note que p_4, estando do lado direito da escada, não participa e se mantém inalterado. Assim, os processos que solicitam com menos frequência tendem a estar à direta e os que solicitam com mais frequência tendem a manter-se mais à esquerda.

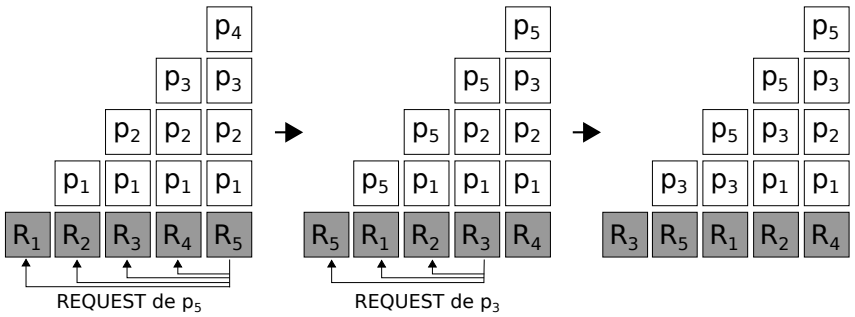


Figura 3.4: Execução do algoritmo de Singhal para $n = 5$.

O total de mensagens necessárias para cada solicitação no algoritmo de Singhal pode variar de zero a $3(n-1)/2$, de acordo com a quantidade de processos solicitando recursos simultaneamente e com a sua posição na escada.

Utilizando uma abordagem dinâmica e baseados no algoritmo de Ricart-Agrawala, Carvalho e Roucairol (1983) propuseram uma otimização que mantém a complexidade de mensagens entre zero e $2(n-1)$. O algoritmo parte da premissa de que, se um processo p_i recebeu um REPLY do processo p_j, p_i pode fazer uso do recurso quantas vezes precisar sem ter que consultar p_j até que p_j solicite permissão a p_i. Esta modificação diminui a quantidade de mensagens quando um pequeno conjunto de processos solicita recursos. Considere, por exemplo, o cenário em que apenas um processo solicita recurso. Após o recebimento de todas as permissões da primeira solicitação, ele poderá fazer uso do recurso infinitas vezes sem ter que enviar novas solicitações.

Posteriormente, Lodha e Kshemkalyani (2000) propuseram uma adaptação do algoritmo de Ricart-Agrawala com base na seguinte observação: se um processo solicitou um recurso e está aguardando para utilizá-lo, ele não precisa aguardar todas as permissões, mas apenas aquelas de processos que possuem solicitações com maior prioridade. Os pedidos de permissão continuam sendo feitos por mensagens de REQUEST(i, ts_i) em *broadcast* e cada processo mantém uma fila local de requisições ordenada pelos *timestamps*. Mensagens de REPLY são enviadas apenas por processos que não solicitaram recurso. Assim, se

apenas um processo está solicitando recursos, ele receberá as $n-1$ mensagens de REPLY esperadas. Entretanto, se dois ou mais processos concorrem por recursos, somente uma mensagem de FLUSH é envida do processo que acabou de utilizá-lo para o próximo processo na fila. Essa mensagem representa uma mensagem de REPLY coletiva, que engloba a permissão de todos os outros processos com requisições de menor prioridade. Nestes casos, são enviadas $n-1$ mensagens de REQUEST e $n-|CSet_i|$ mensagens de REPLY por requisição, sendo $CSet_i$ o conjunto dos processos concorrentes. No total, o algoritmo gera $2n-|CSet_i|$ mensagens por requisição. Se todos os processos são concorrentes, apenas n mensagens são utilizadas.

3.3.4 Algoritmo de Maekawa

Uma variação dos algoritmos de exclusão mútua com pedido de permissão são as soluções baseadas em quóruns. Quóruns são subconjuntos de processos tais que, dado um conjunto de quóruns C, conhecido como *coterie*, $\forall g, h \in C, g \cap h \neq \emptyset$ e $g \not\supseteq h$ (Garcia-Molina e Barbara, 1985). Essas propriedades garantem que sempre existirá um elemento comum entre dois quóruns e que nenhum quórum estará inteiramente contido em outro. Considere, por exemplo, os conjuntos $\{1,2,3\}$, $\{2,5,7\}$, $\{5,7,9\}$ e $\{2,3\}$. Tais conjuntos não são válidos porque o primeiro e o terceiro não têm elementos em comum e o último está contido no primeiro.

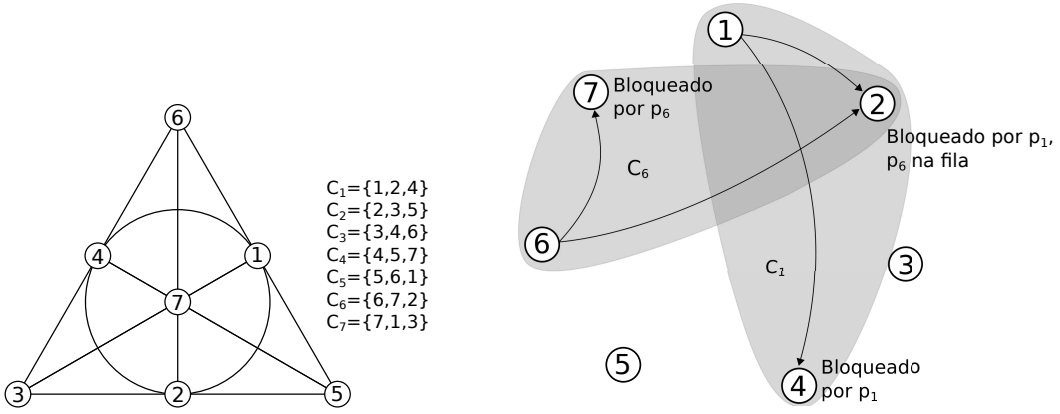
Em termos gerais, em uma solução baseada em quóruns, se um processo p_i pertencente a um quórum C_i deseja utilizar o recurso, ele envia REQUEST a todos os demais processos em C_i. Como C_i contém ao menos um processo p_j em comum com o quórum de cada outro processo e p_j só fornece uma única permissão de cada vez, a exclusão mútua está garantida. A vantagem destes algoritmos é a redução da quantidade de mensagens em cada solicitação, visto que apenas um conjunto de processos precisa ser consultado.

O algoritmo de Maekawa (1985) foi a primeira solução baseada em quóruns. Para construir os quóruns ele utiliza a teoria de planos projetivos para criar quóruns com tamanho $\sqrt{n}$. Planos projetivos são estruturas geométricas derivadas do plano nas quais quaisquer duas linhas se intersectam em um, e somente um, ponto. A Figura 3.5(a) apresenta como exemplo o menor plano projetivo de ordem 2, com 7 pontos e 7 linhas, conhecido como *plano de Fano* (Baez, 2002) (homenagem ao matemático italiano Gino Fano). Em planos projetivos, a ordem d implica em $d+1$ pontos em cada linha e $d+1$ linhas passando por cada ponto. Os conjuntos do plano de Fano são gerados a partir do elemento $\{1,2,4\}$ pela adição de 1 em cada entrada, módulo 7.

A construção dos quóruns proposta por Maekawa obedece quatro condições:

1. $\forall i, j, i \neq j, 1 \leq i, j \leq n : C_i \cap C_j \neq \emptyset$

2. $\forall i, 1 \leq i \leq n : p_i \in C_i$



(a) Plano projetivo de Fano (b) Exemplo

Figura 3.5: Algoritmo de Maekawa para $n = 7$, $|C_i| = 3$.

3. $\forall i, 1 \leq i \leq n : |C_i| = k = \sqrt{n} + 1$

4. $\forall i, j, 1 \leq i, j \leq n$: todo p_i está presente em k conjuntos C_j

Assim como no algoritmo de Lamport, a solução de Maekawa também utiliza mensagens de REQUEST, REPLY e RELEASE. Cada processo p_i que deseja utilizar o recurso envia mensagens de REQUEST a todos os demais processos p_j pertencentes ao seu quórum e aguarda pelas respostas de REPLY. Os processos que recebem o REQUEST verificam se já deram permissão a outro processo. Se a permissão ainda não foi dada, enviam REPLY para p_i. Caso contrário, adicionam a requisição de p_i em uma fila.

Quando um processo libera o recurso, envia RELEASE para todos os processos no seu quórum. Quando um processo recebe um RELEASE, envia o REPLY para o próximo processo na fila de pendentes e o remove da fila. A fila vazia indica que o processo não enviou mensagens de REPLY desde a última mensagem de RELEASE recebida, isto é, o próximo processo que solicitar o recurso a ele receberá o REPLY imediatamente.

Como exemplo, considere um sistema com 7 processos $\{p_1, p_2, .., p_7\}$ representado na Figura 3.5(b) e o conjunto de quóruns obtido da Figura 3.5(a). Cada processo p_i está organizado em um quórum C_i que contém, além do próprio p_i, dois outros identificadores de processos. Assim, se p_1 deseja utilizar o recurso, ele envia REQUEST para o quórum C_1 que contém p_2 e p_4. Os processos p_2 e p_4 enviam REPLY e bloqueiam em p_1, isto é, não respondem a mais nenhuma solicitação enquanto p_1 não enviar o RELEASE. Considere que em seguida p_6 solicita o recurso. O quórum de p_6 inclui p_2 e p_7, mas p_2 já está bloqueado na requisição de p_1. Em função das propriedades dos quóruns, qualquer processo não contido no quórum C_1 de p_1 terá uma intersecção com algum processo contido em C_1, garantindo a exclusão mútua.

Considerando que os quóruns têm tamanho máximo $\sqrt{n}$, cada solicitação de recurso utiliza 3 $\sqrt{n}$ mensagens. Entretanto, conforme comprovado por Singhal (1991), o algoritmo de Maekawa está sujeito a *deadlocks*, visto que as solicitações não são priorizadas

pelos seus *timestamps*. Assim, um processo pode enviar REPLY a um outro processo solicitante e, posteriormente, forçar que um outro processo de maior prioridade aguarde na fila. A solução para este problema requer um mecanismo auxiliar de recuperação e, consequentemente, mensagens adicionais que podem aumentar o total para $5\sqrt{n}$ mensagens por solicitação (Kshemkalyani e Singhal, 2008). O atraso de sincronização é de $2t$, composto pela mensagem de RELEASE do processo que liberou o recurso e pelo REPLY do processo na intersecção dos quóruns.

Uma abordagem híbrida utilizando os algoritmos de Maekawa e Singhal foi proposta por Chang (1996). Os processos do sistema são divididos em g grupos disjuntos de tamanho n/g. O algoritmo de Singhal é utilizado para resolver conflitos locais (intra-grupo), minimizando o atraso, e o algoritmo de Maekawa para comunicação global (inter-grupos), reduzindo a quantidade de mensagens. Resultados de simulação apontam uma redução de 52% no total de mensagens e 29% no atraso de sincronização em relação ao algoritmo de Maekawa.

Outras soluções de exclusão mútua baseadas em quóruns podem ser encontradas em Agrawal e El Abbadi (1991) e Cao et al. (1998).

3.4 Soluções com Passagem de *Token*

Nos algoritmos de exclusão mútua baseados em passagem de *token*, um único *token* compartilhado é utilizado para indicar quem tem a permissão de utilizar o recurso. Um *token* pode ser implementado com uma mensagem especial de privilégio. Uma diferença importante em relação às soluções baseadas em solicitação de permissão é que na passagem de *token* não são utilizados *timestamps*, mas sim números de sequência (Kshemkalyani e Singhal, 2008). Cada processo incrementa seu contador sequencial individualmente e este contador é utilizado fundamentalmente para distinguir solicitações antigas das novas.

Esta seção apresenta as duas soluções de exclusão mútua com passagem de *token* mais conhecidas. Outros algoritmos podem ser encontrados em: Helary et al. (1988) (*broadcast* e estático), Singhal (1989); Yan et al. (1996) (*broadcast* e dinâmicos) e Naimi et al. (1996); Hélary et al. (1994) (organização lógica e dinâmicos).

3.4.1 Algoritmo de Suzuki-Kassami

Um exemplo de algoritmo com passagem de *token* é o algoritmo de Suzuki e Kasami (1985). Nesta solução, cada processo p_i mantém um número de sequência de requisições sn_i e um vetor RN_i para armazenar o valor da última requisição recebida de cada processo. Este vetor é utilizado para distinguir uma requisição antiga atrasada de uma atual. Requisições atrasadas são identificadas quando $RN_j[i] > sn_i$. Quando um processo p_i deseja acessar o recurso, ele incrementa o seu contador sn_i e envia por *broadcast* uma

mensagem de REQUEST(i, sn_i) a todos os outros processos. Os processos p_j que recebem o pedido de p_i, atualizam $RN_j[i] = max(RN_j[i], sn_i)$. Se o processo que está com o *token* não está acessando o recurso, ele envia o *token* imediatamente para o solicitante. Caso contrário, após liberar o recurso, o processo verifica as solicitações pendentes e envia o *token* para o próximo processo na fila.

Para determinar se uma requisição ainda é válida, o algoritmo utiliza um vetor de requisições LN, mantido no próprio *token*. Cada entrada $LN[i]$ armazena o número de sequência da última solicitação atendida do processo p_i. Toda vez que um processo p_i termina de utilizar o recurso, atualiza $LN[i] = sn_i$ e verifica em cada posição do vetor LN se $RN_i[j] = LN[j] + 1$. Se a condição é verdadeira, o processo p_j possui uma requisição pendente e é adicionado na fila. Ao final, o *token* é enviado para o primeiro processo da fila. Se a fila está vazia, o processo mantém o *token* e pode utilizá-lo quantas vezes desejar até receber uma nova solicitação de outro processo.

A grande vantagem do algoritmo de Suzuki-Kassami é a simplicidade e a eficiência. Se o processo não detém o *token* são necessárias, em média, n mensagens para obtê-lo.

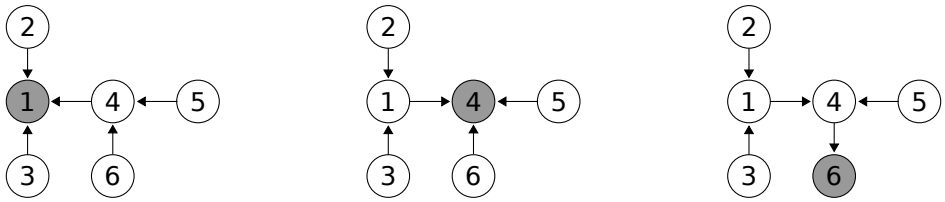
3.4.2 Algoritmo de Raymond

Uma segunda solução eficiente para o problema da exclusão mútua com passagem de permissão foi proposta por Raymond (1989b). O algoritmo utiliza uma árvore geradora (*spanning tree*) mínima para reduzir a quantidade de mensagens. Considerando o sistema distribuído modelado por um grafo $G = (V, E)$, conforme definido na Seção 2.1, uma árvore geradora $T(G) = (V, E')$ é um grafo conexo e acíclico que contém todos os vértices de G. A árvore é mínima se possui custo mínimo. Se todas as arestas têm o mesmo peso, como é considerado no algoritmo, toda árvore geradora é mínima. Um exemplo de árvore geradora mínima está representado na Figura 3.6.

O algoritmo de Raymond pode ser classificado como baseado em *token* porque utiliza o seguinte conceito. Uma mensagem especial circula entre os processos e aquele que a recebe tem permissão para utilizar o recurso compartilhado. Cada processo mantém um atributo `HOLDER` que indica na árvore o sentido do processo que possui o *token*. Desta forma, $\texttt{HOLDER}_i{=}j$ indica que, a partir do processo p_i, o *token* está no sentido do processo p_j. Portanto, se p_i deseja utilizar o recurso, deve enviar o REQUEST para p_j.

Como exemplo, considere uma organização lógica em árvore do algoritmo de Raymond para $n = 6$ processos representada na Figura 3.6. A orientação das arestas indica quem é o `HOLDER` de cada processo. Neste exemplo, p_1 detém o *token* e, portanto, $\texttt{HOLDER}_1{=}1$, $\texttt{HOLDER}_2{=}1$, $\texttt{HOLDER}_3{=}1$, $\texttt{HOLDER}_4{=}1$, $\texttt{HOLDER}_5{=}4$ e $\texttt{HOLDER}_6{=}4$.

Cada processo mantém localmente além de `HOLDER`, uma fila Q com as requisições pendentes. Para exemplificar o uso desta fila, considere que no mesmo cenário da Figura 3.6(a) p_4 solicite o *token*. Como $\texttt{HOLDER}_4{=}1$, p_4 envia REQUEST para p_1 e inclui



(a) *token* em p_1 (b) *token* em p_4 (c) *token* em p_6

Figura 3.6: Algoritmo de Raymond para $n = 6$.

na sua fila local o seu próprio identificador. Portanto, $Q_4 = \{p_4\}$. Considere ainda que, no instante seguinte, p_6 solicite o *token*. O processo p_4 irá receber a solicitação de p_6, já que $\texttt{HOLDER}_6$=4. Neste caso, p_4 adiciona p_6 na sua fila, mas não repassa o REQUEST, visto que já possui uma solicitação pendente para ele próprio. Quando p_1 libera o *token*, envia para p_4 e atualiza $\texttt{HOLDER}_1$=4, conforme Figura 3.6(b). Quando p_4 receber o *token* de p_1, verá que o próximo da fila é ele mesmo, podendo fazer uso do recurso. Ao liberar o recurso, p_4 enviará o *token* para p_6, que é o próximo na fila, e atualizará $\texttt{HOLDER}_4$=6, modificando a árvore de acordo com a Figura 3.6(c). Como todo algoritmo baseado em passagem de permissão, a segurança desta solução está baseada na existência de um único *token* no sistema. Com isso, apenas um único processo pode utilizar o recurso em cada instante de tempo.

O total de mensagens transmitidas a cada solicitação do algoritmo de Raymond é igual a duas vezes o caminho mais longo na árvore. No pior caso, todos os processos estão organizados em uma sequência linear com tamanho máximo $n - 1$. Portanto, são necessárias $2(n-1)$ mensagens por solicitação. No entanto, se todos os processos solicitam recursos com alguma frequência, a média do número de mensagens é $2n/3$, visto que a distância média entre o processo solicitante e o processo que possui o *token* é $(n + 1)/3$. A melhor topologia é a estrela radial Raymond (1989b) que apresenta complexidade $O(\log_{k-1} n)$, onde k é o grau de cada vértice não-folha. Em termos gerais, em uma árvore de topologia arbitrária, a complexidade de mensagens é empiricamente calculada como $O(\log n)$.

3.5 Soluções de k-Exclusão Mútua

Uma extensão do problema da exclusão mútua é a k-exclusão mútua. Nesta categoria, ao invés de um, existem k cópias idênticas do recurso compartilhadas no sistema. Uma vez que cada uma das cópias pode ser acessada por somente um processo em cada instante de tempo, o objetivo é garantir que, nó máximo, k processos obtenham acesso aos recursos de cada vez e que todos os processos que solicitarem recursos consigam obtê-lo em um tempo finito. Estas duas propriedades: segurança e progressão, respectivamente, são as mesmas exigidas pela exclusão mútua com somente um recurso.

As soluções de k-exclusão mútua também dividem-se em pedido e passagem de permissão. Esta seção apresenta os algoritmos de Raymond (1989a); Pissinou et al. (1996); Srimani e Reddy (1992); Bulgannawar e Vaidya (1995). Outra soluções são propostas em Makki et al. (1992); Huang et al. (1993); Kakugawa et al. (1994). Os dois últimos utilizam o conceito de k-*coterie* para abordar a k-exclusão mútua utilizando a estrutura de quóruns descrita na Seção 3.3.

3.5.1 Algoritmo de Raymond

O algoritmo de Raymond (1989a) foi a primeira solução proposta para o problema da k-exclusão mútua. Ele utiliza a mesma abordagem de pedido de permissão do algoritmo de exclusão mútua de Ricart e Agrawala (1981) apresentado na Seção 3.3. Quando um processo deseja utilizar um recurso compartilhado, ele envia mensagens de requisição aos $n-1$ outros processos e aguarda por, no mínimo, $n-k$ mensagens de permissão. Se nenhum processo está utilizando ou solicitando recursos, o total de respostas pode chegar a $n-1$. Portanto, no pior caso, são geradas $2(n-1)$ mensagens por solicitação.

A Figura 3.7 ilustra o comportamento do algoritmo de Raymond em um sistema com 3 processos e 2 recursos. Neste exemplo, um processo deve aguardar pelo menos $n-k=1$ permissão para acessar o recurso. Inicialmente, o processo p_1 envia uma mensagem de requisição para todos os demais, solicitando a permissão. Como nenhum deles está utilizando um recurso ou tentando obtê-lo, todos respondem imediatamente, permitindo que p_1 utilize o recurso. Em seguida, o processo p_2 efetua o pedido de permissão. Como o processo p_3 não está interessado em recursos, ele responde imediatamente. O processo p_1, ao contrário, retém a resposta até a liberação do recurso. De qualquer forma, p_2 obtém o número de permissões necessárias e acessa o recurso.

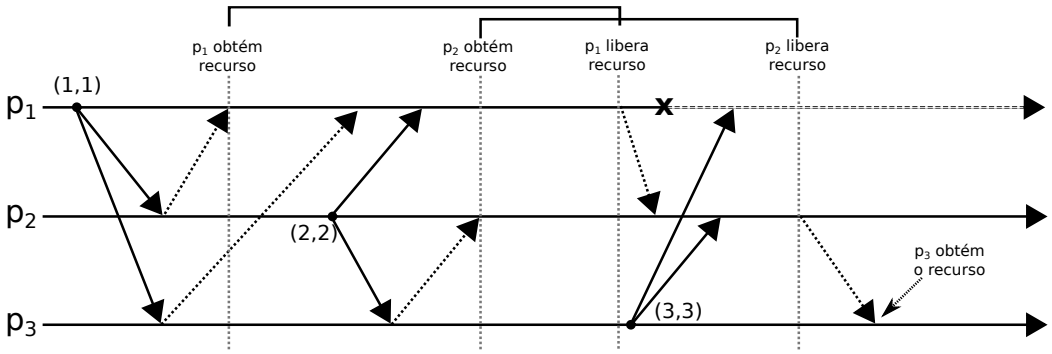


Figura 3.7: Execução do algoritmo de Raymond para $n=3$ e $k=2$ sem falhas.

Como é preciso aguardar somente $n-k$ respostas para iniciar o uso do recurso, é possível que respostas às solicitações anteriores sejam confundidas com novas permissões. Para evitar este problema um número de sequência é incluído em cada mensagem.

Intrinsecamente, o algoritmo de Raymond tolera $k-1$ processos falhos. No entanto, cada processo falho degrada a solução, pois pode ser que o número de processos não-falhos que não desejam recursos seja insuficiente, fazendo com que o processo solicitante tenha que aguardar um processo sem-falha liberar o recurso e dar a permissão, mesmo que um recurso esteja livre. Voltando ao exemplo anterior, considere o instante em que p_3 solicita recursos. O processo p_1 está falho e o processo p_2 retém a resposta porque está utilizando um recurso. Neste caso, p_3 fica bloqueado aguardando a permissão até que p_2 envie a resposta retida, embora ainda exista um recurso livre.

Visando aumentar a eficiência do algoritmo de Raymond na obtenção de recursos, o trabalho de Bouillaguet et al. Bouillaguet et al. (2008) propôs uma solução que utiliza um detector de falhas e aumenta a eficiência, tolerando até $n-1$ processos falhos. Posteriormente, os mesmos autores propuseram em Bouillaguet et al. (2009) uma segunda solução que dispensa o uso de detectores de falhas e de mensagens extras para detecção de nodos falhos. As informações de estado dos processos são integradas às mensagens do próprio algoritmo de exclusão mútua, que tolera até $k-1$ falhas.

3.5.2 Algoritmo de Pissinou et al.

O algoritmo proposto por Pissinou et al. (1996) utiliza como base a solução de Lamport descrita na Seção 3.3. Trata-se, portando, de um algoritmo de k-exclusão mútua com pedido de permissão. O algoritmo utiliza quatro tipos de mensagens. As mensagens de REQUEST(i, ts_i) são utilizadas para solicitar recursos. As mensagens de GRANT(j, ts_j) são enviadas pelos processos que recebem o REQUEST e não estão utilizando um recurso ou estão tentando utilizar, mas possuem menor prioridade. As mensagens de REPLY(j, ts_j) são enviadas pelos processos que também estão em processo de requisição e possuem maior prioridade que a do pedido recebido. Por fim, as mensagens de RELEASE são enviadas quando o processo libera um recurso.

Cada processo possui dois grupos distintos *RequestSet* e *GrantSet* de tamanho $(n-1)/2$ cada, isto é, a metade do número total de processos. Os processos no grupo $RequestSet_i$ do processo p_i possuem identificadores entre $i+1$ e $(i+n/2) \mod n$. Os demais processos estão em *GrantSet*. Desta forma, dado qualquer par de processos p_i e p_j, $p_i \in RequestSet_j$ e $p_j \in GrantSet_i$ ou vice-versa.

