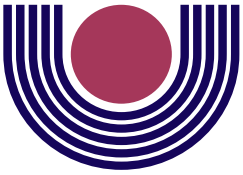


Unioeste - Universidade Estadual do Oeste do Paraná
CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS
Colegiado de Ciência da Computação
Curso de Bacharelado em Ciência da Computação

Uma Solução de Exclusão Mútua Para Sistemas Distribuídos baseada em Quóruns no VCube

Igor Steuck Lopes

CASCAVEL
2016

IGOR STEUCK LOPES

Uma Solução de Exclusão Mútua Para Sistemas Distribuídos baseada em Quóruns no VCube@

Monografia apresentada como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação, do Centro de Ciências Exatas e Tecnológicas da Universidade Estadual do Oeste do Paraná - Campus de Cascavel

Orientador: Prof. Dr. Luiz Antonio Rodrigues

CASCAVEL
2016

Igor Steuck Lopes

Uma Solução de Exclusão Mútua Para Sistemas Distribuídos baseada em Quóruns no VCube

Monografia apresentada como requisito parcial para obtenção do Título de Bacharel em Ciência da Computação, pela Universidade Estadual do Oeste do Paraná, Campus de Cascavel, aprovada pela Comissão formada pelos professores:

Prof. Dr. Luiz Antonio Rodrigues (Orientador)
Colegiado de Ciência da Computação,
UNIOESTE

Prof. Dr. Guilherme Galante
Colegiado de Ciência da Computação,
UNIOESTE

Prof. Dr. Márcio Seiji Oyamada
Colegiado de Ciência da Computação,
UNIOESTE

Cascavel, 9 de março de 2017

Resumo

Em sistemas distribuídos um ou mais processos podem vir a concorrer por um mesmo recurso e, para que todos obtenham acesso ao recurso desejado de forma exclusiva, utilizam-se algoritmos de exclusão mútua. Nestes sistemas podem ocorrer falhas e portanto, além da necessidade de um sistema que monitore o estado de cada processo, é preciso que todos os algoritmos utilizados também sejam tolerantes à falhas. Para tanto, foi proposto um algoritmo de exclusão mútua baseado em quóruns que usa do sistema de diagnóstico VCube para monitorar os processos e gerar, a partir dos processos corretos, quóruns. Para se avaliar o desempenho do algoritmo (número de mensagens enviadas por seção crítica obtida), executou-se testes em diferentes cenários com três geradores de quóruns: gerador de qúoruns em grade, gerador de quóruns em árvore e o gerador de quóruns no VCube O algoritmo garantiu a exclusão mútua em todos os cenários para todos os geradores de quóruns e os resultados mostraram que cada tipo de gerador apresenta um desempenho mais vantajoso a depender do cenário de teste, sendo o de quóruns em árvore que resultou em um menor número de mensagens enviadas para todos os cenários.

Palavras-chave: exclusão mútua, gerador de quóruns, tolerância à falhas, sistemas distribuídos

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

Um sistema computacional distribuído é uma coleção de subsistemas que trabalham de forma cooperativa na resolução de problemas que são insolúveis ou inviáveis por apenas um membro desse sistema. Em um sistema desse tipo, várias unidades (cada qual com memória, processador e disco de armazenamento próprios) estão distribuídas em um dado espaço geográfico e estão conectadas por uma rede de comunicação. Apesar de ser composto por unidades independentes, um sistema distribuído é apresentado ao usuário como um sistema único, e utiliza de diferentes camadas de abstração para reduzir a complexidade de uso [Kshemkalyani e Singhal 2008] .

Ao funcionar como um único sistema, todos os recursos que cada unidade possui passam a ser disponibilizado para os diversos processos do sistema de forma compartilhada, dando origem a uma série de problemas de concorrência por estes recursos. Os algoritmos de exclusão mútua oferecem uma solução para esses problemas de concorrência.