Quando um processo p_i deseja utilizar um recurso, envia REQUEST para todos os processos no seu *RequestSet*. Os processos p_j que recebem o REQUEST respondem com GRANT ou REPLY. É possível que, enquanto aguarda pelas respostas, p_i receba solicitações de recurso de outros processos no seu *GrantSet*. Se o REQUEST recebido por p_i tem menor prioridade, ele envia REPLY. Caso contrário, p_i envia GRANT. Considerando k recursos no sistema, o processo p_i pode acessar um deles quando a soma do total de mensagens de REPLY em $RequestSet_i$ e de GRANT em *GrantSet* é menor que k. Ao

finalizar o uso do recurso, o processo p_i envia RELEASE a todos os processos no seu *RequestSet* e aos processos do *GrantSet* aos quais p_i enviou REPLY.

O fato de enviar solicitações para apenas metade dos processos diminui em 25% a complexidade de mensagens em relação a proposta de Lamport. No pior caso, $2(n-1)$ mensagens são utilizados por requisição, mas na média o número de mensagens é menor.

3.5.3 Algoritmo de Srimani-Reddy

Uma solução para k-exclusão mútua com passagem de *token* foi proposta por Srimani e Reddy (1992). A solução é similar ao algoritmo de Suzuki e Kasami (1985) para um único *token* abordada na Seção 3.4. Inicialmente são introduzidos k *tokens* no sistema. Cada processo que deseja utilizar uma cópia do recurso, envia uma mensagem a todos os outros processos e aguarda o recebimento do *token*.

A complexidade de mensagem do algoritmo de Srimani-Reddy varia de 0 a $n+k-1$. Se o processo que deseja usar o recurso possui o *token*, nenhuma mensagem é enviada. No pior caso, se o processo não detém o *token* e cada *token* está em um processo diferente, serão enviadas $n-1$ mensagens de REQUEST e recebidas $n-1$ mensagens de REPLY, além das k mensagens contendo o *token*. Considerando que um processo solicite L vezes um recurso, em média serão utilizadas $(n+k-1)+2(n-1)/L$. Sendo L muito grande, a complexidade fica em $n+k-1$ mensagens por requisição.

3.5.4 Algoritmo de Chaudhuri-Edward

Chaudhuri e Edward (2008) propuseram uma solução hierárquica de k-exclusão mútua baseada em *tokens* que apresenta complexidade de mensagens em $O(\sqrt{n})$ e atraso de sincronização 4. O nodos são divididos em $\sqrt{n}$ grupos locais (LG) de $\sqrt{n}$ elementos. Cada grupo possui um nodo responsável pelas requisições locais, chamado LRC (*Local Request Collector*). Cada nodo LRC mantém uma fila local LRQ (*Local Request Queue*). O conjunto de nodos LRC formam o grupo global (GG) e um nodo pertencente a GG é eleito como GRC (*Global Request Collector*). Este nodo é responsável por gerenciar os *tokens* e manter a fila global GRQ (*Global Request Queue*). Quando um processo deseja utilizar um recurso, envia uma solicitação para o seu LRC. Se o LRC não possui um *token*, ele repassa a solicitação para o GRC.

A Figura 3.8 ilustra a organização hierárquica utilizada pelo algoritmo de Chaudhuri-Edward para 16 nodos. Cada nodo está identificado por (grupo, identificador). Os nodos (1,1), (2,1), (3,1) e (4,1) formam o grupo global e o nodo (3,1) foi eleito o GRC, responsável por manter a fila global de requisições. No exemplo, os nodos (1,2) e (1,3) solicitam o *token* ao LRC_1 do grupo 1 (LG_1), que armazena os pedidos na fila local e repassa as solicitações para o GRC.

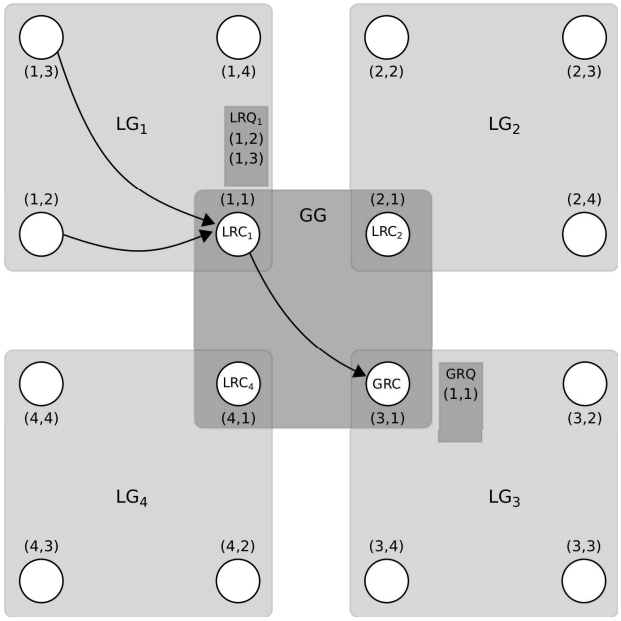


Figura 3.8: Organização hierárquica do algoritmo de Chaudhuri-Edward para $n = 16$.

No melhor caso, a carga do sistema é baixa e o processo que deseja utilizar o recurso é o *GRC*. Neste caso, o atraso é nulo e nenhuma mensagem é enviada. Se o processo solicitante está em um grupo local que não possui um *token*, como ilustrado no exemplo da Figura 3.8, serão geradas duas mensagens de REQUEST para alcançar o *LRC* e o *GRC*, mais duas mensagens de resposta com *token* e mais duas mensagens para o retorno do *token* até o *GRC*. Para m requisições são geradas $6m + \sqrt{n}$ mensagens, isto é, $6 + \sqrt{n}/m$ mensagens por solicitação. No pior caso, quando a carga do sistema é alta e todos os processos possuem requisições pendentes, n mensagens de REQUEST são geradas. Destas, $\sqrt{n} - 1$ mensagens são de processos do *LRC* para o *GRC*. k *tokens* são enviados para k *LRC*s que circulam o *token* entre os $\sqrt{n}$ processos do grupo local. Em síntese, a complexidade é $O(\sqrt{n})$.

O atraso de sincronização máximo é $4t$, composto pelo retorno do *token* de um processo em um grupo local até o *GRC* (2 mensagens) e o envio deste mesmo *token* para um outro processo em outro grupo local (2 mensagens).

3.6 Considerações Finais

Este capítulo apresentou o problema da exclusão mútua em sistemas distribuídos e os principais algoritmos que o resolvem. Foram apresentadas soluções tanto baseadas em pedido de permissão quanto em passagem de *token*, além das soluções relevantes de k-exclusão mútua. Como visto na Seção 3.2, a avaliação das soluções é realizada principalmente com base na complexidade de mensagens e no atraso de sincronização. A Tabela 3.1 apresenta

Tabela 3.1: Comparativo dos principais algoritmos de exclusão mútua.

Algoritmos de 1-exclusão mútua	Pedido de permissão				Passagem de *token*	
	Lamport (1978)	Ricart e Agrawala (1981)	Singhal (1992)	Maekawa (1985)	Suzuki e Kassami (1985)	Raymond (1989)
Complexidade de mensagens	$3(n-1)$	$2(n-1)$	$(n-1)$ a $3(n-1)/2$	$3\sqrt{n}$ a $5\sqrt{n}$	n	$O(\log n)$
Atraso de sincronização	$1t$	$1t$	$1t$	$2t$	$1t$	$t \log n$

Algoritmos de k-exclusão mútua	Pedido de permissão		Passagem de *token*	
	Raymond (1989)	Pissinou et al. (1996)	Srimani e Reddy (1992)	Chaudhuri e Edward (2008)
Complexidade de mensagens	$2(n-1)$	$2(n-1)$	0 a $n+k-1$	0 a $O(\sqrt{n})$
Atraso de sincronização	$1t$	$1t$	$1t$	$4t$

um comparativo dos algoritmos apresentados neste capítulo, sendo n o total de processos do sistema e t o tempo de transmissão de uma mensagem entre dois processos.

O algoritmo de Lamport foi a primeira solução proposta para o problema da exclusão mútua distribuída e é também a mais custosa, utilizando $3(n-1)$ mensagens por solicitação. Alguns algoritmos apresentam complexidade menor por utilizarem soluções adaptativas, que se ajustam de acordo com o estado do sistema. O algoritmo de Singhal, por exemplo, consegue reduzir para $n-1$ o total de mensagens por solicitação quando a carga do sistema é baixa. Soluções baseadas em quóruns também reduzem significativamente a quantidade de mensagens, mas podem aumentar a latência, como é o caso do algoritmo de Maekawa. O mesmo ocorre com soluções que fazem uso de topologias arbitrárias para disseminação de mensagens ao invés de *broadcast*. Em geral, a redução da quantidade de mensagens tem impacto negativo na latência.

Em relação à tolerância a falhas, poucos algoritmos apresentam soluções para manter a eficiência na obtenção de recursos em cenários com falhas. O algoritmo de k-exclusão mútua de Raymond, por exemplo, consegue tolerar de forma implícita até $k-1$ falhas, mas com eficiência degradada. Nas soluções com pedido de permissão, uma forma de tratar falhas é utilizar mecanismos de monitoramento. Uma vez detectado que o processo está falho, não é mais preciso aguardar por permissões dele. No caso dos algoritmos com passagem de ficha, além de detectar a falha dos processos, é preciso tratar os casos em que as fichas são perdidas durante uma falha de processo. O Capítulo 6 apresenta uma solução de k-exclusão mútua tolerante a falhas com pedido de permissão que garante a eficiência para até $n-1$ processos falhos.

CAPÍTULO 4

UM ALGORITMO AUTONÔMICO DE ÁRVORE GERADORA

Árvores geradoras (*spanning trees*) são abstrações utilizadas em sistemas distribuídos para definir uma rede de baixo custo que conecta todos os processos do sistema, sendo empregadas na solução de diversos problemas, como exclusão mútua, agrupamento, fluxo em redes, sincronização e difusão de mensagens (*broadcast*) (Elkin, 2006; England et al., 2007; Dahan et al., 2009). Para a difusão de mensagens, por exemplo, uma solução alternativa simples é utilizar inundação (*flooding*) para enviar uma mensagem a todos os processos do sistema. O inconveniente desta solução é o custo da comunicação devido ao grande número de vezes que a mensagem é retransmitida a um mesmo processo. Por outro lado, as árvores geradoras minimizam este custo, visto que a mensagem precisa ser transmitida somente pelas $n-1$ arestas da árvore (Gärtner, 2003).

O algoritmo proposto neste capítulo permite a construção sob demanda de uma árvore geradora mínima distribuída a partir de um processo fonte qualquer, que atua como raiz principal da árvore. A partir da raiz principal, o restante da árvore é construído com base na topologia baseada em hipercubo virtual, denominada VCube (Ruoso, 2013), descrita na Seção 4.3. Em função da topologia em hipercubo, a solução apresenta importantes propriedades logarítmicas, mesmo quando processos falham. Além disso, as árvores são reconstruídas de forma autonômica após a ocorrência de falhas.

O algoritmo é dito autonômico porque organiza automaticamente os processos do sistema, reconstrói a árvore dinamicamente à medida que processos falhos são detectados e mantém a árvore mínima, provendo eficiência. Uma das vantagens da solução é que, ao invés de iniciar a reconstrução a partir do processo fonte, a regeneração da árvore é realizada apenas localmente, de acordo com o ramo afetado.

4.1 Definição do Problema

Seja $G = (V, E)$ o grafo conexo e não-direcionado que representa o sistema distribuído Π, no qual $n = |V|$ são os vértices (processos) e $m = |E|$ são as arestas (enlaces de comunicação). Uma aresta (i, j) indica que o processo i pode se comunicar diretamente com o processo j e vice-versa. Uma árvore geradora de G é um sub-grafo $T = (V, E')$ conexo e acíclico no qual $E' \subseteq E$, isto é, T contém todos os vértices de G e $|E'| = |V| - 1$. Se as arestas possuem pesos, uma *árvore geradora mínima* é aquela cuja soma dos pesos das arestas é mínima. Se cada aresta possui um peso diferente, existe uma única árvore mínima. Se todas as arestas possuem o mesmo peso, todas as árvores do grafo são mínimas (Gallager et al., 1983). Neste trabalho todos os enlaces possuem o mesmo peso.

A possibilidade de ocorrência de falhas é intrínseca aos sistemas distribuídos. Uma aplicação distribuída tolerante à falhas deve continuar sua execução corretamente na presença de falhas, de preferência, sem um comprometimento do desempenho. Idealmente a adaptação do serviço deve ocorrer de forma transparente, como acontece nos sistemas denominados autonômicos (Kephart e Chess, 2003). Para os algoritmos de árvores geradoras distribuídas, além do problema de construção, existe ainda o custo de manutenção das árvores quando ocorrem falhas dos seus nodos, o que implica na sua reconstrução ou reconfiguração após uma falha.

4.2 Trabalhos Relacionados

Os dois algoritmos clássicos para a obtenção de árvores geradoras mínimas a partir de um grafo são o algoritmo de Kruskal Jr. (1956) e o proposto por Prim (1957). O algoritmo de Kruskal inicialmente cria uma floresta na qual cada vértice é uma árvore. A cada passo, as árvores são conectadas entre si através das arestas de menor peso. As arestas que não interligam duas árvores são descartadas, evitando ciclos. Ao final, uma única componente conexa é gerada e esta constitui a árvore geradora mínima do grafo. O algoritmo de Prim utiliza uma abordagem diferente, que emprega cortes mínimos para escolher as arestas de menor peso para incluí-las na árvore.

Muitos algoritmos distribuídos para construção de árvores geradoras são baseados nos algoritmos centralizados de Kruskal e Prim. O primeiro deles foi definido por Gallager et al. (1983). O processo é semelhante ao utilizado por Kruskal. Inicialmente cada nodo é uma árvore. A cada nível, um nodo é eleito líder e uma aresta de peso mínimo que o interliga a um nodo em outra árvore é adicionada. O processo é repetido até formar uma única componente conexa. O algoritmo proposto por Dalal (1987) utiliza o modelo de Prim para conectar segmentos da árvore escolhendo a aresta de menor peso que conecta dois segmentos.

Avresky (1999) apresenta três algoritmos para construção e manutenção de árvores em sistemas baseados em hipercubos sujeitos a falhas. Na fase inicial a árvore é construída usando busca em largura. Em caso de falha, o processo falho é desconectado da árvore e uma nova árvore é reconstruída pela conexão dos filhos do processo falho. Os algoritmos toleram falhas simples de processo e enlace, mas podem bloquear em certas combinações com falhas múltiplas.

Leitão et al. (2007) propõem um protocolo híbrido combinando árvore com uma estratégia de *gossip*. Nesta solução, chamada HyParView, uma árvore de *broadcast* é criada sobre uma rede de sobreposição baseada em *gossip*. Outras soluções utilizando inundação são empregas para criar protocolos probabilísticos. Eugster et al. (2003) propõem um mecanismo no qual cada processo tem conhecimento de uma quantidade fixa de vizinhos escolhidas aleatoriamente. Pereira et al. (2004) utilizam um protocolo hierárquico que se

adapta de acordo com a capacidade dos nodos em disseminar mensagens.

O trabalho de Flocchini et al. (2012) propõe uma solução tolerante a falhas para árvores geradoras que reconstrói a árvore após uma falha simples utilizando árvores alternativas pré-computadas. Isto é feito pelo cálculo e armazenamento distribuído das n árvores que podem ser geradas com um único processo falho. Em caso de recuperação do processo, a árvore anterior é restaurada para incluí-lo novamente na topologia.

4.3 A Topologia Virtual VCube

O hipercubo virtual utilizado neste trabalho, denominada VCube, é criado e mantido com base nas informações de diagnóstico obtidas por meio de um sistema de monitoramento de processos descrito em Ruoso (2013). Cada processo que executa o VCube é capaz de testar outros processos no sistema para verificar se estão corretos ou falhos. Para isso, o processo executa um procedimento de teste e aguarda por uma resposta. Um processo é considerado correto ou sem-falha se a resposta ao teste for recebida corretamente dentro do intervalo de tempo esperado. Os processos são organizados em clusters progressivamente maiores. Cada cluster $s = 1, .., \log_2 n$ possui 2^{s-1} elementos, sendo n o total de processos no sistema. Os testes são executados em rodadas. Para cada rodada um processo i testa o primeiro processo sem-falha j na lista de processos de cada cluster s e obtém informação sobre os processos naquele cluster.

Os membros de cada cluster s e a ordem na qual eles são testados por um processo i são dados pela lista $c_{i,s}$, definida a seguir. O símbolo $\oplus$ representa a operação binária de OU exclusivo (XOR):

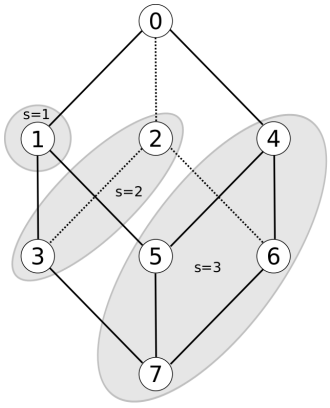
$$c_{i,s} = (i \oplus 2^{s-1}, c_{i\oplus 2^{s-1},1}, ..., c_{i\oplus 2^{s-1},s-1}) \tag{4.1}$$

Como exemplo, o cluster 3 do processo 0 é dado por:

$$\begin{aligned} c_{0,3} &= (0 \oplus 2^2, c_{0\oplus 2^2,1}, c_{0\oplus 2^2,2}) = (4, c_{4,1}, c_{4,2}) \\ c_{4,1} &= (4 \oplus 2^0) = (5) \\ c_{4,2} &= (4 \oplus 2^1, c_{4\oplus 2^1,1}) = (6, c_{6,1}) \\ c_{6,1} &= (6 \oplus 2^0) = (7) \\ c_{0,3} &= (4, 5, 6, 7) \end{aligned}$$

A Figura 4.1 exemplifica a organização hierárquica dos processos em um hipercubo de três dimensões com $n = 8$ elementos, bem como os clusters visualizados a partir do processo p_0. A tabela da direita apresenta os elementos de cada cluster $c_{i,s}$. Como exemplo, na primeira rodada o processo p_0 testa o primeiro processo no cluster $c_{0,1} = (1)$ e obtém informações sobre o estado do processo p_1. Em seguida, p_0 testa o processo p_2, que

é o primeiro processo no cluster $c_{0,2} = (2, 3)$, e obtém informações sobre p_2 e p_3. Por fim, p_0 executa testes no processo p_4 do cluster $c_{0,3} = (4, 5, 6, 7)$ e obtém informações sobre os processos p_4, p_5, p_6 e p_7.



s	$c_{0,s}$	$c_{1,s}$	$c_{2,s}$	$c_{3,s}$	$c_{4,s}$	$c_{5,s}$	$c_{6,s}$	$c_{7,s}$
1	1	0	3	2	5	4	7	6
2	2 3	3 2	0 1	1 0	6 7	7 6	4 5	5 4
3	4 5 6 7	5 4 7 6	6 7 4 5	7 6 5 4	0 1 2 3	1 0 3 2	2 3 0 1	3 2 1 0

Figura 4.1: Organização hierárquica do VCube.

De acordo com o algoritmo, o processo i testa o processo $j \in c_{i,s}$ somente se o processo i é o primeiro processo correto em $c_{j,s}$. Assim, todo processo (falho ou correto) é testado uma única vez em cada rodada. Isso garante uma latência de diagnóstico de $\log_2 n$ rodadas na média e $\log_2^2 n$ no pior caso.

4.4 O Algoritmo Autonômico para Árvores Geradoras

Esta seção apresenta o algoritmo proposto para construir de forma autonômica uma árvore geradora mínima em um sistema distribuído com base na topologia VCube.

Com base na organização lógica do VCube e na função $c_{i,s}$ foram definidas as seguintes funções. Seja i um processo que executa o algoritmo de árvore geradora e $d = \log_2 n$ a dimensão do VCube com 2^d processos. A lista de processos considerados corretos por i é armazenada em $correct_i$.

A função $cluster_i(j) = s$ calcula o identificador s do cluster do processo i que contém o processo j, $1 \leq s \leq d$. Seja $\oplus$ o operador binário de "ou exclusivo" (xor). Esta função pode ser implementada por $msb(i \oplus j) + 1$; msb é o bit mais significativo da representação binária do resultado da operação xor executada sobre os identificadores dos processos i e j. Por exemplo, considerando o 3-VCube da Figura 4.1, $cluster_0(1) = 1$, $cluster_0(2) = cluster_0(3) = 2$ e $cluster_0(4) = cluster_0(5) = cluster_0(6) = cluster_0(7) = 3$.

A função $FF\_neighbor_i(s) = j$ identifica o primeiro processo sem-falha j no cluster s do processo i, isto é, $j \in correct_i$. Por exemplo, considerando a tabela da Figura 4.1, $FF\_neighbor_0(1) = 1$, $FF\_neighbor_0(2) = 2$ e $FF\_neighbor_0(3) = 4$. Em caso de falha do processo 2, $FF\_neighbor_0(2) = 3$.

A função $neighborhood_i(s') = \{k \mid k = FF\_neighbor_i(s), 1 \leq s \leq s'\}$ é usada pelo

processo i para identificar o conjunto dos vizinhos sem-falha identificados individualmente por $k = FF\_neighbor_i(s)$ em relação aos clusters $1..s'$. Por exemplo, considerando o VCube de 3 dimensões da Figura 4.1, $neighborhood_0(3) = \{1, 2, 4\}$, $neighborhood_1(1) = \emptyset$, $neighborhood_2(2) = \{3\}$ e $neighborhood_4(3) = \{5, 6\}$. Se o processo 4 é o único processo falho, $neighborhood_0(3) = \{1, 2, 5\}$.

O Algoritmo 4.1 apresenta o pseudocódigo da solução.

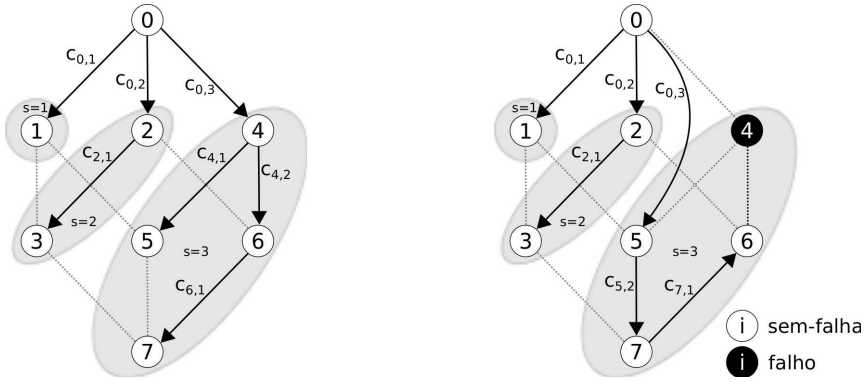
Algoritmo 4.1 Algoritmo Distribuído de Árvore Geradora Mínima no processo i

```
1:  correct_i ← {0, .., n − 1} //lista dos processos corretos
2:  procedure StartTree( )
3:      //envia a todos os vizinhos
4:      for all k ∈ neighborhood_i(log_2 n) do
5:          Send(⟨TREE⟩) para p_k
6:  procedure Receive(⟨TREE⟩) from p_j
7:      if j ∈ correct_i then
8:          //retransmite aos vizinhos dos clusters internos
9:          for all k ∈ neighborhood_i(cluster_i(j) − 1) do
10:             send(⟨TREE⟩) para p_k
11: procedure Crash(processo j) //j é detectado falho
12:     correct_i ← correct_i ∖ {j}
13:     if k = FF_neighbor_i(cluster_i(j)), k ≠ ⊥ then
14:         send(⟨TREE⟩) to p_k
```

Considere inicialmente uma execução sem falhas. No primeiro passo, a propagação é iniciada no procedimento StartTree. Uma mensagem TREE é enviada para os $\log_2 n$ vizinhos sem falha do processo i, um em cada cluster $s = 1, .., \log_2 n$ (linhas 4-5). Ao receber uma mensagem de um processo j, o procedimento Receive é executado e o processo i encaminha a mensagem para os *clusters* internos ao seu próprio cluster, isto é, aos clusters $s' = 1, .., cluster_i(j) - 1$ (linha 9). Este segundo passo é repetido até que a mensagem alcance um nodo folha, isto é, o processo j recebeu a mensagem do processo i e $cluster_i(j) = 1$.

Como exemplo, considere o VCube sem processos falhos da Figura 4.2(a). O processo p_0 é a raiz e envia TREE para os vizinhos $FF\_neighbor_0(1) = 1$, $FF\_neighbor_0(2) = 2$ e $FF\_neighbor_0(3) = 4$. O processo p_1 recebe a mensagem, mas não retransmite, visto $cluster_1(0) = 1$ e $neighborhood_1(0) = \emptyset$. O processo p_2 recebe a mensagem e retransmite para seu vizinho p_3 no cluster $s = 1$. Quando p_3 recebe a mensagem ele calcula $cluster_3(2) = 1$ e para a retransmissão. No caso de p_4 a mensagem é recebida e retransmitida para os vizinhos $5 \in c_{4,1}$ e $6 \in c_{4,2}$. Finalmente, sendo $FF\_neighbor_6(1) = 7$, o processo p_6 envia a mensagem para o processo p_7.

Os casos com falhas podem ser divididos em dois cenários. Primeiramente, considere que um processo $j \in c_{i,s}$ está falho e o processo i já foi informado sobre esta falha pelo detector, isto é, $j \notin correct_i$ (linha 11). Neste caso, o processo i envia a mensagem para o vizinho sem falha $k = FF\_neighbor_i(s)$ e a mensagem é propagada corretamente por



(a) sem-falhas (b) processo p_4 falho

Figura 4.2: Árvores geradoras no VCube de 3 dimensões.

k a todos os clusters internos do cluster s. Em um segundo cenário, este mesmo processo j está falho, mas o processo i ainda não foi informado pelo detector, isto é, $j \in correct_i$. Neste caso, se $FF\_neighbor_i(s) = j$, a mensagem é enviada ao processo falho j e é descartada. A propagação termina prematuramente e a sub-árvore interna ao cluster s é desconectada. Entretanto, assim que o módulo detector informa i sobre a falha, um novo $FF\_neighbor_i(s) = k$ é eleito e a mensagem é retransmitida para k, reconstruindo-se assim a sub-árvore do cluster s (linha 13).

A Figura 4.2(b) ilustra um cenário com falhas. Seja o processo p_0 novamente a raiz e p_4 falho. Estando p_0 ciente da falha do processo p_4, ao invés de enviar TREE para p_4, p_0 transmite a mensagem a p_5 que é o primeiro processo sem falha de $c_{0,3}$. O processo p_5 por sua vez, repassa a mensagem para $p_7 \in c_{5,2}$. Por fim, p_7 retransmite a mensagem para $p_6 \in c_{7,1}$, completando a árvore. Se quando p_0 inicia o *broadcast* e $p_4 \in correct_i$, p_0 envia a mensagem para p_4 e, quando o detector informá-lo sobre a falha de p_4, a mensagem será retransmitida para p_5. Deste ponto em diante a propagação é análoga ao caso anterior.

O Teorema 4.1 formaliza a propagação em árvore proposta pelo Algoritmo 4.1.

Teorema 4.1. *Seja m uma mensagem propagada por um processo fonte src correto. Todo processo correto no sistema Π recebe m.*

Prova. A prova deste teorema é por indução. Considere como base da indução um sistema com $n = 2$ processos: p_0 é o processo *src* que inicia o envio da mensagem m e $p_1 \in c_{0,1}$. Se p_1 é correto, $FF\_neighbor_0(1) = 1$ e p_0 envia m para p_1 (linha 4). Portanto, p_1 recebe m e o teorema é válido.

Como hipótese da indução, considere que o teorema é válido para um sistema com $n = 2^k$ processos.

No passo da indução é demonstrado que o teorema é válido para um sistema com $n = 2^{k+1}$ processos. Pela organização hierárquica do VCube, este sistema é constituído de dois subsistemas com $n = 2^k$ processos, como ilustrado pela Figura 4.3. A figura mostra

que src e j são as raízes destes subsistemas. O processo src executa o algoritmo e envia m para cada processo retornado por $FF\_neighbor_{src}(s)$, $s = 1, .., k$ (linha 4). Sendo $j = FF\_neighbor_{src}(k)$ um processo correto, j corretamente recebe m. Se j é detectado como falho, uma cópia da mensagem é retransmitida para o próximo processo correto no mesmo cluster de j (linha 13). Assim, a mensagem m é transmitida nos dois subsistemas e, pela hipótese, todo processo correto recebe m em cada subsistema. Como todo processo em Π pertence a um destes subsistemas, todo processo correto em Π recebe m. ■

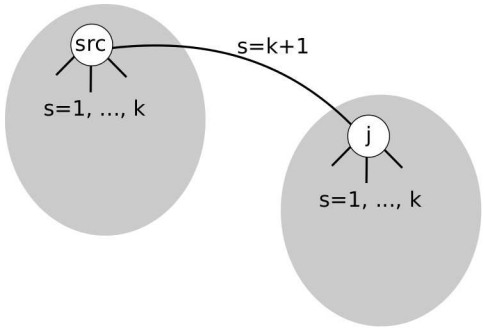


Figura 4.3: Mecanismo de propagação das mensagens na árvore geradora

.

4.5 Considerações Finais

Este capítulo apresentou a abordagem distribuída e autonômica para criação e manutenção de árvores geradoras em um sistema distribuído no qual os processes são organizados em um hipercubo virtual denominado VCube. Os resultados estão publicados em Rodrigues, Duarte Jr. e Arantes (2014). O algoritmo de árvore geradora constitui um bloco básico para a implementação das soluções de *broadcast* do Capítulo 5 que, por sua vez, é utilizado pelo algoritmo de k-exclusão mútua apresentado no Capítulo 6. Resultados de testes de desempenho demonstrando a eficiência da utilização da árvore em comparação com outras soluções de disseminação são apresentados no contexto de difusão de mensagens do Capítulo 5 e na solução final de exclusão mútua do Capítulo 6.

CAPÍTULO 5

ALGORITMOS AUTONÔMICOS PARA DIFUSÃO DE MENSAGENS

Difusão de mensagens (*broadcast*) é um componente básico para implementar muitos algoritmos e serviços distribuídos como notificação, entrega de conteúdo, replicação e comunicação em grupo (Leitão et al., 2007; Yang et al., 2009; Bonomi et al., 2013).

Neste capítulo, dois algoritmos de difusão tolerante a falhas são descritos. O primeiro implementa difusão de melhor-esforço (*best-effort broadcast*), que garante a entrega das mensagens quando o emissor é correto; e o segundo propõe uma solução de difusão confiável (*reliable broadcast*), que garante a entrega das mensagens mesmo se o emissor falhar antes de completar a difusão. As duas soluções utilizam o modelo de árvores geradoras proposto no Capítulo 4, incluindo a detecção de falhas perfeita implementada pelo VCube, descrito na Seção 4.3 daquele mesmo capítulo.

5.1 Definição do Problema

Um processo em um sistema distribuído utiliza difusão para enviar uma mensagem a todos os outros processos do sistema. No entanto, se este processo falha durante o procedimento de difusão, alguns processos podem receber a mensagem enquanto outros não. A difusão de melhor-esforço garante que, se o emissor é correto, todos os processos corretos recebem a mensagem difundida por ele. Por outro lado, se o emissor pode falhar, estratégias de difusão confiável precisam ser implementadas (Hadzilacos e Toueg, 1993).

Algoritmos de difusão tolerante a falhas são normalmente implementados utilizando enlaces ponto-a-ponto confiáveis e primitivas SEND e RECEIVE. Os processos invocam BROADCAST(m) e DELIVER(m) para difundir e receber uma mensagem m para/de outros processos da aplicação, respectivamente. Para incluir tolerância a falhas, um detector de falhas perfeito (Chandra e Toueg, 1996) pode ser utilizado para notificar o algoritmo de *broadcast*, que deve reagir apropriadamente quando uma falha é detectada.

5.2 Trabalhos Relacionados

Na implementação de difusão confiável descrita em Kaashoek et al. (1989), o protocolo utiliza um nodo central (chamado sequenciador) para ordenar as mensagens. O sequenciador recebe uma mensagem, acrescenta o número de sequência e transmite a mensagem. Como apontado pelos autores, o sequenciador pode tornar-se um gargalo e prejudicar a escalabilidade do sistema quando o número de nodos é muito grande. Além disso, se o

sequenciador falhar, o sistema todo para. Assim, mecanismos extras devem ser implementados para permitir a recuperação de falhas.

Outra implementação de difusão confiável, proposta por Garcia-Molina e Kogan (1988), utiliza um mecanismo de *multicast* não-confiável em uma rede assíncrona ponto-a-ponto. Listas de prioridade são usadas para especificar a ordem em que nodos devem acessar a rede depois de uma falha e as listas são baseadas em informações sobre a topologia da rede.

Mais recentemente, Bonomi et al. (2013) propuseram uma estratégia de difusão confiável que tolera falhas arbitrárias. Para tratar as falhas maliciosas, o autores utilizam um detector de falhas bizantino.

Outros trabalhos descrevem soluções em redes sem-fio. Pagani e Rossi (1997) propuseram um protocolo de difusão para este tipo de rede que tolera falhas de comunicação e mobilidade de nodos. Partições de rede são toleradas se forem temporárias. Em Yang et al. (2009) os autores propuseram o R-Code, um protocolo de difusão confiável que constrói uma árvore de difusão no *backbone* da rede, usando como peso de cada enlace o número de transmissões possíveis neles.

Vários algoritmos de difusão são baseados em árvores geradoras (Fragopoulou e Akl, 1996; Liu et al., 2009; Wang et al., 2012). Schneider, Gries e Schlichting introduziram um algoritmo de difusão tolerante a falhas baseado em árvore no qual a raiz é o processo que inicia a transmissão, ou seja, o remetente. Cada nodo, incluindo o remetente, envia a mensagem para todos os seus sucessores na árvore. Se um processo p que pertence à árvore falhar, outro processo assume a responsabilidade de retransmitir as mensagens que p deveria ter transmitido se estivesse correto. Os processos podem falhar por *crash* e a falha de um processo é detectada por um módulo de detecção de falhas após um intervalo finito, mas não conhecido. Um processo pode enviar uma próxima mensagem somente após a difusão anterior ter sido concluída. No entanto, os autores não descrevem como a detecção de falhas é implementada, tampouco fornecem um algoritmo para construir e reorganizar a árvore após a falha.

Kim et al. (2010) propuseram uma solução baseada em árvore para difundir uma mensagem para um grande número de receptores que usam caminhos múltiplos. Ramanathan e Shin (1988) propuseram uma difusão confiável que é executada em um hipercubo e usa árvores geradoras disjuntas para o envio de uma mensagem através de vários caminhos. Algoritmos de caminhos múltiplos são particularmente úteis em sistemas que não podem tolerar a sobrecarga de tempo para a detecção de processos com falhas, mas há uma sobrecarga no número de mensagens duplicadas.

Em Wu (1996), os autores apresentam um algoritmo de difusão tolerante a falhas para hipercubos baseado em árvores binomiais. O algoritmo pode recursivamente regenerar uma subárvore falha, induzida por um nodo com defeito, através de uma das folhas da árvore. No entanto, ao contrário da abordagem proposta neste trabalho, a solução exige

uma mensagem especial para indicar que a árvore deverá ser reorganizada e, neste caso, as mensagens de difusão não são tratados pelos nodos até que a árvore seja reconstruída.

Liebeherr e Beam (1999) apresentam um procolo, chamado HyperCast, que organiza os membros de um grupo *multicast* em um hipercubo lógico usando o código *gray* para ordená-los. A árvore é sobreposta no hipercubo para evitar implosão de ACKs. O processo com o identificar mais alto é considerado a raiz da árvore. Entretanto, em função de falhas, múltiplos nodos podem considerar a si próprios como raiz e/ou diferentes nodos podem ter visões diferentes sobre a identidade da raiz.

Um protocolo híbrido combinando estratégias de árvore e *gossip* foi proposto por Leitão et al. (2007). Nesta solução, chamada HyParView, uma árvore de difusão é criada sobre uma rede *gossip*. Outras soluções utilizam *gossiping* para criar protocolos probabilísticos. Eugster et al. (2003) propuseram um algoritmo no qual cada processo conhece um número fixo de vizinhos escolhidos aleatoriamente. Pereira et al. (2004) propuseram um protocolo epidêmico no qual as mensagens são envidas usando os nodos com maior capacidade de transmissão, em uma tentativa de otimizar a taxa de disseminação.

5.3 Difusão de Melhor-esforço

A difusão de melhor-esforço garante que todos os processos corretos entregam o mesmo conjunto de mensagens se o emissor (fonte) é correto. Três propriedades caracterizam este modelo: entrega confiável (*validity*), não-duplicação e não-criação de mensagens (Guerraoui e Rodrigues, 2006). A entrega confiável garante que, se um processo i envia uma mensagem m para um processo j e nenhum deles falha, j recebe m em um tempo finito. A não-duplicação garante que nenhuma mensagem é entregue mais de uma vez e a não-criação garante que nenhuma mensagem é entregue a menos que tenha sido previamente enviada. Note que, em função da falha do processo fonte, alguns processos podem receber a mensagem enquanto outros não, o que não invalida as propriedades.

O Algoritmo 5.1 apresenta uma solução para difusão de melhor-esforço que utiliza o mecanismo de árvore proposto no Capítulo 4. Dois tipos de mensagens são utilizados: $\langle TREE, m\rangle$ para identificar a mensagem de aplicação m que está sendo propagada e $\langle ACK, m\rangle$ para confirmar o recebimento de m pelo destinatário. Cada mensagem m contém ainda dois parâmetros: (1) o identificador da origem, isto é, o processo que iniciou a difusão, obtido com a função $source(m)$; e (2) o *timestamp*, um contador sequencial local que identifica de forma única cada mensagem gerada em um processo, obtido pela função $ts(m)$. Um processo obtém as informações sobre o estado dos demais processos pelo algoritmo VCube. O algoritmo executa corretamente mesmo que $n-1$ falhas ocorram. Um versão preliminar deste algoritmo foi publicada em Rodrigues (2013) aplicado a uma solução tolerante a falhas de exclusão mútua distribuída. O algoritmo descrito nesta seção foi publicado em Rodrigues, Duarte Jr. e Arantes (2014) .

As variáveis locais mantidas pelos processos são:

- $correct_i$: conjunto dos processos considerados corretos pelo processo i;
- $last_i[n]$: a última mensagem recebida de cada processo fonte;
- ack_set_i: o conjunto com todos os ACKs pendentes no processo i. Para cada mensagem $\langle TREE, m\rangle$ recebida pelo processo i de um processo j e retransmitida para o processo k, um elemento $\langle j, k, m\rangle$ é adicionado a este conjunto.

O símbolo $\perp$ representa um elemento nulo. O asterisco é usado como curinga para selecionar ACKs no conjunto ack_set. Um elemento $\langle j, *, m\rangle$, por exemplo, representa todos os ACKs pendentes para uma mensagem m recebida de um processo j e retransmitida para qualquer outro processo.

Um processo i que deseja difundir uma mensagem m por difusão invoca o método BROADCAST. A linha 5 garante que um novo *broadcast* só é iniciado após o término do anterior, isto é, quando não há mais ACKs pendentes para a mensagem $last_i[i]$. Nas linhas 9-11 a nova mensagem é enviada a todos os vizinhos considerados corretos. Para cada mensagem enviada, um ACK é incluído na lista de ACKs pendentes.

Quando um processo recebe uma mensagem TREE de um processo j (linha 16), ele primeiramente verifica se tanto o processo fonte da mensagem quanto o processo j são considerados corretos. Se um deles está falho, o recebimento é abortado, pois se j está falho, o processo que transmitiu m para j fará uma nova transmissão quando detectar a falha e i irá receber a mensagem através da nova árvore que será reconstruída. Além disso, se o fonte está falho, não é mais necessário continuar a retransmissão. Se o fonte e j estão corretos, o processo i verifica se a mensagem é nova comparando os *timestamps* da última mensagem armazenada em $last_i[j]$ e da mensagem recebida m (linha 21). Se m é nova, $last_i[j]$ é atualizado e a mensagem é entregue à aplicação. Em seguida, m é retransmitida para os vizinhos em cada cluster interno ao cluster de i. Se não existe vizinho correto ou se i é uma folha na árvore ($cluster_i(j) = 1$), nenhum ACK pendente é adicionado ao conjunto ack_set_i e CHECKACKS envia um ACK para j.

Se uma mensagem $\langle ACK, m\rangle$ é recebida, o conjunto ack_set_i é atualizado e, se não existem mais ACKs pendentes para a mensagem m, CHECKACKS envia um $\langle ACK, m\rangle$ para o processo k do qual i recebeu a mensagem TREE anteriormente. No entanto, se $k = \perp$, o ACK alcançou o processo fonte e não precisa mais ser propagado.

A detecção de um processo falho j é tratada no procedimento CRASH. Três ações são realizadas: (1) atualização da lista de processos corretos; (2) remoção dos ACKs pendentes que contém o processo j como destino ou aqueles em que a mensagem m foi originada em j; (3) reenvio das mensagens anteriormente transmitidas ao j para o novo vizinho k no mesmo cluster de j, se existir um. Esta retransmissão desencadeia uma propagação na nova estrutura da árvore.

Algoritmo 5.1 Difusão de melhor-esforço hierárquico no processo i

1: $last_i[n] \leftarrow \{\perp, .., \perp\}$
2: $ack\_set_i = \emptyset$
3: $correct_i = \{0, .., n-1\}$

4: **procedure** BROADCAST(message m)
5: **wait until** $ack\_set_i \cap \{\langle \perp, *, last_i[i] \rangle\} = \emptyset$
6: $last_i[i] = m$
7: DELIVER(m)
8: //envia a todos os vizinhos
9: **for all** $j \in neighborhood_i(\log_2 n)$ **do**
10: $ack\_set_i \leftarrow ack\_set_i \cup \{\langle \perp, j, m \rangle\}$
11: SEND($\langle TREE, m \rangle$) **to** p_j

12: **procedure** CHECKACKS(processo j, mensagem m)
13: **if** $ack\_set_i \cap \{\langle j, *, m \rangle\} = \emptyset$ **then**
14: **if** $\{source(m), j\} \subseteq correct_i$ **then**
15: SEND($\langle ACK, m \rangle$) **to** p_j

16: **procedure** RECEIVE($\langle TREE, m \rangle$) **from** p_j
17: **if** $\{source(m), j\} \nsubseteq correct_i$ **then**
18: **return**
19: //verifica se m é nova
20: **if** $last_i[source(m)] = \perp$ **or**
21: $ts(m) = ts(last_i[source(m)]) + 1$ **then**
22: $last_i[source(m)] \leftarrow m$
23: DELIVER(m)
24: //retransmite aos vizinhos dos clustes internos
25: **for all** $k \in neighborhood_i(cluster_i(j) - 1)$ **do**
26: **if** $\langle j, k, m \rangle \notin ack\_set_i$ **then**
27: $ack\_set_i \leftarrow ack\_set_i \cup \{\langle j, k, m \rangle\}$
28: SEND($\langle TREE, m \rangle$) **to** p_k
29: CHECKACKS(j, m)

30: **procedure** RECEIVE($\langle ACK, m \rangle$) **from** p_j
31: $k \leftarrow x : \langle x, j, m \rangle \in ack\_set_i$
32: $ack\_set_i \leftarrow ack\_set_i \setminus \{\langle k, j, m \rangle\}$
33: **if** $k \neq \perp$ **then**
34: CHECKACKS(k, m)

35: **procedure** CRASH(processo j) //j é detectado falho
36: $correct_i \leftarrow correct_i \setminus \{j\}$
37: $k \leftarrow FF\_neighbor_i(cluster_i(j))$
38: **for all** $p = x, q = y, m = z : \langle x, y, z \rangle \in ack\_set_i$ **do**
39: **if** $\{source(m), p\} \nsubseteq correct_i$ **then**
40: //remove ACKs pendentes para $\langle j, *, * \rangle$ e
41: $//\langle *, *, m \rangle : source(m) = j$
42: $ack\_set_i \leftarrow ack\_set_i \setminus \{\langle p, q, m \rangle\}$
43: **else if** $q = j$ **then** //envia m para vizinho k
44: **if** $k \neq \perp$ **and** $\langle p, k, m \rangle \notin ack\_set_i$ **then**
45: $ack\_set_i \leftarrow ack\_set_i \cup \{\langle p, k, m \rangle\}$
46: SEND($\langle TREE, m \rangle$) **to** p_k
47: $ack\_set_i \leftarrow ack\_set_i \setminus \{\langle p, j, m \rangle\}$
48: CHECKACKS(p, m)

5.3.1 Prova Formal

O correto funcionamento do Algoritmo 5.1 como uma solução de difusão de melhor-esforço (Teorema 5.1) é garantido pelas propriedades de entrega confiável (Lema 5.1) e integridade (não-duplicação e não-criação) (Lema 5.2).

Lema 5.1 (Entrega confiável)**.** *Se um processo correto i efetua a difusão de uma mensagem m, então ele também entrega m em um intervalo de tempo finito.*

Prova. Considere a função BROADCAST(m) do Algoritmo 5.1. O processo i aguarda até receber todos as mensagens de confirmação (ACKs) relacionadas à última mensagem enviada por ele, armazenada em $last_i[i]$ (linha 5). O algoritmo garante que, ou todos os ACKs dos processos corretos são recebidos por i (linha 30) ou o processo j que não enviou ACK está falho e i, após detectar a falha de j, retransmite a mensagem ao próximo processo sem-falha k no mesmo cluster de j (se existir algum) e remove o ACK da lista de pendentes (linha 35). Assim, em um tempo finito, não haverá mais ACKs pendentes; o valor de $last_i[i]$ é atualizado e i entrega m para a aplicação (linhas 5-7). ■

Lema 5.2 (Integridade)**.** *Todo processo correto i entrega uma mensagem m no máximo uma vez (não-duplicação) e somente se m foi previamente enviada por broadcast por algum processo (não-criação).*

Prova. A entrega da mensagem m pelo processo fonte é garantida pelo Lema 5.1. O Teorema 4.1 garante que todo processo correto j recebe a mensagem m propagada por i através da árvore geradora. Ao receber m, j verifica se m é uma nova mensagem através do seu *timestamp* (linha 21) e, em caso positivo, entrega m. Assim, mesmo que uma mensagem seja retransmitida após a detecção de um falha e alcance um processo que já a recebeu, o receptor nunca fará a entrega duplicada. A propriedade de não-criação é garantida pela característica dos enlaces confiáveis. ■

Teorema 5.1. *O Algoritmo 5.1 é uma solução para difusão de melhor-esforço: se o emissor é correto, todo processo correto receberá o mesmo conjunto de mensagens.*

Prova. O Teorema é validado pelas propriedades da difusão de melhor-esforço: entrega confiável (Lema 5.1) e Integridade (Lema 5.2). ■

Teorema 5.2. *O total de mensagens enviadas para cada mensagem de aplicação em uma execução sem falhas do Algoritmo 5.1 é $2 * (n - 1)$. Se um processo j falha antes de confirmar o recebimento de uma mensagem TREE recebida de um processo i, o total de mensagens extras depende da quantidade de processos no cluster de j em relação ao processo i.*

Prova. Em uma execução sem falhas, para cada mensagem TREE enviada, uma mensagem ACK é retornada. Seja $n-1$ o número de arestas na árvore com n processos, o total de mensagens é o dobro do total de arestas.

Se um processo j é detectado como falho por um processo i depois que a mensagem TREE foi enviada para j, uma nova mensagem será enviada para o próximo vizinho $k = FF\_neighbor_i(cluster_i(j))$. Seja $s = cluster_i(j)$. No melhor caso, $k = \perp$ e nenhuma mensagem extra é enviada ($j \in c_{i,1}$ ou não existem mais processos corretos no cluster s). No entanto, se $k \neq \perp$, a quantidade de mensagens extras depende do número de processos detectados como corretos no cluster s. Seja $n' = |c_{i,s}|$ o total de processos no cluster $c_{i,s}$. No pior caso, todos os processos do cluster estão corretos, exceto j, e j enviou a mensagem a todos os vizinhos antes de falhar. Assim, o total de mensagens extras será $1+2*(n'-2)$, uma TREE extra para k e $(n'-2)$ TREEs + $(n'-2)$ ACKs na sub-árvore. De forma geral, se existem f processos falhos em $c_{i,s}$ incluindo j, a quantidade de mensagens extras retransmitidas é $1 + 2 * (n' - 1 - f)$. ■

5.4 Difusão Confiável

Um algoritmo de difusão confiável (*reliable broadcast*) garante que o mesmo conjunto de mensagens é entregue a todos os processos corretos, mesmo se o processo emissor (fonte) falhar durante o procedimento de difusão. Para tanto, a difusão confiável herda as propriedades de entrega confiável, não-criação e não-duplicação do melhor-esforço e acrescenta a propriedade de acordo (*agreement*). Assim, a solução para difusão confiável proposta neste trabalho é uma modificação da difusão de melhor-esforço descrito na Seção 5.3 que inclui o tratamento para a falha do processo fonte. O Algoritmo 5.2 apresenta as modificações realizadas. As variáveis $last_i$, ack_set e $correct_i$ são as mesmas do Algoritmo 5.1, bem como os métodos RECEIVE(ACK, m) e CHECKACKS.

Um processo p_i que deseja efetuar a difusão de uma mensagem m invoca o procedimento BROADCAST(m). Se m tem origem no mesmo processo, $source(m) = i$, então o tratamento é o mesmo realizado pelo algoritmo de melhor-esforço e a mensagem é então propagada para todos os vizinhos de p_i. Quando um processo p_i recebe a mensagem $\langle TREE, m\rangle$ de um processo p_j ele primeiramente verifica se $j \notin correct_i$. Se p_j está falho, a mensagem é descartada. Note que esta verificação difere do algoritmo de melhor-esforço, que não descarta mensagens recebidas de um $source(m)$ falho. A segunda modificação está nas linhas 20-22. Se p_i recebe uma mensagem nova de um processo fonte considerado falho, ele inicia um novo *broadcast* com a mensagem recebida para garantir que os demais processos recebam m corretamente. Neste caso, quando a linha 2 é executada, $source(m)$ não será igual a i e a mensagem é retransmitida aos demais processos através da árvore geradora de p_j.

A recuperação em caso de falhas é semelhante à solução de melhor-esforço, exceto pelo

Algoritmo 5.2 Difusão confiável hierárquica no processo i

```
1:  procedure BROADCAST(message m)
2:      if source(m) = i then
3:          wait until ack_set_i ∩ {⟨⊥, ∗, last_i[i]⟩} = ∅
4:          last_i[i] = m
5:          DELIVER(m)
6:      //envia a todos os vizinhos
7:      for all j ∈ neighborhood_i(log_2 n) do
8:          ack_set_i ← ack_set_i ∪ {⟨⊥, j, m⟩}
9:          SEND(⟨TREE, m⟩) to p_j

10: procedure CHECKACKS(processo j, mensagem m)
11:     //o mesmo do Algoritmo 5.1

12: procedure RECEIVE(⟨TREE, m⟩) from p_j
13:     if j ∉ correct_i then
14:         return
15:     //verifica se m é nova
16:     if last_i[source(m)] = ⊥ or
17:      ts(m) = ts(last_i[source(m)]) + 1 then
18:         last_i[source(m)] ← m
19:         DELIVER(m)
20:         if source(m) ∉ correct_i then
21:             BROADCAST(m)
22:             return
23:     //retransmite aos vizinhos dos clustes internos
24:     for all k ∈ neighborhood_i(cluster_i(j) − 1) do
25:         if ⟨j, k, m⟩ ∉ ack_set_i then
26:             ack_set_i ← ack_set_i ∪ {⟨j, k, m⟩}
27:             SEND(⟨TREE, m⟩) to p_k
28:     CHECKACKS(j, m)

29: procedure RECEIVE(⟨ACK, m⟩) from p_j
30:     //o mesmo do Algoritmo 5.1

31: procedure CRASH(processo j) //j é detectado falho
32:     correct_i ← correct_i ∖ {j}
33:     k ← FF_neighbor_i(cluster_i(j))
34:     for all p = x, q = y, m = z : ⟨x, y, z⟩ ∈ ack_set_i do
35:         if p ∉ correct_i then
36:             //remove ACKs pendentes para ⟨j, ∗, ∗⟩
37:             ack_set_i ← ack_set_i ∖ {⟨p, q, m⟩}
38:         else if q = j then //envia m para vizinho k
39:             if k ≠ ⊥ and ⟨p, k, m⟩ ∉ ack_set_i then
40:                 ack_set_i ← ack_set_i ∪ {⟨p, k, m⟩}
41:                 SEND(⟨TREE, m⟩) to p_k
42:             ack_set_i ← ack_set_i ∖ {⟨p, j, m⟩}
43:             CHECKACKS(p, m)
44:     if last_i[j] ≠ ⊥ then
45:         BROADCAST(last_i[j])
```

broadcast da última mensagem recebida do processo j detectado como falho (linha 45). Esta retransmissão, em conjunto com aquela da linha 21, garante que todos os demais processos corretos receberão a última mensagem transmitida pelo processo j falho, mesmo que um único processo correto tenha recebido a mensagem antes de j falhar.

5.4.1 Prova Formal

Nesta seção é demonstrado formalmente que o Algoritmo 5.2 implementa uma solução de difusão confiável (Teorema 5.4), garantindo entrega confiável, integridade e acordo (*agreement*). Sendo uma derivação do algoritmo de melhor-esforço da Seção 5.3 e, estando as propriedades de entrega confiável e integridade já provadas na ocasião, será apresentada apenas a validação da propriedade de acordo, no Lema 5.3 a seguir.

Lema 5.3 (Acordo)**.** *Se um processo fonte i envia uma mensagem m para ao menos um processo correto $j \in neighborhood_i(\log_2 n)$ e j entrega m, então todo processo correto receberá e entregará m, mesmo que i venha a falhar antes de terminar o procedimento de broadcast.*

Prova. Se um processo fonte é correto, o Teorema 4.1 em conjunto com as propriedades de integridade (Lema 5.2) e entrega confiável (Lema 5.1) garante que m será entregue por todos os processos corretos.

Se o processo fonte i falha antes de enviar a mensagem m para algum processo correto $j \in neighborhood_i(\log_2 n)$, nenhum processo correto receberá m e a propriedade de acordo está mantida. Por outro lado, se ao menos um processo correto j recebe m, este será notificado sobre a falha do processo fonte i. Neste caso, o acordo é garantido por duas ações: (1) difusão durante a recepção da última mensagem recebida de i (neste caso uma nova mensagem) se i já foi detectado como falho (linha 21); (2) difusão da última mensagem de um processo detectado como falho e removido de $correct_i$ (linha 45). Nos dois casos, o processo que recebe a mensagem m assume a responsabilidade de garantir a entrega de m para todos os demais processos corretos. No caso da segunda ação, todos os processos executam a difusão após a detecção da falha de j, visto que não se sabe qual deles possui a mensagem mais atual. ■

Teorema 5.4. *O Algoritmo 5.2 é uma solução para difusão confiável. Ele garante as propriedades de "entrega confiável", "integridade" e "acordo".*

Prova. As propriedades de entrega confiável, integridade e acordo são garantidas pelo Lema 5.1, Lema 5.2 e Lema 5.3, respectivamente. ■

5.5 Avaliação Experimental

Por ser utilizado pelo algoritmo de k-exclusão mútua proposto no Capítulo 6, o algoritmo de difusão de melhor-esforço foi implementado implementado utilizando o Neko (Urbán et al., 2002). O Neko é uma ferramenta Java desenvolvida para a simulação e emulação de algoritmos distribuídos baseada em microprotocolos e troca de mensagens. Cada protocolo

é instanciado em um processo (que pode representar um *nodo*) e utiliza uma rede real ou simulada para se comunicar com protocolos em outros processos.

O algoritmo hierárquico de monitoramento VCube também foi implementado utilizando o Neko. As falhas de processo foram geradas com o mecanismo de *crash* proposto em Rodrigues e Jansh-Pôrto (2008).

Além da solução de difusão de melhor-esforço proposta, nomeada ABB (*Autonomic Best-effort Broadcast*), outros dois algoritmos foram implementados e utilizados para comparação. O primeiro utiliza um modelo um-para-todos (ALL - *one-to-all*), no qual a mensagem é enviada através de enlaces ponto-a-ponto. A segunda solução implementa um modelo de árvore não-autonômica (NABB - *Non-Autonomic Best-effort Broadcast*), que constrói uma árvore a partir de um processo fonte e reconstrói uma nova árvore sempre que uma falha é detectada.

Para implementar a estratégia ALL, a função *neighborhood* foi modificada para incluir todos os processos sem-falha, fazendo com que o emissor envie a mensagem a todos os demais processos diretamente utilizando enlaces ponto-a-ponto.

A estratégia NABB constrói uma árvore utilizando inundação. A árvore é criada utilizando a própria mensagem TREE de *broadcast*. Cada processo que recebe a mensagem pela primeira vez se junta à árvore e retransmite a mensagem aos demais processos do sistema, exceto àquele do qual ele recebeu a mensagem. Se um processo já está na árvore e recebe uma outra cópia da mensagem ele envia uma mensagem de NACK e não retransmite mais a mensagem. Uma vez criada a árvore, as demais mensagens são enviadas somente pelas arestas da árvore. Em caso de falha, uma nova árvore é criada a partir do processo fonte utilizando novamente o procedimento de inundação descrito.

5.5.1 Parâmetros de Simulação

Os parâmetros de simulação são baseados em Bulgannawar e Vaidya (1995). Cada mensagem trocada entre dois processos consome $t_s + t_t + t_r$ unidades de tempo: t_s unidades para ser enviada e t_r para ser recebida. O tempo de transmissão utilizado pela rede é representado por t_t. Considera-se que não existem mecanismos auxiliares de *broadcast* e *multicast*. Portanto, quando um processo precisa enviar uma mensagem para mais de um destinatário, primitivas SEND são invocadas sequencialmente. Assim, para toda mensagem enviada a mais de um destinatário, t_s é computado novamente para cada cópia.

Para avaliar o desempenho de soluções de difusão, duas métricas são frequentemente utilizadas (Boichat e Guerraoui, 2005): (1) vazão (*throughput*), dada pelo total de mensagens de *broadcast* completos durante um intervalo de tempo; (2) a latência para entregar a mensagem de *broadcast* a todos os processos corretos.

Os algoritmos propostos foram avaliados em diferentes cenários variando o número de processos e a quantidade de processos falhos. Os parâmetros de comunicação foram

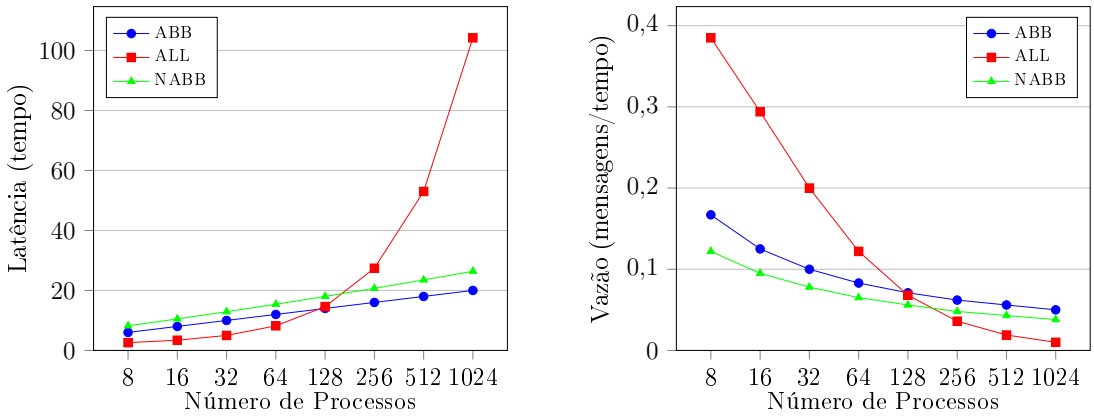
definidos em $t_s = t_r = 0,1$ e $t_t = 0,8$. O intervalo de testes do detector foi definido em $5,0$ unidades de tempo. Um processo é considerado falho se não responder ao teste após $4 * (t_s + t_r + t_t)$ unidades de tempo.

5.5.2 Resultado dos Experimentos

Os experimentos foram realizados em duas etapas. Inicialmente, foram utilizados cenários sem falhas com sistemas de 8 a 1024 processos. Além disso, a latência e a quantidade de mensagens enviadas por cada aplicação foram analisadas pontualmente em um sistema com 512 processos. Em seguida, cenários com falhas foram gerados aleatoriamente para um sistema com 512 processos contendo de 1 a 8 processos falhos.

5.5.2.1 Experimentos em Cenários sem Falhas

Nos cenários sem-falhas, como não há retransmissões, os algoritmos de difusão de melhor-esforço e confiável propostos possuem o mesmo comportamento. A Figura. 5.1 mostra a latência e a vazão considerando que uma única mensagem é enviada pelo processo p_0. O caminho mais longo em um VCube com n processos é $\log_2 n$. Portanto, quando n é pequeno, o tempo para enviar a mensagem pelo caminho mais longo é maior que o tempo para enviar as $n-1$ mensagens sequencialmente pela estratégia ALL. Considerando que o tempo de envio de cada mensagem é $t_s = 0,1$, o intervalo entre o envio da mensagem TREE e a recepção do ACK correspondente pelo caminho mais longo da árvore é $2 \log_2 n(t_s + t_r + t_t)$. Já na estratégia ALL, para enviar $n-1$ mensagens são utilizadas $(n-2)t_s + t_t + tr$ unidades de tempo.



(a) Latência (b) Vazão

Figura 5.1: Broadcast de melhor-esforço em uma execução sem-falhas.

Assim, embora ALL seja mais eficiente para sistemas pequenos, seu desempenho cai rapidamente em sistemas maiores. A solução com árvore não-autonômica NABB apresenta comportamento semelhante a ABB, exceto pela latência extra para aguardar as

mensagens de ACK/NACK durante a configuração da árvore. Em cenários sem-falha, o desempenho das duas soluções passa a ser o mesmo após a árvore estar completa.

A vazão foi calculada como $1/latência$, visto que apenas um processo envia uma única mensagem. Assim como a latência, na solução ALL o desempenho é melhor para sistemas pequenos, até 128 processos, mas diminui rapidamente quando n aumenta. Estes resultados indicam a escalabilidade da estratégia hierárquica proposta.

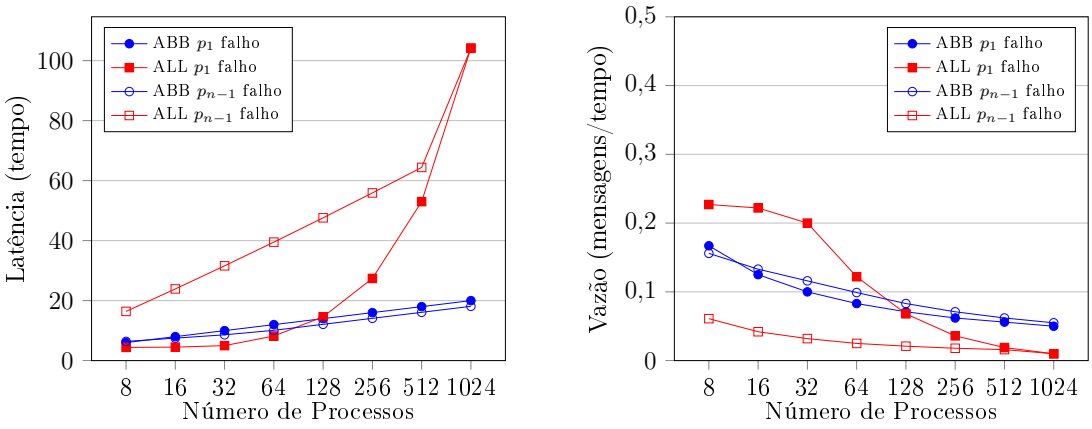
5.5.2.2 Experimentos em Cenários com Falhas

O desempenho dos três algoritmos em cenários com falhas foi comparado utilizando diferentes abordagens. Inicialmente foram comparados os algoritmos ABB e ALL em cenários com falhas de processos localizados nas folhas da árvore. Em seguida, ABB e ALL foram novamente testados em cenários com falhas em processos não-folha. Por fim, cenários com falhas aleatórias foram utilizados para comparar os três algoritmos.

Cenários com Falhas em Processos Folha. Neste experimento foi investigado qual o impacto de uma falha em processo folha no desempenho do sistema. As falhas foram configuradas para acontecer no tempo zero. Foram testados o melhor e o pior caso. O pior caso é aquele em que a folha é a mais distante do processo fonte. Em um 3-VCube sem falhas, por exemplo, quando p_0 é o fonte, o processo p_7 é a folha mais distante. Neste mesmo exemplo, o melhor caso é a falha do processo p_1, que é o processo folha mais próximo da raiz.

A Figura 5.2(a) e a Figura 5.2(b) mostram a latência e a vazão para ambos os cenários. É possível observar que em ABB, a posição da falha nas folhas não acarreta em grande impacto o desempenho, o que não é valido para ALL: a falha no pior caso (p_{n-1}) é mais danosa que a falha de p_1. A latência de ALL é dada pelo máximo entre o tempo que o VCube leva para detectar a falha e o tempo que o processo fonte consome para enviar $(n-1)$ mensagens. Lembre-se que p_0 é o fonte. Note no gráfico que para 1024 processos, ALL apresenta a mesma latência no melhor e no pior caso, definida pelo tempo gasto para enviar cada uma das $(n-1)$ mensagens. Na Figura 5.2(b) há um ponto de inflexão em $n = 16$ – quando a latência de ABB p_1 falho torna-se menor que a latência de ABB p_{n-1} falho – e em $n = 128$ quando a latência de ALL p_1 falho torna-se menor que a latência de ABB p_{n-1} falho. Estes pontos ocorrem porque, à medida que o número de processos aumenta, a latência de detecção para p_{n-1} (pior caso) predomina na latência.

Quando considera-se o pior caso, isto é, a falha do processo folha mais distante da raiz, a difusão com árvore geradora ABB é mais eficiente que a solução ALL porque a falha do processo folha é detectada enquanto a mensagem TREE ainda está sendo propagada na árvore. Neste caso, quando um processo correto na árvore recebe a mensagem e já foi notificado sobre a falha do processo folha, ele retorna o ACK imediatamente. Por outro lado, na solução ALL, o processo fonte controla o *broadcast* (p_0 neste experimento)

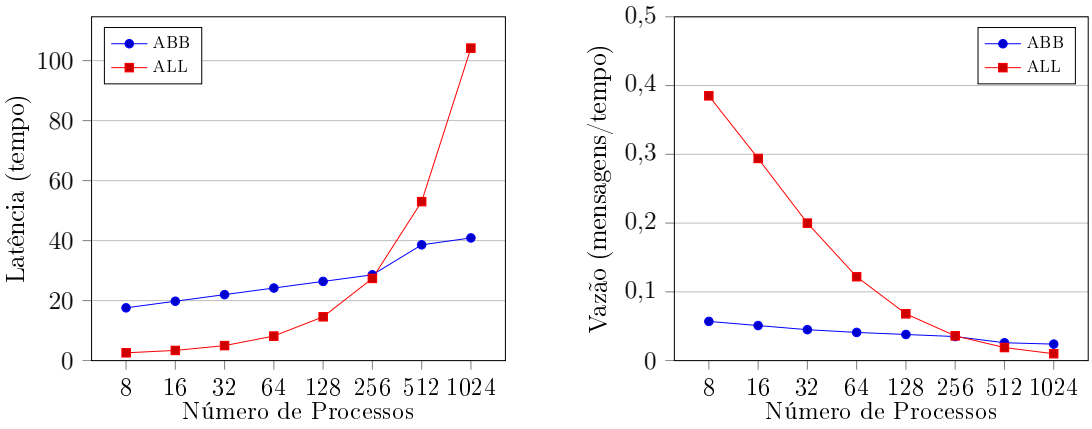


(a) Latência (b) Vazão

Figura 5.2: Execução com falha de processos folha (p_1 ou p_{n-1}).

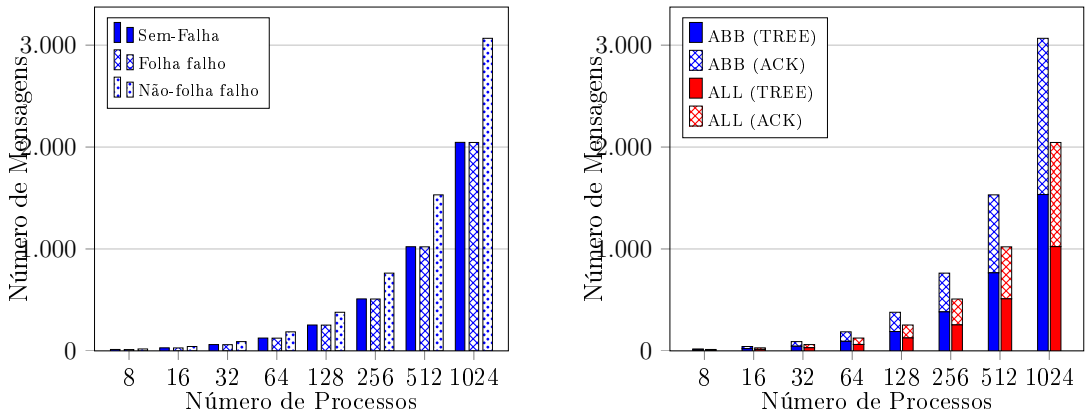
e deve aguardar ser notificado pelo VCube para remover o ACK pendente e finalizar o procedimento de difusão. Como o intervalo de testes do VCube foi configurado para 5.0, o fonte será notificado somente após ($\log_2 n * 5.0$) acrescido da latência da transmissão das mensagens de teste, o que explica a latência mais alta na solução ALL. No experimento com a falha de p_1, ALL é mais eficiente quando o sistema é menor, visto que a latência do detector é menor quando o processo está mais próximo do testador (p_0 testa p_1 rapidamente). No entanto, à medida que o número de processes aumenta, a latência para enviar $n-1$ mensagens sobrepõe a latência para detectar a falha de p_1, o que explica o crescimento da latência de difusão para ALL apresentada no gráfico.

Cenários com Falha em Processos Não-folha. Neste experimento foi simulada a estratégia autonômica ABB proposta neste trabalho considerando a falha do primeiro processo do cluster $\log_2 n$ em relação ao processo fonte p_0. Este processo corresponde a $p_{n/2}$; por exemplo, e um sistema com 8 processos, o processo p_4 está falho. A falha do processo foi configurada para acontecer no instante $\log_2 n$, garantindo que a mensagem TREE já foi transmitida a todos os filhos do processo falho. Assim, consideramos o pior caso para o número de mensagens. Como no experimento anterior, o processo p_0 envia uma única mensagem por *broadcast*. A Figura 5.3(a) e a Figura 5.3(b) mostram a latência e a vazão obtidas. Neste cenário, a abordagem ALL apresenta menor latência que ABB para sistemas com até 256 processos. Na estratégia ABB, além da latência de detecção do VCube, p_0 precisa retransmitir a mensagem após a detecção para o próximo processo sem-falha no cluster $\log_2 n$, tendo que aguardar até que a mensagem seja novamente propagada naquele cluster para então receber o ACK e terminar a difusão. Entretanto, quando o tamanho do sistema aumenta, mais uma vez o tempo despendido por ALL para enviar a mensagem a todos os $n-1$ processos sobrepõe a latência usada na propagação empregada por ABB. Assim, embora o número de mensagens seja maior, a latência ainda se mantém menor para sistemas maiores.



(a) Latência (b) Vazão

Figura 5.3: Execução com falha de processo não-folha $p_{n/2}$.



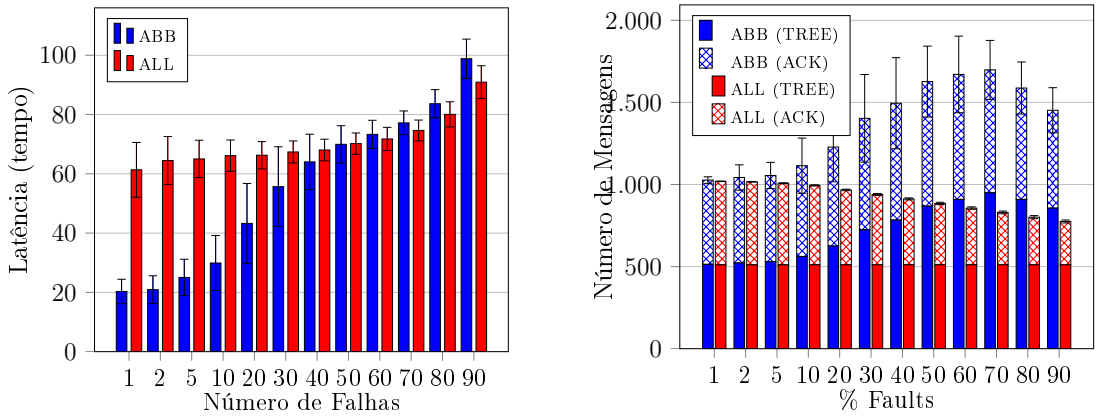
(a) Falhas para ABB (b) Falhas de não-folha

Figura 5.4: Total de mensagens em execuções com e sem falhas.

O total de mensagens TREE e ACK utilizados pelo algoritmo ABB é apresentado na Figura 5.4(a). Em cenários sem falha e com falha do processo folha o total de mensagens é basicamente o mesmo, exceto pela mensagem TREE e ACK que não são enviadas pelos processos falhos. Nos cenários com falha de não-folhas, o total de mensagens extras pode chegar a $2(n/2-1)$ menos os ACKs não enviados pelos processos falhos. A Figura 5.4(b) compara o total de mensagens utilizado pelos algoritmos ABB e ALL. Embora a latência de ABB seja menor para sistemas com grande número de processos, o total de mensagens é maior, visto que ALL não precisa retransmitir mensagens após a detecção de uma falha.

Cenários com Múltiplos Processos Falhos. Experimentos foram realizados em um sistema com 512 processos e diferente número de processos falhos. Como nos testes anteriores, o processo p_0 é o fonte, nunca falha e efetua o *broadcast* de uma única mensagem. Para cada porcentagem de falhas foram gerados 100 cenários nos quais os processos 1 a $n-1$ falham de forma aleatória em um tempo também aleatório entre 0.0 e 50.0.

No primeiro cenário, ABB foi comparado com ALL. A Figura 5.5(a) e a Figura 5.5(b)



(a) Latência (b) Total de Mensagens

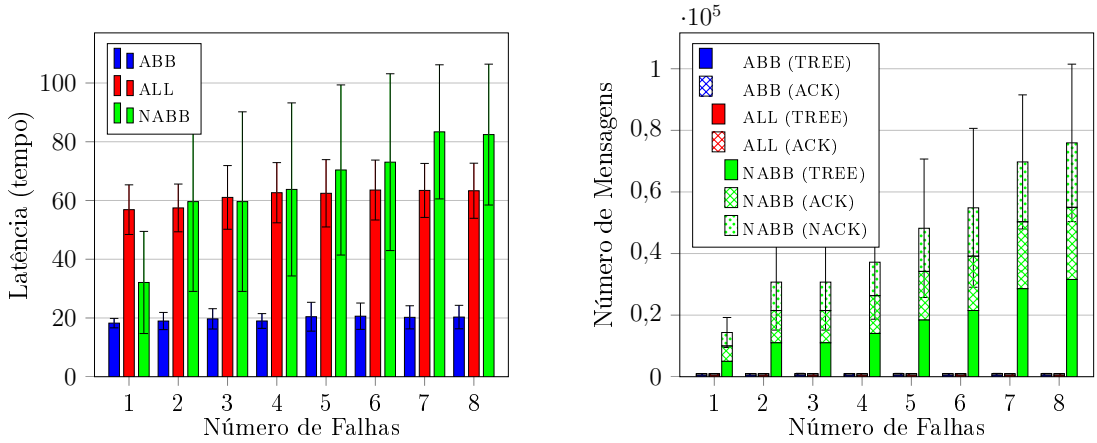
Figura 5.5: Execuções de ABB e TREE com 512 processos e falhas múltiplas.

mostram as médias de latência e vazão. É possível observar que para um número pequeno de falhas, ABB possui latência menor que ALL, como nos experimentos anteriores.

À medida que o número de falhas aumenta, a latência dos dois algoritmos também aumenta. Entretanto, a latência de ABB aumenta mais rápido e ultrapassa ALL após 60% de falhas. A vantagem de ABB neste caso é o controle não centralizado no processo fonte. No algoritmo ALL, mesmo que não haja retransmissão após a falha, o fonte precisa aguardar até ser notificado sobre todos os processos falhos antes de finalizar o processo de difusão.

Em termos de número de mensagens, como mostrado no experimento com falhas nos processos não-folhas, em ABB, se o processo falha após retransmitir as mensagens aos seus filhos, mensagens extras serão retransmitidas após a detecção da falha. Isto explica valores mais altos tanto de latência quanto de número de mensagens mostrados na Figura 5.5(a) e Figura 5.5(b), respectivamente. Quando mais de 50% dos processos falham, a latência de ABB é maior que a latência de ALL, visto que muitas subárvores precisam ser refeitas (mensagens retransmitidas). Uma vez que os clusters do VCube são progressivamente maiores, para altas taxas de falhas, a maioria delas ocorrerá nos clusters com maior quantidade de processos. Assim, dependendo da posição do processo falho, do instante em que a falha acontece e da latência do detector, mais retransmissões podem ser necessárias, o que explica o maior número de mensagens mostrado na Figura 5.5(b). Contudo, é possível notar que ABB apresenta boa escalabilidade, mesmo em cenários com múltiplos processos falhos.

Por fim, ABB, ALL e NABB foram comparados em diferentes cenários com falha, mas restringindo o número de falhas a $\log_2 n - 1$, uma vez que esta é a limitação de NABB para garantir uma única componente conexa no VCube. A latência de NABB é maior em todos os casos, mas é menor que ALL quando ocorrem poucas falhas, como mostrado na Figura 5.6(a). Em termos de número de mensagens, a Figura 5.6(b) mostra que a



(a) Latência (b) Total de Mensagens

Figura 5.6: Execução de ABB, NABB e TREE com 512 processos e falhas múltiplas.

reconstrução da árvore feita por NABB após cada falha diminui consideravelmente o seu desempenho. Este resultado confirma mais uma vez a eficiência da solução autonômica para manutenção da árvore geradora proposta neste trabalho.

5.6 Considerações Finais

Este capítulo apresentou soluções de difusão de melhor-esforço e difusão confiável baseadas na estrutura da árvore geradora proposta no Capítulo 4. Ficou demonstrado que ambas toleram até $n - 1$ processos falhos. A solução de difusão de melhor-esforço é utilizada pelo algoritmo de k-exclusão mútua proposto no Capítulo 6 para propagar as mensagens de requisição de maneira mais eficiente. Os resultados deste trabalho estão publicados em Rodrigues, Arantes e Duarte Jr. (2014a) e Rodrigues, Duarte Jr. e Arantes (2014b).

CAPÍTULO 6

UMA SOLUÇÃO AUTONÔMICA PARA K-EXCLUSÃO MÚTUA

A k-exclusão mútua refere-se a um conjunto de problemas bem conhecidos relacionados ao compartilhamento de recursos em sistemas distribuídos, como manutenção de cópias em banco de dados compartilhados. Conforme apresentado no Capítulo 3, as soluções de k-exclusão mútua existentes são basicamente adaptações dos algoritmos de 1-exclusão mútua. Entretanto, a maior parte destas soluções não aborda a questão de ocorrência de falhas no sistema (Naimi, 1996; Raymond, 1989a).

Este capítulo descreve a solução de k-exclusão mútua proposta baseada no mecanismo de monitoramento de processos (VCube) discutido no Capítulo 4 e no algoritmo de difusão de melhor-esforço apresentado no Capítulo 5. O algoritmo utiliza a abordagem de pedido de permissão e foi elaborado visando manter a eficiência na obtenção de recursos na presença de até $n - 1$ processos falhos. Uma versão preliminar da solução foi publicada em Rodrigues, Duarte Jr. e Arantes (2012) e uma segunda versão mais otimizada em Rodrigues et al. (2013).

Como uma segunda solução para a exclusão mútua, é apresentado um algoritmo de quóruns baseado no VCube, publicado em Rodrigues, Duarte Jr. e Arantes (2014c). Resultados experimentais demonstram a estabilidade de carga e tamanho dos quóruns gerados, em cenários com e sem falhas.

Considerando o resultado apresentado por Delporte-Gallet et al. (2005) que demonstra a necessidade de um detector perfeito para a solução do problema da exclusão mútua em sistemas baseados em troca de mensagens e sujeitos à falhas, considera-se que o VCube é implementado sobre um sistema síncrono, o que garante que não ocorrerão falsas suspeitas e que todo processo falho será detectado por todos os processos corretos em um tempo finito. Além disso, os enlaces são considerados confiáveis, isto é, não perdem, não duplicam e não alteram as mensagens transmitidas.

6.1 O Algoritmo de k-Exclusão Mútua Proposto

O algoritmo de k-exclusão mútua proposto neste trabalho é um algoritmo baseado em pedidos de permissão, como o de Raymond (1989a). O algoritmo faz uso de mensagens de requisição (REQUEST) e resposta (REPLY) para solicitar e dar permissão, respectivamente. As mensagens de requisição são enviadas utilizando o mecanismo hierárquico de difusão de melhor-esforço descrito no Capítulo 5. Para tratar os casos de falha, a

solução em questão faz uso do mecanismo auxiliar de monitoramento distribuído VCube apresentado no Capítulo 4, que informa o estado (falho ou sem-falha) de cada processo no sistema. A propagação em árvore das mensagens de difusão e a adaptação do algoritmo frente a ocorrência de falhas garantem a eficiência na obtenção de recursos para até $n-1$ processos falhos.

O Algoritmo 6.1 ilustra o pseudocódigo do algoritmo proposto. As variáveis locais mantidas pelos processos são:

- $correct_i$: conjunto dos processos considerados corretos pelo processo i;
- $state_i$: armazena o estado atual do processo, que pode ser $not\_requesting$, $requesting$ ou $executing$;
- $clock_i$: usada como relógio lógico local. Inicialmente em zero, é atualizado sempre que uma mensagem de requisição é enviada ou recebida, passando a armazenar o maior valor entre o relógio local e o *timestamp* da mensagem recebida;
- $last_i$: o valor de *timestamp* da última mensagem de requisição enviada;
- $perm\_count_i$: o total de permissões recebidas (mensagens de REPLY) desde a última tentativa de obtenção do recurso;
- $reply\_count_i[n]$: vetor que armazena a quantidade de respostas esperadas de cada processo;
- $defer\_count_i[n]$: contador que armazena a quantidade de respostas adiadas para cada processo. Após a liberação do recurso, o processo envia todas as permissões pendentes baseando-se nestas informações.

O algoritmo possui duas funções principais: uma função REQUEST() para iniciar o processo de requisição do recurso e uma função RELEASE() para liberar o recurso obtido. O processo de requisição inicia com a mudança de estado do processo para *requesting* (linha 9). Esta mudança garante que se p_i receber um pedido de permissão de um outro processo p_j, ela só será dada caso o relógio de p_j tenha um valor maior que o de p_i ou, em caso de empate, que o identificador de p_j seja menor (linha 26). Em seguida, uma mensagem marcada com o relógio local de p_i é enviada para os demais processos utilizando o algoritmo de *broadcast* de melhor-esforço apresentado no Capítulo 5.

Originalmente, para obter acesso ao recurso é preciso que p_i obtenha permissão de $n-k$ processos. Na proposta em questão, considerando que processos podem falhar, foi adicionada uma espera ativa que consulta o algoritmo de monitoramento para verificar dinamicamente a quantidade de processos falhos (linha 15). Este valor é descontado do total de respostas esperadas, permitindo que o processo obtenha acesso aos recursos

Algoritmo 6.1 Algoritmo autonômico de k-exclusão mútua

```
1:  correct_i ← {0, .., n − 1}
2:  state_i ← not_requesting
3:  ∀j ∈ n : reply_count_i[j] ← 0
4:  ∀j ∈ n : defer_count_i[j] ← 0
5:  perm_count_i ← 0
6:  clock_i ← 0
7:  last_i ← 0
8:  procedure REQUEST( )
9:      state_i ← requesting
10:     clock_i ← clock_i + 1
11:     last_i ← clock_i
12:     perm_count_i ← 0
13:     BROADCAST(REQUEST(i, last_i)) //usando Melhor-esforço
14:     ∀j ≠ i, j ∈ correct_i : reply_count_i[j] + +
15:     wait until (perm_count_i ≥ correct_i − k)
16:     state_i ← executing
17: procedure RELEASE( )
18:     state_i ← not_requesting
19:     for all (j ≠ i : j ∈ correct_i) do
20:         if (defer_count_i[j] ≠ 0) then
21:             SEND(REPLY(defer_count_i[j])) to p_j
22:             defer_count_i[j] ← 0
23: procedure RECEIVE(REQUEST(j, last_j)) from p_j
24:     clock_i ← max(clock_i, last_j)
25:     if (p_j ∈ correct_i) then
26:         if (state_i = executing or (state_i = requesting and (last_i, i) < (last_j, j))) then
27:             defer_count_i[j] + +
28:         else
29:             SEND(REPLY(i, 1)) to p_j
30: procedure RECEIVE(REPLY(j, count)) from p_j
31:     if (j ∈ correct_i) then
32:         reply_count_i[j] ← reply_count_i[j] − count
33:         if (state_i = requesting and reply_count_i[j] = 0) then
34:             perm_count_i + +
35: procedure CRASH(processo j)//j é detectado falho
36:     if (state_i = requesting and reply_count_i[j] = 0) then
37:         perm_count_i − −
38:     correct_i ← correct_i ∖ {j}
```

com mais eficiência. Ao receber a quantidade de permissões mínima, p_i obtém acesso ao recurso e é colocado no estado *executing*. Quando uma falha é detectada durante o *requesting*, o algoritmo verifica se já recebeu a permissão do processo falho (linhas 36-37). Se já recebeu, *perm_count* é decrementada para não interferir na condição da linha 15, garantindo a propriedade de *safety*. A liberação de um recurso por um processo p_i implica na mudança de estado do processo para *not_requesting* e no envio de todas as mensagens de requisição recebidas de outros processos que foram retidas e contabilizadas na estrutura *defer_count* (linhas 18-22). Com isso, os processos que estão aguardando permissões podem verificar a condição mínima da linha 15 e obter o recurso.

Para garantir o correto funcionamento do algoritmo, duas propriedades precisam ser satisfeitas. Em primeiro lugar, em cada instante de tempo, no máximo k processos diferentes podem estar utilizando os k recursos existentes, caracterizada pela propriedade de segurança (*safety*) e, em segundo lugar, mas não menos importante, é preciso garantir que se um processo correto solicita um recurso, ele o obterá em um intervalo de tempo finito, de acordo com a propriedade de progressão (*liveness*). Estas propriedades são garantidas e provadas formalmente pelos Lemas 6.1 e 6.2 e pelo Teorema 6.3.

Lema 6.1 (Segurança)**.** *No máximo k processos diferentes podem estar utilizando os k recursos existentes em cada instante de tempo.*

Prova. Considere que em um dado instante de tempo k processos estejam utilizando os k recursos disponíveis. Em seguida, um outro processo p_i inicia a operação de requisição, enviando mensagens de REQUEST para os $n-1$ outros processos. Cada processo p_j ao receber a solicitação pode tomar uma das seguintes decisões:

- Se p_j está no estado *not_requesting* ele responde imediatamente com um REPLY para p_i;
- Se p_j está no estado *executing* ele retém a resposta até que libere o recurso e, só então, envia REPLY para p_i;
- Se p_j está no estado *requesting* e $(last_i, i) < (last_j, j)$, ele envia imediatamente REPLY para p_i, pois neste caso p_i tem prioridade. Caso contrário, retém a resposta até que ele consiga obter o recurso e, só então, envia REPLY para p_i;
- O processo p_j está falho e, após a notificação do detector, p_i irá removê-lo da lista de corretos e desconsiderá-lo do conjunto de REPLYs esperados.

De acordo com a especificação do algoritmo, no máximo $n-k-1$ processos irão tomar a primeira decisão e permitir imediatamente o uso do recurso, uma vez que k processos estão utilizando os recursos (e adiarão a resposta) e o processo solicitante não contribui na decisão. Assim, p_i não terá acesso a um recurso e terá que aguardar por uma resposta adiada, que será enviada apenas quando um dos processos que estão utilizando recursos efetuar a liberação ou falhar. Com isso, garante-se que no máximo k processos estarão utilizando os k recursos em cada instante de tempo. ■

Lema 6.2 (Progressão)**.** *Se um processo correto solicita um recurso, ele o obterá em um intervalo de tempo finito.*

Prova. A única exceção que comprometeria a propriedade de progressão seria o caso em que o processo solicitante ficasse bloqueado indefinidamente na linha 15 do algoritmo, aguardando pela quantidade mínima de respostas exigida. No entanto, se existe um

recurso livre, em algum momento as respostas de autorização serão recebidas pelo processo solicitante, de acordo com as possíveis decisões já discutidas no Lema 6.1. Se algum processo falhar durante a operação, em algum momento o algoritmo de monitoramento irá detectar a falha e $correct_i$ será atualizado (linha 38), permitindo que o solicitante não mais aguarde pela resposta do processo falho. Se dois ou mais processos disputam um recurso, a propriedade em questão é garantida pela ordenação do relógio lógico. Logo, todo processo que solicitar um recurso o conseguirá em um tempo finito, de acordo com a ordem de solicitação ou, em caso de empate, com a prioridade do seu identificador. ■

Teorema 6.3. *O Algoritmo 6.1 soluciona o problema da k-exclusão mútua utilizando um algoritmo de diagnóstico e tolera até* $n-1$ *processos falhos.*

Prova. O Teorema 6.3 decorre diretamente do Lema 6.1 e do Lema 6.2. ■

6.2 Avaliação Experimental

Nesta seção são apresentados os resultados dos testes obtidos por simulação utilizando o *framework* Neko (Urbán et al., 2002). No contexto das simulações elaboradas, os algoritmos de exclusão mútua e o algoritmo de monitoramento são protocolos executados em cada processo. Uma camada intermediária foi inserida entre o o algoritmo de exclusão mútua e a rede para transportar as mensagens de REQUEST com base na *spanning tree* do algoritmo de difusão.

Assim como nas simulações dos algoritmos de difusão do Capítulo 5, o VCube foi implementado utilizando as classes de detecção disponíveis no pacote de tolerância a falhas do Neko. A Figura 6.1 representa a arquitetura utilizada na construção do ambiente simulado. O algoritmo de k-exclusão mútua (Mutex) envia as mensagens *broadcast* de REQUEST que são interceptadas pela camada responsável pela propagação na árvore (STA - *Spanning Tree Algorithm*). O VCube envia e recebe mensagens de ARE-YOU-ALIVE e I-AM-ALIVE diretamente na rede. Além disso, toda vez que o estado de um dos processos monitorados é modificado, o algoritmo de $k-$ exclusão mútua (Mutex) é notificado.

O algoritmo de k-exclusão mútua proposto, nomeado PROPO foi comparado com dois outros algoritmos: O Algoritmo de Raymond (Raymond, 1989a) (RAY) e o algoritmo de Bouillaguet, Arantes e Sens (Bouillaguet et al., 2008) (BAS). O algoritmo de Raymond não utiliza detecção de falhas, mas tolera intrinsecamente $k-1$ processos falhos. No entanto, cada falha implica em degradação da eficiência. O algoritmo BAS usa um detector de falhas e tolera $n-1$ falhas. Como já discutido na Seção 6.1, PROPO faz uso da árvore geradora para propagar as mensagens de REQUEST. Já os algoritmos RAY e BAS utilizam a estratégia um-para-todos, isto é, cada mensagem é enviada diretamente entre o processo fonte emissor e cada um dos $n-1$ processos do sistema.

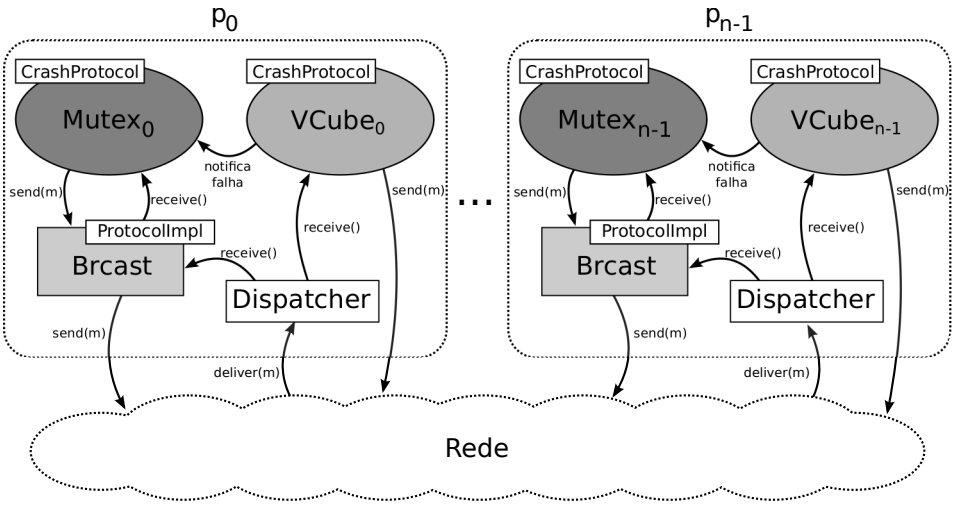


Figura 6.1: Organização dos módulos de simulação no Neko.

6.2.1 Parâmetros de Simulação

O modelo de simulação é o mesmo utilizado na Seção 5.5, baseado em Bulgannawar e Vaidya (1995). Cada processo executa uma nova requisição de recurso τ unidades de tempo após completar com sucesso a solicitação anterior e utiliza o recurso após conseguir permissão durante e unidades de tempo. Cada mensagem trocada entre dois processos consome $t_s + t_t + t_r$ unidades de tempo: t_s unidades de tempo para ser enviada e t_r para ser recebida. O tempo de transmissão utilizado pela rede é representado por t_t. Se uma mensagem é envidada para mais de um destino, t_s é computado novamente para cada cópia.

Para comparar os algoritmos foram utilizadas as seguintes métricas: (1) o total de recursos alocados durante a execução; (2) o tempo para um processo obter o recurso, definido como o intervalo de tempo a partir do instante que o processo solicita o recurso até o instante em que ele obtém permissão para acessá-lo.

6.2.2 Resultado dos Experimentos

Os testes foram divididos em duas categorias, sem falhas e com falhas. Inicialmente, os três algoritmos foram comparados em cenários sem falhas com sistemas de 8 a 1024 processos. Em seguida, as métricas de total de recursos alocados e tempo de obtenção de recursos das três soluções foram avaliadas um sistema com 512 processos, considerando situações com e sem falhas. Por fim, um cenário com 16 processos foi utilizado para comparar o algoritmo proposto com o seu original RAY. Neste último experimento, além dos recursos alocados em cada instante de tempo, foi medido também o total de mensagens utilizadas.

Os parâmetros e resultados são descritos a seguir. O parâmetro f indica o total de processos falhos em cada teste.

6.2.2.1 Experimentos em Cenários sem Falhas

Os três algoritmos, PROPO, RAY e BAS, foram executados em cenários com diferente número de processos, variando de 8 a 512 em potência de 2 e empregando dois tipos de carga: baixa (LOW) e alta (HIGH). Quando a carga é "LOW" no máximo k processos solicitam recursos ao mesmo tempo. Por outro lado, na carga "HIGH", todos os processos solicitam recursos concorrentemente. Os parâmetros foram configurados da seguinte forma em todas as execuções: tempo de uso do recurso $e = 0,0002$, tempos de envio e recebimento de mensagens $t_s = t_r = 0,1$, tempo de transmissão da mensagem pela rede $t_t = 0,8$, intervalo entre requisições $\tau = 0,1$ e total de recursos $k = 3$. Cada execução teve duração de 1.000 unidades de tempo.

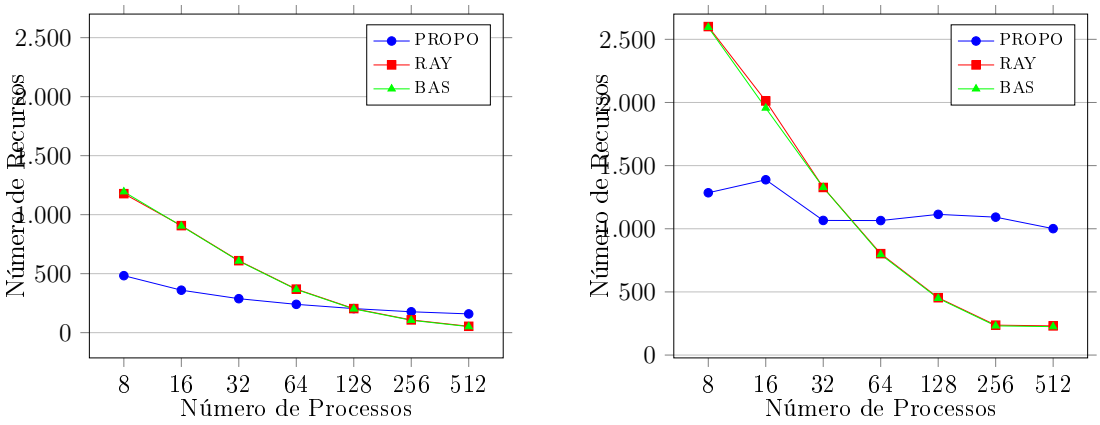
O total de recursos alocados em cada execução está registrado na Tabela 6.1.

Tabela 6.1: Total de recursos alocados nos cenários sem falhas ($f = 0$ e $k = 3$).

n	Carga	RAY	BAS	PROPO
8	LOW	1178	1194	483
	HIGH	2601	2595	1285
16	LOW	907	904	360
	HIGH	2012	1956	1388
32	LOW	610	607	288
	HIGH	1327	1327	1066
64	LOW	369	366	240
	HIGH	803	796	1065
128	LOW	204	201	204
	HIGH	454	449	1114
256	LOW	108	105	177
	HIGH	237	231	1092
512	LOW	54	51	159
	HIGH	231	225	1001

A Figura 6.2 ilustra graficamente o total de recursos alocados para cada tipo de carga. Por meio destes resultados, é possível concluir que a medida que o número de processos aumenta, a solução PROPO é mais eficiente que as outras duas, como pode ser observado nos sistemas a partir de 64 processos.

A Tabela 6.2 mostra o tempo mínimo (*min*), máximo (*max*) e a média (*mean*) para obter um recurso. Os valores de RAY e BAS são muito próximos em função do mecanismo de *broadcast* ponto-a-ponto que ambos utilizam. Quando o número de processos é maior que 64, a solução proposta apresenta maior eficiência tanto para tempo mínimo quanto para máximo. Isso se dá especialmente pelo mecanismo de difusão em árvore empregado na solução PROPO.



(a) LOW (b) HIGH

Figura 6.2: Total de recursos alocados em cenários sem falhas.

Tabela 6.2: Tempo para obtenção do recurso em cenários sem falhas ($f = 0$ e $k = 3$).

		RAY			BAS			PROPO		
n	Carga	min	max	mean	min	max	mean	min	max	mean
8	LOW	2.40	2.60	2.44	2.40	2.60	2.40	4.50	6.10	6.09
	HIGH	2.40	4.90	2.97	2.40	4.90	2.97	4.50	9.00	6.11
16	LOW	3.20	3.40	3.20	3.20	3.40	3.20	6.70	8.20	8.19
	HIGH	3.20	11.00	7.82	3.20	11.00	8.02	6.70	18.70	11.37
32	LOW	4.80	5.00	4.80	4.80	5.00	4.80	8.80	10.30	10.28
	HIGH	4.80	28.40	23.73	4.80	27.70	23.62	8.80	36.80	29.46
64	LOW	8.00	8.20	8.01	8.00	8.20	8.01	10.90	12.40	12.38
	HIGH	8.00	90.20	76.73	8.00	90.20	76.86	10.90	72.10	58.33
128	LOW	14.40	14.60	14.41	14.40	14.60	14.41	13.00	14.50	14.48
	HIGH	14.40	310.91	245.68	14.40	310.91	245.68	13.00	141.41	108.27
256	LOW	27.20	27.40	27.23	27.20	27.40	27.23	15.10	16.60	16.57
	HIGH	27.20	998.82	363.00	27.20	951.12	346.80	15.10	279.12	208.18
512	LOW	52.80	53.00	52.86	52.80	53.00	52.86	17.20	18.70	18.67
	HIGH	52.80	976.72	372.40	52.80	930.21	356.60	17.20	553.23	392.99

6.2.2.2 Experimentos em Cenários com Falhas

Neste segundo experimento foram utilizados 512 processos e $k = 10$ recursos. Foram utilizados cenários com e sem falhas, também empregando as cargas LOW e HIGH. A Tabela 6.3 mostra o número total de recursos alocados em cada teste com duração de 1.000 unidades de tempo. No cenário com falhas, $f = 10$ processos são escolhidos aleatoriamente para falhar. Os resultados mostram que a solução PROPO novamente apresenta melhores resultados em ambos os modelos de carga.

Para o último experimento, foram utilizados $n = 16$ processos e $k = 5$ recursos. O algoritmo de Raymond e a solução proposta PROPO foram configurados para enviar solicitações no modelo de carga LOW.

Falhas *crash* foram injetadas a cada 5,0 intervalos de tempo a partir do tempo 5,0. Em

Tabela 6.3: Recursos alocados nos cenários com falhas ($n = 512$, $f = 10$ e $k = 10$).

	Sem-falhas		Com-falhas	
	LOW	HIGH	LOW	HIGH
RAY	209	231	165	408
BAS	180	225	180	426
PROPO	530	1001	243	2274

Tabela 6.4: Tempo de obtenção dos recursos nos cenários com falhas ($n = 512$, $f = 10$ e $k = 10$).

	RAY			BAS			PROPO		
Sem-falhas	min	max	mean	min	max	mean	min	max	mean
LOW	52.10	54.00	52.32	52.10	53.00	52.29	17.00	19.10	18.68
HIGH	52.80	976.72	372.40	52.80	930.21	356.60	17.20	553.23	392.99
Com-falhas									
LOW	52.10	54.30	52.83	52.10	53.30	52.37	16.00	27.60	18.77
HIGH	52.10	996.21	355.70	52.10	939.91	357.12	17.00	302.92	182.30

cada intervalo um processo falha, iniciando do processo p_{15} até o processo p_1. O processo p_0 nunca falha. Cinco processos (p_0 a p_4) solicitam os recursos periodicamente. A seguir são apresentados os resultados de número de mensagens e o total de recursos alocados para o cenário descrito acima.

Número de Mensagens. No gráfico da Figura 6.3 é possível perceber nos intervalos iniciais que o número de mensagens enviadas pelo algoritmo proposto (já com as confirmações do STA) é 50% maior que na solução de Raymond, sem contabilizar as mensagens de monitoramento. Em valores absolutos, o algoritmo proposto enviou 8884 mensagens (3.007 requisições, 2.919 respostas e 2.958 confirmações) ao passo que o algoritmo de Raymond enviou 2.690 mensagens (1.425 requisições e 1.265 respostas). Isto representa uma sobrecarga significativa em relação à solução de Raymond, sem considerar as 7.448 mensagens de diagnóstico (5.100 requisições e 2.348 respostas). Entretanto, essa diferença deve-se ao fato de que o algoritmo de Raymond ficou bloqueado a partir da quinta falha, deixando de enviar mensagens de requisição após $t = 25,0$. Além disso, pode-se notar que na solução proposta o total de mensagens permanece equilibrado à medida que as falhas são injetadas.

Recursos alocados. A Figura 6.4 mostra a eficiência do algoritmo proposto. Como esperado, após $k-1 = 4$ falhas o algoritmo de Raymond não consegue mais obter recursos. Isso ocorre após a injeção da quinta falha em $t = 25,0$. Já o algoritmo proposto, com base nas informações fornecidas pelo algoritmo de monitoramento, consegue melhorar significativamente a eficiência até a falha de $n - 1$ processos. Após a falha de $n - k$ processos em $t = 60,0$ ocorre uma degradação constante, que é justificada pela inicio

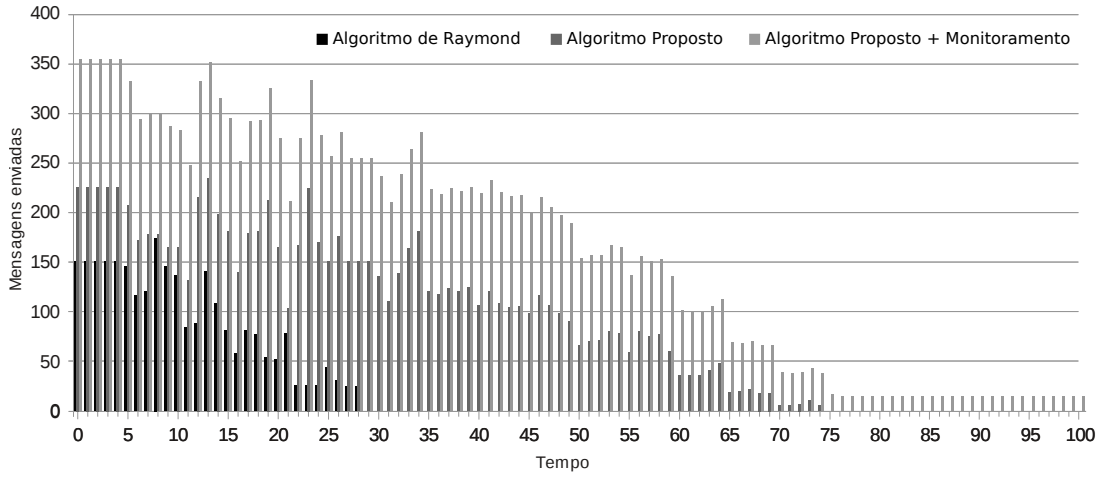


Figura 6.3: Comparativo de quantidade de mensagens enviadas.

da falha dos cinco processos solicitantes. Após o tempo 75.0 apenas o processo 0 (p_0) continua em execução.

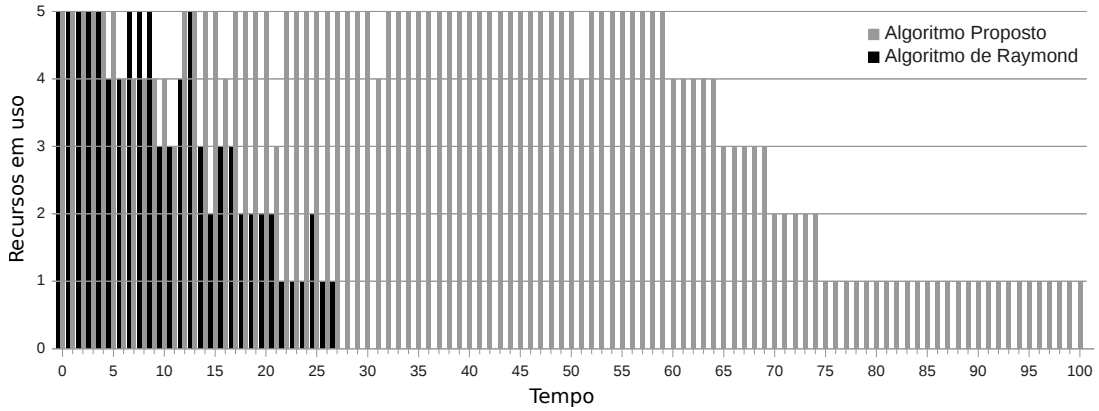


Figura 6.4: Comparativo de eficiência na obtenção dos recursos.

6.3 Uma Abordagem para Sistemas de Quóruns

Sistemas de quóruns foram introduzidos por Thomas (1979) como uma solução para coordenar ações e garantir consistência em um sistema de bancos de dados replicado. Considere um sistema distribuído como um conjunto finito P de $n > 1$ processos independentes $\{p_0, ..., p_{n-1}\}$ que se comunicam usando troca de mensagens. Um *sistema de quóruns* em P é um conjunto de subconjuntos de P, chamados *quóruns*, no qual cada par de subconjuntos tem uma intersecção não-vazia (Merideth e Reiter, 2010). Garcia-Molina e Barbara (1985) estenderam esta definição introduzindo o conceito de *coteries*, isto é, grupos exclusivos. Este modelo inclui a propriedade de minimalidade, na qual nenhum conjunto contém outro conjunto do sistema, o que o torna mais eficiente.

O algoritmo proposto por Thomas implementa um mecanismo tradicional de votação no qual o vencedor é o valor com a maioria dos votos. Cada atualização em um item de dados é marcada com um *timestamp* e executada em um quórum composto pela maioria dos servidores. Quando um cliente precisa ler um valor do banco, ele acessa novamente um quórum com a maioria das réplicas e escolhe o item com o *timestamp* mais atual. Em um sistema com três servidores, por exemplo, se a escrita for replicada em quaisquer dois servidores, toda leitura posterior em no mínimo dois servidores retornará o valor mais atual, isto é, a intersecção dos quóruns garante que, no mínimo, uma replica terá o item de dado mais recente. O trabalho de Thomas considera que cada processo contribui de forma igualitária na votação. Gifford (1979) estendeu este modelo atribuindo pesos aos processos. Além disso, os quóruns são divididos em duas classes, leitura e escrita, e somente quóruns de classes diferentes precisam se intersectar.

Além da replicação de dados (Liu et al., 2011; Abawajy e Deris, 2013), sistemas de quórum têm sido utilizados por diversas outras aplicações, como a exclusão mútua (Maekawa, 1985; Fujita, 1998; Atreya et al., 2007; Naimi e Thiare, 2013). Em uma solução de exclusão mútua baseada em pedidos de permissão sem a utilização de quóruns, se um processo deseja obter acesso a um recurso compartilhado, ele deve enviar uma mensagem de requisição para *todos* os outros processos do sistema e aguardar pelas mensagens de permissão (Ricart e Agrawala, 1981). Neste caso, um sistema de quórum pode reduzir o total de mensagens, visto que este mesmo processo precisa enviar a requisição apenas para um subconjunto dos processos que forma um quórum. Os processos que respondem ao pedido de permissão ficam bloqueados para novos pedidos até que o recurso seja liberado. A intersecção entre os quóruns garante a integridade da exclusão mútua.

Em geral, as soluções de quóruns são avaliadas em termos de tamanho, carga e disponibilidade (Vukolić, 2010). O tamanho é o número de processos em cada quórum e a carga indica em quantos quóruns cada processo está contido. A disponibilidade está ligada ao impacto das falhas no conjunto de quóruns do sistema. O modelo de Maekawa (1985), por exemplo, gera quóruns de tamanho próximo a $\sqrt{n}$, sendo n o número de processos no sistema. No entanto, a solução não pode ser aplicada para qualquer valor de n. Agrawal e El Abbadi (1991) utilizam árvores binárias para criar quóruns de tamanho $\log_2 n$, mas que podem chegar a $\lceil (n+1)/2 \rceil$ em cenários com falhas. Além disso, a carga é altamente concentrada na raiz e proporcionalmente nos nodos próximos a ela.

A solução de Thomas (1979) faz uso dos chamados quóruns majoritários. Um quórum majoritário Q em P tem $|Q| = \lceil (n+1)/2 \rceil$ elementos. A grande vantagem deste tipo de quórum é a alta disponibilidade. Mesmo em uma solução estática, os quóruns majoritários toleram $f < n/2$ processos falhos (Lipcon, 2012). A tolerância a falhas pode ser ainda aprimorada se o modelo permite a atualização dinâmica do sistema (Amir e Wool, 1996). Além disso, embora outras soluções apresentem modelos teóricos com propriedades otimizadas em termos do número de participantes em cada quórum, as soluções majoritárias

continuam sendo as mais utilizadas na prática.

Este trabalho apresenta um sistema de quórum majoritário construído sobre a topologia de hipercubo virtual do VCube. Processos podem falhar por *crash* e uma falha é permanente. Após uma falha, a topologia do VCube é reestruturada estabelecendo-se novos enlaces entre os processos anteriormente conectados ao processo falho. As propriedades logarítmicas herdadas do hipercubo são mantidas mesmo após a ocorrência de falhas. A avaliação teórica e testes comparativos mostram que os quóruns formados possuem tamanho e carga balanceados e o sistema tolera até $n-1$ processos falhos.

6.3.1 Definição das Funções

Com base na organização virtual dos processos utilizada pelo VCube e na função $c_{i,s}$ apresentados na Seção 4.3, foram definidas algumas funções para auxiliar a implementação do algoritmo de quóruns proposto neste trabalho.

Inicialmente, o conjunto $correct_i$ é definido para registrar as informações que o processo i tem sobre o estado dos processos informadas pelo VCube. Estas informações são obtidas através dos testes realizados pelo monitoramento. Sendo assim, em razão da latência do detector, é possível que um processo j falho ainda seja considerado como correto pelo processo i. No entanto, tão logo o processo i seja informado sobre a falha, j é removido de $correct_i$ e, após o diagnóstico completo, todos os processos terão a mesma informação sobre processos corretos e falhos.

Seja o resultado da função $c_{i,s}$ uma lista $(a_1, a_2, .., a_m)$, $m = 2^{s-1}$ dos processos do cluster s em relação ao processo i. A lista de m' elementos considerados corretos por i em um cluster $c_{j,s}$ é denominada como $FF\_cluster_i(s) = (b_1, .., b_{m'}), b_k \in (c_{i,s} \cap correct_i)$, $k = 1..m', m' \leq m$. Desta forma, a função $FF\_mid_i(s) = (b_1, .., b_{\lceil m'/2 \rceil})$ gera um conjunto que contém a metade absoluta dos processos considerados corretos pelo processo i em um cluster $c_{i,s}$. Se não existem processos corretos no cluster s, $FF\_cluster_i(s) = \perp$ e, por conseguinte, $FF\_mid_i(s) = \perp$. Note que esta função é dependente do conhecimento atual que um processo tem a respeito das falhas no sistema. Devido à latência de detecção, em um mesmo espaço de tempo, é possível que dois processos possuam visões diferentes sobre quais processos estão corretos ou falhos.

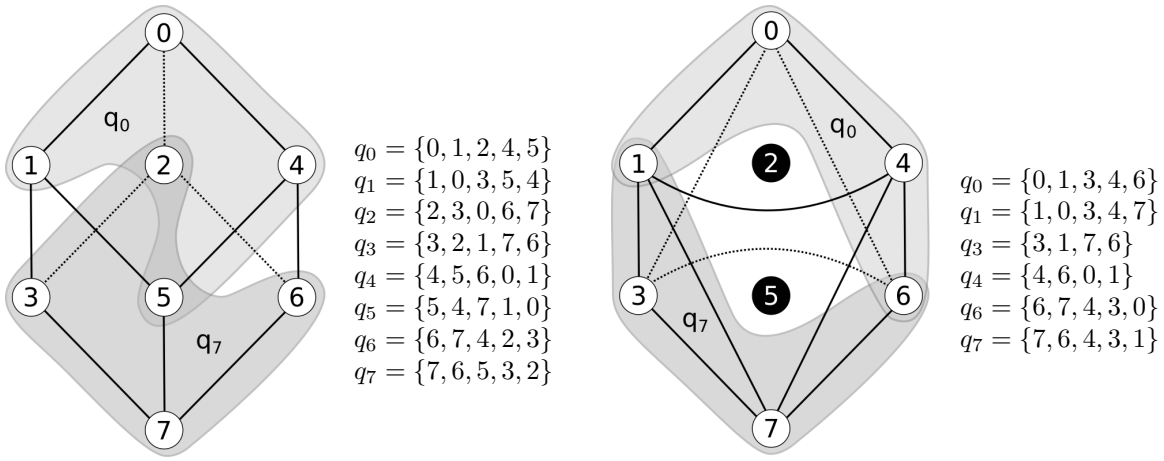
6.3.2 Descrição do Algoritmo

Em termos gerais, cada processo i constrói o seu próprio quórum adicionando a si mesmo e a metade absoluta dos elementos j que ele considera sem-falha ($j \in correct_i$) em cada cluster $c_{i,s}$ do hipercubo virtual definido por VCube. O estado dos processos é informado pelo VCube. O Algoritmo 6.2 apresenta um pseudo-código desta solução.

Como exemplo, considere o cenário sem falhas representado pelo hipercubo de três

Algoritmo 6.2 Obtenção do quórum de um processo j pelo processo i

```
1:  correct_i ← {0, .., n − 1}
2:  function GetQuorum
3:      q_i ← {i}
4:      for s ← 1, .., log_2 n do
5:          //adicionar ao quórum q_i a metade absoluta dos
6:          //processos considerados corretos por i no seu cluster s
7:          q_i ← q_i ∪ FF_mid_i(s)
8:      return q_i
9:  upon notifying crash(j)
10:     correct_i ← correct_i ∖ {j}
```



(a) Sem falhas

(b) Processos p_2 e p_5 falhos

Figura 6.5: Quóruns do 3-VCube com representação gráfica para os processos p_0 e p_7.

dimensões da Figura 6.5(a), que representa os quóruns dos processos p_0 e p_7. Estes processos foram escolhidos por estarem o mais distante possível um do outro no hipercubo. O quórum q_0 calculado pelo processo p_0 é composto por ele mesmo e pela metade absoluta dos clusters $c_{0,1} = (1)$, $c_{0,2} = (2, 3)$ e $c_{0,3} = (4, 5, 6, 7)$. Neste caso, $FF\_mid_0(1) = (1)$, $FF\_mid_0(2) = (2)$ e $FF\_mid_0(3) = (4, 5)$. Logo, $q_0 = \{0, 1, 2, 4, 5\}$. O quórum q_7 referente ao processo p_7 é formado por $FF\_mid_7(1) = (6)$, $FF\_mid_7(2) = (5)$ e $FF\_mid_7(3) = (3, 2)$. Assim, $q_7 = \{7, 6, 5, 3, 2\}$. Logo, a intersecção, $q_0 \cap q_7 = \{2, 5\}$. A Figura 6.5(b) ilustra um cenário após a falha dos processos da interseção p_2 e p_5. Neste caso, a topologia virtual é reestruturada e o processo p_6 passa a fazer parte do quórum de p_0 pois, dada a falha de p_5, $FF\_mid_0(3) = (4, 6)$. O processo p_1 é incluído no quórum de p_7 por razões semelhantes. Assim, p_1 e p_6 passam a compor a nova interseção.

Teorema 6.4 (Intersecção)**.** *Todo quórum q_i construído por um processo i utilizando o Algoritmo 6.2 tem uma interseção com cada outro quórum do sistema de, no mínimo, dois elementos.*

Prova. A prova é por indução baseada nos clusters do VCube. Sem perda de generalidade, considera-se que o número de processos n do sistema é uma potência de 2. Para os demais valores de n basta considerar o VCube com dimensão suficiente para acomodar todos os processos do sistema e considerar os vértices do hipercubo não utilizados por processos válidos como falhos.
Base: o teorema é valido para um VCube de dimensão 1 com dois processos $\{p_0, p_1\}$. O processo p_0 adiciona a si mesmo e o processo p_1 ao seu quórum, visto que $c_{0,1} = (1)$ e $FF\_mid_0(1) = (1)$. O processo p_1 adiciona p_0 ao seu quórum de forma semelhante. Neste caso, $q_0 \cap q_1 = \{0, 1\}$.
Hipótese: suponha que o teorema é válido para um VCube de d dimensões.
Passo: considere um VCube com $d+1$ dimensões com $n = 2^{d+1}$ processos. Cada cluster $c_{i,s}$ tem $2^s/2$ elementos. A soma dos elementos de todos os clusters s de um processo i é dada por $2^1/2 + .. + 2^d/2 + 2^{d+1}/2 = 2^{d+1} - 1 = n - 1$. Portanto, a soma das metades absolutas dos elementos de cada cluster s é dada por:

$$\left\lceil \frac{\frac{2^1}{2} + .. + \frac{2^d}{2} + \frac{2^{d+1}}{2}}{2} \right\rceil = \left\lceil \frac{2^{d+1} - 1}{2} \right\rceil = \left\lceil \frac{n-1}{2} \right\rceil \tag{6.1}$$

Sendo n uma potência de 2 e, portanto, par, $\left\lceil \frac{n-1}{2} \right\rceil$ equivale a $\frac{n}{2}$.

Sejam q_i e q_j dois quóruns construídos por dois processos quaisquer i e j, respectivamente. Como cada processo i adiciona a si mesmo ao seu quórum q_i, $|q_i| = 1 + \frac{n}{2}$. Seja $P = \{p_0, .., p_{n-1}\}$ o conjunto de processos que representa o VCube em questão. Considere que $q_i = \{p_0, .., p_{n/2-1}\}$ e $q_j = \{p_{n/2}, .., p_{n-1}\}$ contém cada um, a metade distinta dos elementos de P. A adição de qualquer elemento $p_a \in q_i$ a q_j e de $p_b \in q_j$ a p_i garante a maioria para ambos e, por conseguinte, a intersecção em, no mínimo, dois elementos: $\{p_a, p_b\}$.

Assim, por indução, o teorema é válido. ■

6.3.3 Análise do Algoritmo

Nesta seção são discutidas as propriedades do sistema de quóruns proposto. As métricas avaliadas são o tamanho e a carga dos quóruns, e os aspectos de disponibilidade e tolerância a falhas.

Tamanho e Carga dos Quóruns. Em um sistema de quóruns ideal, todos os quóruns são do mesmo tamanho, a interseção entre dois quóruns quaisquer tem o mesmo número de elementos e cada elemento pertence ao mesmo número de quóruns.

O tamanho de um quórum q_i no VCube pode ser dado por:

$$|q_i| = 1 + \sum_{s=1}^{\log_2 n} |FF\_mid_i(s)| \tag{6.2}$$

Considerando que cada cluster possui 2^s processos, a metade absoluta quando não existem processos falhos no cluster s é $\lceil |c_{i,s}|/2 \rceil$. No total, a metade dos processos é $\sum_{s=1}^{\log_2 n} \lceil \frac{2^s}{2} \rceil$. A inclusão do próprio processo i completa a maioria. Assim, nos cenários sem falha, os quóruns são uniformes e possuem exatamente $(n/2+1)$ elementos. Além disso, nenhum conjunto está contido em outro (*coteries*).

Seja f_s o total de processos falhos em um cluster s. Se $|c_{i,s}| - f_s$ é par, a metade que corresponde aos processos corretos é mantida. Se é ímpar, a quantidade de elementos que fazem parte do quórum do cluster é o resultado da divisão inteira de $\frac{|FF\_cluster_i(s)|}{2}$ acrescido de 1. Assim, no pior caso, o tamanho do quórum será:

$$|q_i| = 1 + \sum_{s=1}^{\log_2 n} \left\lfloor \frac{|c_{i,s}| - f_s}{2} \right\rfloor + c, c = \begin{cases} 0 & \text{se } |c_{i,s}| - f_s \text{ é par} \\ 1 & \text{se } |c_{i,s}| - f_s \text{ é ímpar} \end{cases} \quad (6.3)$$

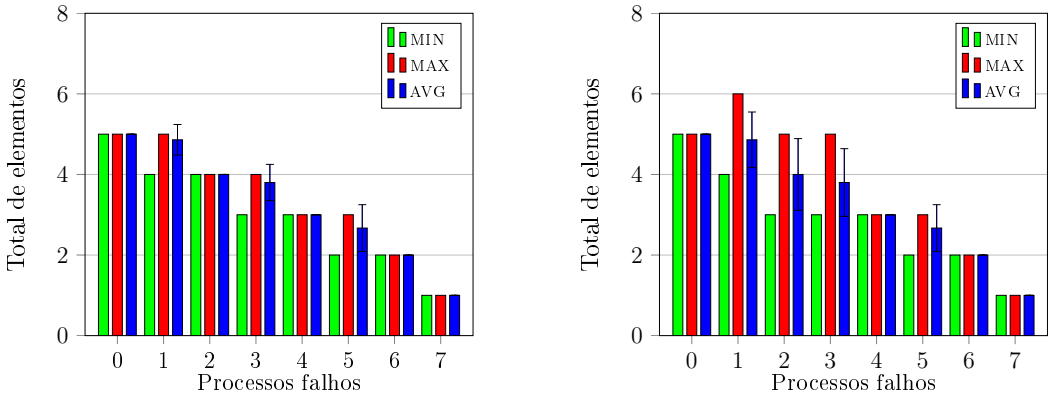
Em relação à carga, quando não existem processos falhos, a característica simétrica da $c_{i,s}$ garante a igual distribuição dos processos nos clusters e, portanto, o equilíbrio na inclusão dos elementos em cada quórum. Em caso de falha de um processo j, se $j \in c_{i,s}$, os processos corretos k que pertencem ao cluster s do processo i terão uma maior probabilidade de pertencer a um novo quórum. Se após a falha $c_{i,s} - f_s$ é ímpar, ao menos um processo k será incluído no quórum do processo i, possivelmente aumentando a carga de k, exceto se k pertencia ao quórum do processo j que falhou.

Como exemplo, considere o sistema P com $n = 8$ processos. O gráfico da Figura 6.6(a) mostra o tamanho dos quóruns quando não existem processos falhos e após a detecção de falhas. Inicialmente, quando não existem processos falhos cada quórum possui exatamente $(n/2+1) = 5$ elementos. Após a ocorrência de uma falha, restam 7 processos corretos e os quóruns passam a ter entre 4 (a maioria absoluta de 7) e 5 elementos (a maioria mais um). Quando existem mais processos falhos o comportamento é semelhante. A carga está representada na Figura 6.6(b). Quando não há falhas o tamanho e a carga são idênticos, o que mostra a uniformidade dos quóruns. Após a ocorrência de uma falha, a carga varia entre 4 e 6, mas a média se mantém em 5. Para 4 processos falhos o tamanho e a carga são iguais novamente e cada quórum possui a maioria dos processos corretos, isto é, $4/2 + 1 = 3$ elementos.

Disponibilidade. A disponibilidade é a capacidade do sistema de quóruns em tolerar falhas. Na solução com VCube, após o diagnóstico completo do nodo falho, todos os processos atualizam seus quóruns automaticamente. Assim, mesmo após $n-1$ falhas é possível reconstruir o sistema de quóruns, ou seja, o sistema proposto é $(n-1)$-resiliente.

6.3.4 Avaliação Experimental

Para realizar os experimentos, o algoritmo proposto foi implementado em Java. A solução de quóruns proposta foi comparada com o algoritmo de quóruns em árvore de Agrawal



(a) Tamanho (b) Carga

Figura 6.6: Tamanho e carga dos quóruns em um sistema VCube com 8 processos.

e El-Abbadi (1991), denominada TREE. Foi utilizada uma árvore binária balanceada com raiz em p_0, na qual os demais processos são adicionados sequencialmente seguindo uma distribuição em largura. Primeiro são apresentados os resultados para cenários sem processos falhos e, em seguida, para os cenários com falhas.

6.3.4.1 Cenários sem Falhas

A Tabela 6.5 mostra as propriedades dos quóruns TREE e VCube para cenários com número de processos n variando de 8 a 1024 em potência de dois. A quantidade de quóruns em cada algoritmo está representada por $|Q|$. Para o tamanho dos quóruns e a carga de um processo, isto é, o total de elementos em cada quórum e a quantidade de quóruns a qual cada elemento pertence, são mostrados o menor ($<$), o maior ($>$), a média ($\overline{q_i}$) e o desvio padrão (σ).

Quando não há falhas, o algoritmo TREE gera $|Q| = n/2$ quóruns com tamanho médio $\overline{q_i}$ muito próximo a $\log_2 n$. Neste exemplo, em função do valor de n, a árvore gerada é completa e balanceada, exceto pelo último elemento folha que compõe o último nível da árvore. Este elemento faz parte do único quórum formado pelo ramo esquerdo mais longo da árvore e que possui tamanho $\log_2 n + 1$. A solução VCube, por outro lado, gera n quóruns, um para cada processo, e cada quórum possui $n/2 + 1$ elementos. O tamanho e a carga são idênticos, visto que, em função da simetria do cálculo de $c_{i,s}$, cada processo está contido em $n/2 + 1$ quóruns.

Em relação à carga, verifica-se que à medida que o VCube a distribui igualitariamente entre os processos, TREE sobrecarrega a raiz, bem como proporcionalmente os nodos nos níveis mais próximos a ela. Considerando uma árvore binária, a raiz está contida em todos os quóruns, os elementos do nível seguinte em $|Q|/2$ conjuntos e assim sucessivamente até as folhas, que pertencem a apenas um quórum. Nas medidas de carga da Tabela 6.5, a maior carga ($>$) é da raiz é a menor ($<$) representa as folhas. O desvio padrão confirma

Tabela 6.5: Resultados dos testes em cenários sem falhas.

	TREE	Tamanho				Carga				VCube	Tamanho/
n	$\|Q\|$	$<$	$>$	$\overline{q_i}$	σ	$<$	$>$	$\overline{q_i}$	σ	$\|Q\|$	Carga
8	4	3	4	3,25	0,50	1	4	1,63	1,06	8	5
16	8	4	5	4,13	0,35	1	8	2,06	1,88	16	9
32	16	5	6	5,06	0,25	1	16	2,53	3,07	32	17
64	32	6	7	6,03	0,18	1	32	3,02	4,77	64	33
128	64	7	8	7,02	0,13	1	64	3,51	7,18	128	65
256	128	8	9	8,01	0,09	1	128	4,00	10,58	256	129
512	256	9	10	9,00	0,06	1	256	4,50	15,35	512	257
1024	512	10	11	10,00	0,04	1	512	5,00	22,07	1024	513

esta dispersão. Além disso, em um sistema que utiliza TREE, se todos os processos precisam acessar um quórum a carga pode ser ainda maior, visto que $|Q|$ representa metade do número de processos.

6.3.4.2 Cenários com Falhas

Inicialmente foi testado o impacto da falha de um único processo nas duas soluções. A Tabela 6.6 mostra os valores de tamanho e carga dos quóruns no VCube quando um único processo está falho. Nota-se que o tamanho dos quóruns se mantém estável e a carga, embora não mais simétrica, é ainda bem distribuída.

Tabela 6.6: Resultados dos testes com VCube para a falha de um único processo.

VCube		Tamanho				Carga			
n	$\|Q\|$	$<$	$>$	$\overline{q_i}$	σ	$<$	$>$	$\overline{q_i}$	σ
8	7	4	5	4,86	0,38	4	6	4,86	0,69
16	15	8	9	8,93	0,26	8	10	8,93	0,70
32	31	16	17	16,97	0,18	16	18	16,97	0,71
64	63	32	33	32,98	0,13	32	34	32,98	0,71
128	127	64	65	64,99	0,09	64	66	64,99	0,71
256	255	128	129	129,00	0,06	128	130	129,00	0,71
512	511	256	257	257,00	0,04	256	258	257,00	0,71
1024	1023	512	513	513,00	0,03	512	514	513,00	0,71

A posição do nodo falho no VCube influencia os quóruns independentemente, não tem impacto no desempenho geral do sistema. Em TREE, por outro lado, cada falha em um nível da árvore altera o sistema como um todo, exceto pelos elementos folha, que invalidam o quórum nos caminhos da árvore que os contém e nos quais o impacto é maior quando combinados com múltiplas falhas. A Tabela 6.7 mostra os resultados para um único processo falho no segundo nível da árvore, isto é, um filho da raiz. O tamanho dos quóruns aumenta em relação ao cenário sem falhas, especialmente para os sistemas com maior número de processos. Isto é esperado em função do maior número de combinações entre as sub-árvores esquerda e direita do nodo falho. A carga também é aumentada, mas a carga mínima ainda é 1, presente nos quóruns da sub-árvore esquerda da raiz.

Tabela 6.7: Resultados dos testes com TREE para um processo falho não-raiz (p_1).

TREE		Tamanho				Carga			
n	$\|Q\|$	$<$	$>$	$\overline{q_i}$	σ	$<$	$>$	$\overline{q_i}$	σ
8	3	3	4	3,33	0,58	1	3	1,43	0,79
16	8	4	6	4,75	0,89	1	8	2,53	1,85
32	24	5	8	6,50	1,14	1	24	5,03	5,24
64	80	6	10	8,50	1,29	1	80	10,79	15,54
128	288	7	12	10,61	1,30	1	288	24,06	45,91
256	1088	8	14	12,74	1,20	1	1088	54,34	134,12
512	4224	9	16	14,83	1,04	1	4224	122,61	388,01
1024	16640	10	18	16,90	0,87	1	16640	274,89	1114,43

A Tabela 6.8, também com resultados obtidos para TREE, apresenta os dados para o caso em que a raiz (processo p_0) está falha. Em função das combinações entre os elementos da árvore esquerda e direita da raiz, a quantidade de quóruns disponíveis representada na coluna $|Q|$ aumenta consideravelmente. O tamanho dos quóruns praticamente dobrou em relação ao cenário sem falhas. As cargas mínima e máxima também aumentam, mas continuam muito desproporcionais. Como cada quórum é formado pela combinação de um caminho da sub-árvore esquerda com um caminho da sub-árvore direita, a carga mínima aumenta proporcionalmente ao número de possíveis combinações.

Tabela 6.8: Resultados dos testes com TREE para a falha da raiz (p_0).

TREE		Tamanho				Carga			
n	$\|Q\|$	$<$	$>$	$\overline{q_i}$	σ	$<$	$>$	$\overline{q_i}$	σ
8	4	4	5	4,50	0,58	2	4	2,57	0,98
16	16	6	7	6,25	0,45	4	16	6,67	4,19
32	64	8	9	8,13	0,33	8	64	16,77	14,95
64	256	10	11	10,06	0,24	16	256	40,89	49,00
128	1024	12	13	12,03	0,17	32	1024	97,01	152,61
256	4096	14	15	14,02	0,12	64	4096	225,13	449,65
512	16384	16	17	16,01	0,09	128	16384	513,25	1353,97
1024	65536	18	19	18,00	0,06	256	65536	1153,48	3930,10

Em resumo, embora o tamanho dos quóruns de VCube seja maior que TREE, a carga de VCube é uniformemente distribuída e o sistema é mais estável em cenários com falhas.

6.4 Considerações Finais

A maior eficiência obtida pela solução de k-exclusão mútua proposta nos testes com maior número de processos é justificada pela abordagem hierárquica, pelo sistema de monitoramento VCube e pelo mecanismo de disseminação de mensagens. Mesmo considerando a latência de propagação da árvore, e não havendo mecanismo de *broadcast* disponível em nível de rede/enlace, é mais eficiente para o processo fonte enviar $\log_2 n$ mensagens que serão retransmitidas em paralelo em cada nível da árvore que tratar $n-1$ mensagens ponto-a-ponto. Além disso, criar e gerenciar conexões confiáveis para um grande número

de destinatários é, muitas vezes, impraticável em nível de sistema operacional.

Uma proposta feita para melhorar a eficiência da solução de exclusão mútua é um algoritmo baseado em quóruns. Para tanto, um sistema de quóruns com VCube foi comparado com um modelo em árvore binária. Os experimentos em cenários sem falhas mostram que os quóruns gerados no VCube são uniformes e têm tamanho $\lceil (n+1)/2 \rceil$. Nos cenários com falha, o tamanho e a carga dos quóruns podem aumentar, mas ainda se mantém equilibrados. Além disso, ao contrário de TREE, a posição do processo falho não tem impacto no desempenho geral do sistema. Em relação à disponibilidade, o VCube é capaz de reconstruir dinamicamente os quóruns com até $n-1$ processos falhos.

CAPÍTULO 7

CONCLUSÃO

Neste trabalho foi apresentada uma solução autonômica de k-exclusão mútua com pedido de permissão. A solução faz uso de um mecanismo de monitoramento chamado VCube. No VCube, os processos são organizados em uma topologia virtual baseada em hipercubo, que possui diversas propriedades logarítmicas. Em caso de falhas de processos (*crash*), a topologia se adapta automaticamente, removendo processos falhos e reconectando os processos corretos. Os processos são organizados em clusters progressivamente maiores e testes são realizados em rodadas de forma adaptativa pelos processos entre os clusters de cada nível. Em cada teste, um processo testador obtém informações sobre o processo testado (falho ou sem-falha), bem como sobre os demais processos do sistema, utilizando *timestamps* para obter apenas as informações mais atualizadas. Em termos de latência de diagnóstico, VCube apresenta latência média $\log_2 n$ e máxima $\log_2^2 n$). Com as informações fornecidas pelo VCube, a solução de exclusão mútua proposta é capaz de manter a eficiência na obtenção de recursos para até $n-1$ processos falhos.

Para prover escalabilidade à solução de exclusão mútua, foram propostos dois algoritmos de difusão de mensagens (*broadcast*). O algoritmo de difusão de melhor-esforço proposto garante que uma mensagem enviada por um processo emissor correto é entregue a todos os demais processos corretos do sistema em um tempo finito. No entanto, se o emissor falhar antes de concluir o processo de difusão, alguns processos podem receber a mensagem e outros não. O algoritmo de difusão confiável proposto garante a entrega da mensagem mesmo em caso de falha do emissor. A duas soluções de difusão utilizam um algoritmo de árvore geradora para propagar as mensagens através da topologia do VCube. O algoritmo de árvore geradora, também proposto neste trabalho, utiliza a organização hierárquica em clusters do VCube para construir de forma totalmente distribuída uma árvore geradora mínima contendo todos os processos corretos do sistema. A árvore é construída com base nas informações do estado dos processos obtidas através dos testes realizados pelo VCube. Em caso de falhas, a árvore é regenerada automaticamente apenas nos ramos afetados pela falha, sem a necessidade de troca de informações entre os processos ou paralisação do sistema. Na solução de exclusão mútua proposta, a difusão de melhor-esforço é utilizada para propagar de maneira eficiente e escalável as mensagens de requisição de recursos. A solução é considerada autonômica no sentido que se adapta dinamicamente e de forma transparente frente a ocorrência de falhas dos processos.

Além das especificações e provas formais, os algoritmos de difusão foram comparados por meio de simulação com duas outras soluções, uma baseada em árvore não autonômica e

outra utilizando comunicação ponto-a-ponto. Com base em diferentes cenários com e sem falhas, foi possível observar a eficiência das soluções propostas em termos de latência de propagação e vazão (quantidade de mensagens/tempo). Nos testes com aumento gradativo do número de processos observou-se também a escalabilidade da solução.

O algoritmo de k-exclusão mútua também foi comparado com duas outras soluções da literatura: o algoritmo de Raymond (1989) e a solução de Bouillaguet, Arantes e Sens (2008). O algoritmo de Raymond tolera até $k-1$ processos falhos, mas cada falha degrada a solução. O algoritmo de Bouillaguet, Arantes e Sens utiliza um detector de falhas para tolerar até $n-1$ falhas. Duas variações de carga foram utilizadas. Com caraga baixa, apenas k processos solicitam um dos k recursos disponíveis. Nos cenários com carga alta, todos os processos sem falha disputam os recursos. Nos testes, embora o total de mensagens da solução proposta seja maior em função do monitoramento e das retransmissões nos casos de falhas, a eficiência na obtenção de recursos é mantida durante toda a execução do sistema, independente do número de processos falhos. Além disso, o total de recursos alocados em cenários sem falhas é maior para sistemas com maior quantidade de processos e, em cenários com falhas, é maior em todos os casos testados. Por fim, considerando sistemas com mais de 64 processos, o tempo para a obtenção dos recursos após uma solicitação é menor na solução proposta que nas demais. Estes dados comprovam a eficiência e a escalabilidade da solução proposta.

Este trabalho apresentou ainda uma solução para criação de quóruns utilizando a topologia hierárquica do VCube. Os quóruns são criados adicionando-se a maioria absoluta de processos corretos em cada cluster do hipercubo. Com isso, um quórum majoritário é formado. Resultados de simulação mostram a uniformidade dos quóruns gerados em termos de carga e tamanho, tanto em cenários com falhas, quanto nos cenários incluindo processos falhos. Além disso, como nas demais soluções, os quóruns são criados e atualizados dinamicamente à medida que o VCube atualiza a lista de processos corretos no sistema.

Como trabalhos futuros, propõe-se como um primeiro passo, investigar a aplicação de algoritmos de exclusão mútua em redes de topologia arbitrária. Posteriormente, estas mesmas soluções serão avaliadas e, quando necessário, adaptadas para execução em ambientes dinâmicos, como os oferecidos pela computação em nuvem. Como um terceiro passo, embora o problema da exclusão mútua exija um modelo de detecção de falhas perfeito, espera-se estudar a viabilidade de adaptar as soluções propostas em modelos de sistema com requisitos de sincronia mais flexíveis.

Por fim, considerando a avaliação da eficiência da utilização de quóruns discutida no Capítulo 6, além de buscar formas de reduzir o tamanho dos quóruns no VCube, espera-se implementar uma solução para a exclusão mútua distribuída baseada em quóruns. Um algoritmo de *multicast*, para os quóruns no VCube também será desenvolvido.

REFERÊNCIAS BIBLIOGRÁFICAS

Abawajy, J. e Deris, M. M. **Data replication approach with data consistency guarantee for data grid**. *IEEE Trans. Comput.*, 99(PrePrints):1–14, 2013.

Agrawal, D. e El Abbadi, A. **An efficient and fault-tolerant solution for distributed mutual exclusion**. *ACM Trans. Comput. Syst.*, 9:1–20, fev. 1991.

Almeida, C. e Veríssimo, P. **Timing failure detection and real-time group communication in quasi-synchronous systems**. In *Proc. of the 8th Euromicro Workshop on Real-Time Systems*, págs. 230–235, jun. 1996.

Amir, Y. e Wool, A. **Evaluating quorum systems over the internet**. In *FTCS*, págs. 26–, Washington, DC, USA, 1996. IEEE Computer Society.

Atreya, R., Mittal, N., e Peri, S. **A quorum-based group mutual exclusion algorithm for a distributed system with dynamic group set**. *IEEE Trans. Parallel Distrib. Syst.*, 18(10):1345–1360, 2007.

Augustine, J., Pandurangan, G., Robinson, P., e Upfal, E. **Towards robust and efficient computation in dynamic peer-to-peer networks**. In *Proc. of the 23rd ACM-SIAM Symposium on Discrete Algorithms*, págs. 551–569, 2012.

Avresky, D. R. **Embedding and reconfiguration of spanning trees in faulty hypercubes**. *IEEE Trans. Parallel Distrib. Syst.*, 10(3):211–222, mar. 1999.

Baez, J. C. **The octonions**. *Bull. Amer. Math. Soc*, 39:145–205, 2002.

Bertier, M., Arantes, L., e Sens, P. **Hierarchical token based mutual exclusion algorithms**. In *Proc. of the IEEE Int'l Symp. on Cluster Computing and the Grid*, págs. 539–546, Washington, USA, 2004.

Bertsekas, D. P., Ozveren, C., Stamoulis, G. D., Tseng, P., e Tsitsiklist, J. N. **Optimal communication algorithms for hypercubes**. *J. Parallel Distrib. Comput.*, 11:263–275, 1991.

Birman, K. P. *Building Secure and Reliable Network Applications*. Manning, Greenwich, CT, 1996.

Birman, K. P. e Joseph, T. A. **Reliable communication in the presence of failures**. *ACM Trans. Comput. Syst.*, 5(1):47–76, jan. 1987.

Boichat, R. e Guerraoui, R. **Reliable and total order broadcast in the crash-recovery model**. *J. Parallel Distrib. Comput.*, 65(4):397–413, abr. 2005.

Bona, L. C. E., Duarte Jr., E. P., e Garrett, T. **Monitoring pairwise interactions to discover stable wormholes in highly unstable networks**. In *Proc. of the 8th Int'l Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities*, págs. 1–16, 2012.

Bonomi, S., Del Pozzo, A., e Baldoni, R. **Intrusion-tolerant reliable broadcast**. Technical report, Sapienza Università di Roma,, 2013.

Bouillaguet, M., Arantes, L., e Sens, P. **Fault tolerant k-mutual exclusion algorithm using failure detector**. In *Int'l Symp. on Parallel and Distr. Comp.*, págs. 343–350, jul. 2008.

Bouillaguet, M., Arantes, L., e Sens, P. **A timer-free fault tolerant k-mutual exclusion algorithm**. In *4th Latin-American Symp. on Dep. Comp.*, LADC'09, págs. 41–48, 2009.

Bulgannawar, S. e Vaidya, N. **A distributed k-mutual exclusion algorithm**. In *Proc. of the 15th Int'l Conf. on Distr. Comp. Systems*, págs. 153–160, mai. 1995.

Cao, G., Singhal, M., Deng, Y., Rishe, N., e Sun, W. **A delay-optimal quorum-based mutual exclusion scheme with fault-tolerance capability**. In *Proc. of the The 18th Int'l Conference on Distributed Computing Systems*, ICDCS '98, págs. 444–, Washington, DC, USA, 1998. IEEE Computer Society.

Carvalho, O. e Roucairol, G. **Technical correspondence: On mutual exclusion in computer networks**. *Commun. ACM*, 26(2):146–149, fev. 1983.

Casteigts, A., Chaumette, S., e Ferreira, A. **Characterizing topological assumptions of distributed algorithms in dynamic networks**. In *Proc. of the 16th Int'l Conference on Structural Information and Communication Complexity*, SIROCCO'09, págs. 126–140, Berlin, Heidelberg, 2010. Springer-Verlag.

Casteigts, A., Flocchini, P., Quattrociocchi, W., e Santoro, N. **Time-varying graphs and dynamic networks**. In *Proc. of the 10th Int'l Conference on Ad-hoc, Mobile, and Wireless Networks*, ADHOC-NOW'11, págs. 346–359, Berlin, Heidelberg, 2011. Springer-Verlag.

Chandra, T. D. e Toueg, S. **Unreliable failure detectors for reliable distributed systems**. *Journal of the ACM*, 43(2):225–267, 1996.

Chang, Y.-I. **A hybrid distributed mutual exclusion algorithm**. *Microprocess. Microprogram.*, 41(10):715–731, mai. 1996.

Chaudhuri, P. e Edward, T. **An algorithm for k-mutual exclusion in decentralized systems**. *Comput. Commun.*, 31(14):3223–3235, set. 2008.

Correia, M., Veronese, G. S., Neves, N. F., e Veríssimo, P. **Asynchronous message-passing systems: A survey**. *Int'l Journal of Critical Computer-Based Systems*, 2: 141–161, jul. 2011.

Coulouris, G., Dollimore, J., Kindberg, T., e Blair, G. *Distributed Systems: Concepts and Design (5th Ed.)*. Addison-Wesley, Boston, MA, USA, 2011.

Cristian, F. **Understanding fault-tolerant distributed systems**. *Commun. ACM*, 34:56–78, fev. 1991.

Cristian, F. e Fetzer, C. **The timed asynchronous distributed system model**. *IEEE Trans. Parallel Distrib. Syst.*, 10(6):642–657, jun. 1999.

Dahan, S., Philippe, L., e Nicod, J. **The distributed spanning tree structure**. *Parallel and Distributed Systems, IEEE Transactions on*, 20(12):1738–1751, 2009.

Dalal, Y. **A distributed algorithm for constructing minimal spanning trees**. *Software Engineering, IEEE Transactions on*, SE-13(3):398–405, 1987.

Delporte-Gallet, C., Fauconnier, H., Guerraoui, R., e Kouznetsov, P. **Mutual exclusion in asynchronous systems with failure detectors**. *J. P. Distr. Comp.*, 65:492–505, abr. 2005.

Dixit, M., Casimiro, A., e Veríssimo, P. **Probabilistic adaptive time-aware consensus**. In *Eurosys 2009, WIP session*, Nuremberg, Germany, mar. 2009.

Dwork, C., Lynch, N. A., e Stockmeyer, L. **Consensus in the presence of partial synchrony**. *Journal of the ACM*, 35(2):288–323, 1988.

Elkin, M. **A faster distributed protocol for constructing a minimum spanning tree**. *J. Comput. Syst. Sci.*, 72(8):1282–1308, dez. 2006.

England, D., Veeravalli, B., e Weissman, J. **A robust spanning tree topology for data collection and dissemination in distributed environments**. *IEEE Trans. Parallel Distr. Syst.*, 18(5):608–620, 2007.

Eugster, P. T., Guerraoui, R., Handurukande, S. B., Kouznetsov, P., e Kermarrec, A.-M. **Lightweight probabilistic broadcast**. *ACM Trans. Comput. Syst.*, 21(4):341–374, nov. 2003.

Fetzer, C. e Cristian, F. **An optimal internal clock synchronization algorithm**. In *Proc. of the 10th Annual Conference on Systems Integrity, Software Safety and Process Security, COMPASS '95*, págs. 187 –196, jun. 1995.

Fischer, M., Lynch, N. A., e Paterson, M. **Impossibility of distributed consensus with one faulty process**. *Journal of the ACM*, 32(2):374–382, 1985.

Flocchini, P., Mesa Enriquez, T., Pagli, L., Prencipe, G., e Santoro, N. **Distributed minimum spanning tree maintenance for transient node failures**. *IEEE Trans. Comput.*, 61(3):408–414, 2012.

Fragopoulou, P. e Akl, S. **Edge-disjoint spanning trees on the star network with applications to fault tolerance**. *IEEE Trans. Comput.*, 45(2):174 –185, fev. 1996.

Freiling, F. C., Guerraoui, R., e Kuznetsov, P. **The failure detector abstraction**. *ACM Comput. Surv.*, 43:9:1–9:40, fev. 2011.

Fu, S. S., Tzeng, N.-F., e Chung, J.-Y. **Empirical evaluation of mutual exclusion algorithms for distributed systems**. *Journal of Parallel and Distributed Computing*, 60(7):785–806, 2000.

Fujita, S. **A quorum based k-mutual exclusion by weighted k-quorum systems**. *Inf. Process. Lett.*, 67(4):191–197, ago. 1998.

Gallager, R. G., Humblet, P. A., e Spira, P. M. **A distributed algorithm for minimum-weight spanning trees**. *ACM Trans. Program. Lang. Syst.*, 5:66–77, jan. 1983.

Garcia-Molina, H. e Barbara, D. **How to assign votes in a distributed system**. *J. ACM*, 32:841–860, out. 1985.

Garcia-Molina, H. e Kogan, B. **An implementation of reliable broadcast using an unreliable multicast facility**. In *SRDS'98*, págs. 101–111, 1988.

Gärtner, F. C. **A survey of self-stabilizing spanning-tree construction algorithms**. Rel. téc., Swiss Federal Institute of Technology (EPFL), 2003.

Gifford, D. K. **Weighted voting for replicated data**. In *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*, SOSP '79, págs. 150–162, New York, NY, USA, 1979. ACM.

Goldberg, A. V. e Tsioutsiouliklis, K. **Cut tree algorithms**. In *SODA '99: Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, págs. 376–385, Philadelphia, PA, USA, 1999. Society for Industrial and Applied Mathematics.

Gomory, R. E. e Hu, T. C. **Multi-terminal network flows**. *Journal of the Society for Industrial and Applied Mathematics*, 9(4):551–570, 1961.

Greve, F., Arantes, L., e Sens, P. **What model and what conditions to implement unreliable failure detectors in dynamic networks?** In *Proc. of the 3rd Int'l Workshop on Theoretical Aspects of Dynamic Distributed Systems*, TADDS '11, págs. 13–17, New York, NY, USA, 2011a. ACM.

Greve, F., Sens, P., Arantes, L., e Simon, V. **A failure detector for wireless networks with unknown membership**. In *Proc. of the 17th Int'l Conf. on P. Proc.*, págs. 27–38, Berlin, Heidelberg, 2011b. Springer-Verlag.

Guerraoui, R. e Rodrigues, L., editores. *Introduction to Reliable Distributed Programming*. Springer-Verlag, Berlin, Germany, 2006.

Guerraoui, R. e Schiper, A. **Consensus: the big misunderstanding**. In *Proc. of the 6th IEEE Distributed Computing Systems*, págs. 183–188, out. 1997.

Gusfield, D. **Very simple methods for all pairs network flow analysis**. *SIAM Journal on Computing*, 19(1):143–155, 1990.

Hadzilacos, V. e Toueg, S. *Fault-tolerant broadcasts and related problems*, págs. 97–145. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

Hadzilacos, V. e Toueg, S. **A modular approach to fault-tolerant broadcasts and related problems**. Rel. téc., Ithaca, NY, USA, 1994.

Helary, J. M., Plouzeau, N., e Raynal, M. **A distributed algorithm for mutual exclusion in an arbitrary network**. *Comput. Journal*, 31(4):289–295, ago. 1988.

Hélary, J.-M., Mostefaoui, A., e Raynal, M. **A general scheme for token- and tree-based distributed mutual exclusion algorithms**. *IEEE Trans. Parallel Distrib. Syst.*, 5(11):1185–1196, nov. 1994.

Huang, S.-T., Jiang, J.-R., e Kuo, Y.-C. **K-coteries for fault-tolerant k entries to a critical section**. In *ICDCS*, págs. 74–81, 1993.

Jalote, P. *Fault Tolerance in Distributed Systems*. Prentice Hall, 1994.

Kaashoek, M. F., Tanenbaum, A. S., Hummel, S. F., e Bal, H. E. **An efficient reliable broadcast protocol**. *Operating Systems Review*, 23:5–19, 1989.

Kakugawa, H., Fujita, S., Yamashita, M., e Ae, T. **A distributed k-mutual exclusion algorithm using k-coterie**. *Inf. Process. Lett.*, 49(4):213–218, fev. 1994.

Kephart, J. e Chess, D. **The vision of autonomic computing**. *Computer*, 36(1):41–50, 2003.

Kim, K., Mehrotra, S., e Venkatasubramanian, N. **FaReCast: Fast, reliable application layer multicast for flash dissemination**. In *ACM/IFIP/USENIX 11th International Conference on Middleware*, Middleware'10, págs. 169–190, Berlin, Heidelberg, 2010. Springer-Verlag.

Krakowiak, S. e Shrivastava, S. K., editores. *Advances in Distributed Systems, Advanced Distributed Computing: From Algorithms to Systems*, London, UK, UK, 1999.

Kruskal Jr., J. B. **On the shortest spanning subtree of a graph and the traveling salesman problem**. In *Proc. of the American Mathematical Society*, págs. 48–50, 1956.

Kshemkalyani, A. D. e Singhal, M. *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, 1 edition, mai. 2008.

Kuhn, F., Lynch, N., e Oshman, R. **Distributed computation in dynamic networks**. In *Proc. of the 42nd ACM symposium on Theory of computing*, STOC '10, págs. 513–522, New York, NY, USA, 2010. ACM.

Lamport, L. **Time, clocks and the ordering of events in a distributed system**. *Commun. ACM*, 21:558–565, jul. 1978.

Lamport, L. e Lynch, N. **Distributed computing: models and methods**. In van Leeuwen, J., editor, *Handbook of theoretical computer science (vol. B)*, págs. 1157–1199. MIT Press, Cambridge, MA, USA, 1990.

Lamport, L., Shostak, R., e Pease, M. **The byzantine generals problem**. *ACM Trans. Program. Lang. Syst.*, 4:382–401, jul. 1982.

Le Lann, G. **Distributed systems - towards a formal approach**. In *IFIP Congress*, págs. 155–160, 1977.

Leitão, J., Pereira, J., e Rodrigues, L. **HyParView: A membership protocol for reliable gossip-based broadcast**. In *DSN*, págs. 419–429, 2007.

Liebeherr, J. e Beam, T. **HyperCast: A protocol for maintaining multicast group members in a logical hypercube topology**. In Rizzo, L. e Fdida, S., editores, *Networked Group Communication*, volume 1736 of *LNCS*, págs. 72–89. Springer Berlin Heidelberg, 1999.

Lin, S.-D., Lian, Q., Chen, M., e Zhang, Z. **A practical distributed mutual exclusion protocol in dynamic peer-to-peer systems**. In *Proc. of the Third Int'l Conf. on Peer-to-Peer Systems*, IPTPS'04, págs. 11–21, Berlin, Heidelberg, 2004. Springer-Verlag.

Lipcon, T. **Quorum-based journaling in CDH4.1**, 2012. Disponível em: `http://blog.cloudera.com/blog/2012/10/quorum-based-journaling-in-cdh4-1/`. Acessado em: 12/02/2014.

Liu, T.-J., Wang, W.-C., e Tseng, C.-M. **Organize metadata servers by using quorum system**. In *IEEE/SICE Int'l Symp. Syst. Integration*, págs. 1125–1130, 2011.

Liu, Y.-J., Chou, W., Lan, J., e Chen, C. **Constructing independent spanning trees for hypercubes and locally twisted cubes**. In *10th Int'l Symposium on Pervasive Systems, Algorithms, and Networks*, ISPAN'09, págs. 17 –22, dez. 2009.

Lodha, S. e Kshemkalyani, A. **A fair distributed mutual exclusion algorithm**. *IEEE Trans. Parallel Distrib. Syst.*, 11(6):537–549, jun. 2000.

Luo, A., Wu, W., Cao, J., e Raynal, M. **A generalized mutual exclusion problem and its algorithm**. In *Parallel Processing (ICPP), 2013 42nd International Conference on*, págs. 300–309, Oct 2013.

Maekawa, M. **A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems**. *ACM Trans. Comput. Syst.*, 3:145–159, mai. 1985.

Makki, K., Banta, P., Been, K., Pissinou, N., e Park, E. **A token based distributed k mutual exclusion algorithm**. In *Proc. of the 4th IEEE Symposium on Parallel and Distributed Processing*, págs. 408 –411, dez. 1992.

Merideth, M. G. e Reiter, M. K. **Selected results from the latest decade of quorum systems research**. In *Replication*, volume 5959 of *LNCS*, págs. 185–206. Springer-Verlag, Berlin, Heidelberg, 2010.

Mostefaoui, A., Raynal, M., Travers, C., Patterson, S., Agrawal, D., e Abbadi, A. E. **From static distributed systems to dynamic systems**. In *Proc. of the 24th IEEE Symposium on Reliable Distributed Systems*, SRDS, págs. 109–118, Washington, DC, USA, 2005. IEEE Computer Society.

Mullender, S., editor. *Distributed systems (2nd Ed.)*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

Naimi, M. **Distributed mutual exclusion on hypercubes**. *SIGOPS Oper. Syst. Rev.*, 30:46–51, jul. 1996.

Naimi, M. e Thiare, O. **A distributed deadlock free quorum based algorithm for mutual exclusion**. *IJCSIS*, 11(8):7–13, ago. 2013.

Naimi, M., Trehel, M., e Arnold, A. **A $\log n$ distributed mutual exclusion algorithm based on path reversal**. *J. Parallel Distrib. Comput.*, 34:1–13, abr. 1996.

Pagani, E. e Rossi, G. P. **Reliable broadcast in mobile multihop packet networks**. In *MobiCom '97*, págs. 34–42. ACM, 1997.

Pease, M., Shostak, R., e Lamport, L. **Reaching agreement in the presence of faults**. *Journal of the ACM*, 27:228–234, abr. 1980.

Pedone, F. e Schiper, A., editores. *Modular Approach to Replication for Availability*, volume 5959 of *Lecture Notes in Computer Science*, págs. 41–59. Springer, 2010.

Pereira, J., Rodrigues, L., Pinto, A., e Oliveira, R. **Low latency probabilistic broadcast in wide area networks**. In *SRDS'04*, págs. 299–308, 2004.

Pissinou, N., Makki, K., Park, E., Hu, Z., e Wong, W. **An efficient distributed mutual exclusion algorithm**. In *Proc. of the Int'l Conf. on Parallel Processing*, volume 1, págs. 196 –203 vol.1, ago. 1996.

Prim, R. C. **Shortest connection networks and some generalizations**. *Bell System Technical Journal*, 36(6):1389–1401, nov. 1957.

Ramanathan, P. e Shin, K. **Reliable broadcast in hypercube multicomputers**. *IEEE Trans. Comput.*, 37(12):1654–1657, 1988.

Raymond, K. **A distributed algorithm for multiple entries to a critical section**. *Inf. Process. Lett.*, 30:189–193, fev. 1989a.

Raymond, K. **A tree-based algorithm for distributed mutual exclusion**. *ACM Trans. Comput. Syst.*, 7:61–77, jan. 1989b.

Raynal, M. **A simple taxonomy for distributed mutual exclusion algorithms**. *SIGOPS Oper. Syst. Rev.*, 25:47–50, abr. 1991.

Raynal, M. *Distributed Algorithms for Message-Passing Systems*. Springer, 2013.

Raynal, M. e Beeson, D. *Algorithms for mutual exclusion*. MIT Press, Cambridge, MA, USA, 1986.

Ricart, G. e Agrawala, A. K. **An optimal algorithm for mutual exclusion in computer networks**. *Commun. ACM*, 24:9–17, jan. 1981.

Rodrigues, L. A. **Fault-tolerant broadcast algorithms for the virtual hypercube topology**. In *Student Forum - DSN-W'43*, págs. 1–4, 2013.

Rodrigues, L. A. e Jansh-Pôrto, I. E. **Ampliação do framework neko para a simulação de defeitos em algoritmos distribuídos**. In *Proc. 7th I2TS*, págs. 1–8, 2008.

Rodrigues, L. A., Duarte Jr., E. P., e Arantes, L. **Exclusão mútua distribuída e robusta para k recursos compartilhados**. In *SBRC 2012 – WTF*, Ouro Preto, MG, abr. 2012.

Rodrigues, L. A., Cohen, J., Arantes, L., e Duarte Jr., E. P. **A robust permission-based hierarchical distributed k-mutual exclusion algorithm**. In *ISPDC*, págs. 1–8, 2013.

Rodrigues, L. A., Arantes, L., e Duarte Jr., E. P. **An autonomic implementation of reliable broadcast based on dynamic spanning trees**. In *EDCC*, págs. 1–12, 2014a.

Rodrigues, L. A., Duarte Jr., E. P., e Arantes, L. **Árvores geradoras mínimas distribuídas e autonômicas**. In *SBRC*, págs. 1–14, 2014b.

Rodrigues, L. A., Duarte Jr., E. P., e Arantes, L. **Uma solução autonômica para a criação de quóruns majoritários baseada no VCube**. In *WTF*, págs. 1–14, 2014c.

Romano, P. e Rodrigues, L. **An efficient weak mutual exclusion algorithm**. In *8th Int'l Symp. on Parallel and Distributed Computing*, págs. 205 –212, jul. 2009.

Ruoso, V. K. **Uma estratégia de testes logarítmica para o algoritmo Hi-ADSD**. Dissertação de Mestrado, Universidade Federal do Paraná, mai. 2013.

Sanders, B. A. **The information structure of distributed mutual exclusion algorithms**. *ACM Trans. Comput. Syst.*, 5:284–299, ago. 1987.

Saxena, P. C. e Rai, J. **A survey of permission-based distributed mutual exclusion algorithms**. *Comput. Stand. Interfaces*, 25(2):159–181, mai. 2003.

Schneider, F. B., Gries, D., e Schlichting, R. D. **Fault-tolerant broadcasts**. *Sci. Comput. Program.*, 4(1):1–15, mai. 1984.

Singhal, M. **A heuristically-aided algorithm for mutual exclusion in distributed systems**. *IEEE Trans. Comput.*, 38(5):651–662, mai. 1989.

Singhal, M. **A class of deadlock-free maekawa-type algorithms for mutual exclusion in distributed systems**. *Distributed Computing*, 4:131–138, 1991. 10.1007/BF01798960.

Singhal, M. **A dynamic information-structure mutual exclusion algorithm for distributed systems**. *IEEE Trans. Parallel Distrib. Syst.*, 3(1):121–125, jan. 1992.

Sopena, J., Arantes, L., e Sens, P. **Performance evaluation of a fair fault-tolerant mutual exclusion algorithm**. In *Proc. of the 25th IEEE Symposium on Reliable Distributed Systems*, págs. 225–234, Washington, USA, 2006. IEEE Computer Society.

Srimani, P. K. e Reddy, R. L. N. **Another distributed algorithm for multiple entries to a critical section**. *Inf. Process. Lett.*, 41(1):51–57, jan. 1992.

Suzuki, I. e Kasami, T. **A distributed mutual exclusion algorithm**. *ACM Trans. Comput. Syst.*, 3:344–349, nov. 1985.

Tamhane, S. A. e Kumar, M. **Token based algorithm for supporting mutual exclusion in opportunistic networks**. In *Proc. of the 2nd Int'l Workshop on Mobile Opportunistic Networking*, págs. 126–134, New York, NY, USA, 2010. ACM.

Thomas, R. H. **A majority consensus approach to concurrency control for multiple copy databases**. *ACM Trans. Database Syst.*, 4(2):180–209, jun. 1979.

Turek, J. e Shasha, D. **The many faces of consensus in distributed systems**. *Computer*, 25(6):8–17, 1992.

Urbán, P., Défago, X., e Schiper, A. **Neko: A single environment to simulate and prototype distributed algorithms**. *Journal of Inf. Science and Eng.*, 18(6):981–997, nov. 2002.

Veríssimo, P. **Travelling through wormholes: a new look at distributed systems models**. *SIGACT News*, 37(1):66–81, mar. 2006.

Veríssimo, P. e Almeida, C. **Quasi-synchronism: a step away from the traditional fault-tolerant real-time system models**, 1995.

Veríssimo, P. e Rodrigues, L., editores. *Distributed Systems for System Architects*. Kluwer Academic Publishers, New York, NY, USA, 2001.

Vukolić, M. **The origin of quorum systems**. *Bulletin of the EATCS*, 101:125–147, 2010.

Walter, J. E., Welch, J. L., e Vaidya, N. H. **A mutual exclusion algorithm for ad hoc mobile networks**. *Wirel. Netw.*, 7(6):585–600, nov. 2001.

Wang, Y., Fan, J., Jia, X., e Huang, H. **An algorithm to construct independent spanning trees on parity cubes**. *Theoretical Computer Science*, (0):–, 2012.

Widder, J. e Schmid, U. **The theta-model: achieving synchrony without clocks**. *Distributed Computing*, 22:29–47, 2009.

Wu, J. **Optimal broadcasting in hypercubes with link faults using limited global information**. *J. Syst. Archit.*, 42(5):367–380, 1996.

Yan, Y., Zhang, X., e Yang, H. **A fast token-chasing mutual exclusion algorithm in arbitrary network topologies**. *J. Parallel Distrib. Comput.*, 35(2):156–172, jun. 1996.

Yang, Z., Li, M., e Lou, W. **R-code: Network coding based reliable broadcast in wireless mesh networks with unreliable links**. In *GLOBECOM'09*, págs. 1–6, 2009.

Apêndices

APÊNDICE A

PUBLICAÇÕES

Neste anexo estão listadas as publicações obtidas durante o desenvolvimento da tese, bem como as contribuições no âmbito do Grupo de Pesquisa LARSIS - Laboratório de Redes e Sistemas Distribuídos.

A.1 Trabalhos Publicados no Âmbito da Tese

Trabalhos completos publicados em anais de congressos

1. **RODRIGUES, L. A.**; DUARTE JR, Elias; ARANTES, Luciana. *Árvores Geradoras Mínimas Distribuídas e Autonômicas.* In: XXXII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC), 2014, Florianópolis, SC.

2. **RODRIGUES, L. A.**; ARANTES, Luciana; DUARTE JR, Elias. *Uma Solução Autonômica para a Criação de Quóruns Majoritários Baseada no VCube.* In: XV Workshop de Testes e Tolerância a Falhas (WTF), 2014, Florianópolis, SC.

3. **RODRIGUES, L. A.**; ARANTES, Luciana; DUARTE JR, Elias. *An Autonomic Implementation of Reliable Broadcast Based on Dynamic Spanning Trees.* In: X European Dependable Computing Conference (EDCC), 2014, Newcastle upon Tyne, UK.

4. **RODRIGUES, L. A.**; COHEN, JAIME; ARANTES, Luciana; DUARTE JR, Elias. *A Robust Permission-Based Hierarchical Distributed k-Mutual Exclusion Algorithm.* In: International Symposium on Parallel and Distributed Computing (ISPDC), 2013, Bucharest, Romênia.

5. **RODRIGUES, L. A.**; DUARTE JR, Elias; ARANTES, Luciana. *Exclusão Mútua Distribuída e Robusta para k Recursos Compartilhados.* In: XIV Workshop de Testes e Tolerância a Falhas (WTF), 2012, Ouro Preto, MG.

Resumos expandidos publicados em anais de congressos

1. **RODRIGUES, L. A.**; DUARTE JR, Elias; ARANTES, Luciana. *Fault-Tolerant Broadcast Algorithms for the Virtual Hypercube Topology.* In: IEEE/IFIP 43rd Int'l Conference on Dependable Systems and Networks (DSN-Student Forum), 2013, Budapeste, Hungria.

Trabalhos em Periódicos

1. Um artigo descrevendo a contribuição completa do trabalho está em preparação e será submetida ao *ACM Transactions on Autonomous and Adaptive Systems.*

A.2 Trabalhos Publicados no Âmbito do Grupo de Pesquisa

Em paralelo com o trabalho descrito neste documento, outro esforço de pesquisa tem sido realizado no desenvolvimento de algoritmos paralelos de árvores de cortes.

Seja $G = (V, E)$ um grafo não direcionado cujas arestas estão associadas a capacidades por uma função $c : E(G) \to \mathbb{Z}_+$. Um *corte* de G é uma bipartição de $V(G)$ em $\{X, V - X\}$. Uma aresta $\{u, v\} \in E(G)$ cruza o corte se $u \in \{X\}$ e $v \in \{V - X\}$. A capacidade de um corte é o somatório das capacidades das arestas que cruzam o corte. Considere o problema de calcular a capacidade dos cortes entre todos os pares de vértices do grafo G. A solução trivial consiste em executar $\binom{n}{2}$ algoritmos de corte mínimo (ou fluxo máximo), um para cada par de vértices. Em 1961, Gomory e Hu Gomory e Hu (1961) mostraram que o cômputo de apenas $n - 1$ fluxos máximos é suficiente. Posteriormente, Gusfield Gusfield (1990) propôs uma segunda solução que também resolve o problema com $n - 1$ cálculos de fluxo máximo. As duas soluções constroem uma árvore capacitada sobre o conjunto de vértices do grafo que representa os valores das conectividades locais para todos os pares de vértices. Estas árvores são conhecidas como árvores de cortes.

As árvores de cortes são utilizadas na solução de inúmeros problemas combinatórios em áreas como particionamento de grafos, conectividade e roteamento. Apesar da importância, o único estudo experimental sobre árvores de cortes foi publicado por Goldberg e Tsioutsiouliklis Goldberg e Tsioutsiouliklis (1999), mas nenhuma implementação paralela foi abordada até o momento. Neste sentido, foram implementadas duas soluções paralelas para o algoritmo de Gusfield, uma em OpenMP (*Open Multi-Processing*) e outra em MPI (*Message Passing Interface*), e uma versão em MPI para o algoritmo de Gomory-Hu.

Os resultados completos estão publicados nas referências listadas na sequência. Um artigo para um periódico está em fase final de elaboração.

Trabalhos completos publicados em anais de congressos

1. COHEN, JAIME; **RODRIGUES, L. A.**; DUARTE JR, Elias. *A Parallel Implementation of Gomory-Hu's Cut Tree Algorithm.* In: International Symposium on Computer Architecture and High Performance Computing, 2012, New York, US.

2. COHEN, JAIME ; **RODRIGUES, L. A.**; GUEDES, ANDRÉ L. P.; CARMO, RENATO; DUARTE JR, Elias. *Parallel Implementations of Gusfield's Cut Tree Algorithm.* In: 11th International Conference on Architectures and Algorithms for Parallel Processing (ICA3PP), 2011, Melbourne, Austrália. Lecture Notes in Computer Science, 2011. v. 7016. p. 258-269.

3. COHEN, JAIME; **RODRIGUES, L. A.**; DUARTE JR, Elias. *Improved Parallel Implementations of Gusfield's Cut Tree Algorithm.* In: XII Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD-SSC), 2011, Vitória, ES.

Resumos expandidos publicados em anais de congressos

1. **RODRIGUES, L. A.**; COHEN, JAIME; DUARTE JR, Elias. *A MPI Implementation of Gusfield's Algorithm to Generate Cut Trees.* In: Latin-American Symposium on Dependable Computing (LADC), 2011, São José dos Campos, SP.