A exclusão mútua é um modo de assegurar que quando um recurso compartilhado estiver sendo utilizado por um processo, os outros processos que também necessitarem deste recurso aguardem pela liberação do mesmo. Segundo Tanenbaum [Tanenbaum e Steen 2007] em um sistema com um único processador a exclusão mútua entre processos é garantida com o uso de semáforos, monitores e construções similares. Entretanto, Kshemkalyani e Singhail [Kshemkalyani e Singhal 2008] explicam que em um sistema distribuído o uso de variáveis compartilhadas (semáforos) ou um *kernel* local não podem ser utilizados para implementar a exclusão mútua, sendo usada então uma abordagem distribuída de troca de mensagens entre processos. Neste caso, as soluções podem ser divididas em duas abordagens: algoritmos de passagem de tokens e algoritmos de pedido de permissão.

Nas abordagens baseadas em pedido de permissão, um processo que deseja acessar o recurso envia uma mensagem de requisição a todos os demais processos do sistema e aguarda pelas respostas. Uma variação desta abordagem utiliza sistemas de *quóruns*, no qual a solicitação é enviada apenas para um subgrupo de processos [Kshemkalyani e Singhal 2008].

Nas abordagens não baseadas em passagem de *token*, os processos do sistema trocam mensagens entre si por duas ou mais rodadas para determinar qual dos processos deve ganhar acesso ao recurso compartilhado. Um processo ganha acessso ao recurso compartilhado quando uma asserção, definida em variáveis locais do processo, se torna verdadeira. Nesse caso a exclusão mútua é forçada a partir do fato de que uma asserção só se torna verdadeira em um processo de cada vez, em qualquer ponto do tempo de execução do sistema.

Nos algoritmos baseados em passagem de tokens, a permissão de acesso a seção crítica é materializada em um *token*, que é um objeto único dentro do sistema. Existem duas abordagens para gerenciar a passagem dos tokens: na primeira o *token* é passado de processo em processo seguindo uma ordem lógica e na segunda o processo deve requisitar o *token* aos outros processos.

A abordagem escolhida nesta proposta de trabalho é baseada em sistemas de quóruns. Um sistema de quóruns é uma coleção de subconjuntos de nós, chamados de quóruns com a propriedade de que cada par de subconjuntos possui uma interseção não vazia. Kshemkalyani e Singhail [Kshemkalyani e Singhal 2008] definem as seguintes propriedades para quóruns em um subconjunto: a propriedade da interseção define que cada quórum do sistema deve ter ao menos um elemento em comum com cada um dos outros quóruns e a propriedade da minimalidade define que um quórum não deve conter todos os elementos de um outro quórum.

Para cada elemento $x \in$ ao quórum g, se x quer acesso a algum recurso compartilhado, ele primeiramente irá requisitar a permissão de todos os elementos do quórum. Dada a propriedade de interseção, cada elemento em g possui em um elemento em comum com cada subconjunto h no sistema. Cada elemento em comum enviam permissão de recursos somente para um elemento por vez, o que garante então a exclusão mútua no sistema. Assim, sistema de quóruns podem ser utilizados no desenvolvimento de algoritmos para exclusão mútua em um ambiente distribuído.

Diferentes algoritmos de exclusão mútua baseada em quóruns tem sido propostos como ve-

mos em Maekawa [Maekawa e Mamoru 1985], que utiliza-se da teoria de planos projetivos, e Agrawal e El Abbadi [Divyakant e Amr 1991], que definiram uma estrutura de quóruns em árvore. O trabalho de Rodrigues et al [Rodrigues, Jr e Arantes] apresenta uma solução do quórum majoritários para sistemas sujeitos a falhas de crash. Os processos falhos são detectados por um mecanismo de monitoramento denominado VCube [P., E. e Ruoso 2014].

Este trabalho propõe um algoritmo de exclusão mútua distribuída tolerante a falhas. Este algoritmo foi feito especificamente para fazer uso das características do gerador de quóruns proposto por Rodrigues [Rodrigues 2014].

O texto está organizado nos seguintes capítulos. O capítulo 2 aborda as definições de sistemas distribuídos, modelos de sistemas distribuídos e os tipos de modelo de falhas existentes. No Capítulo 3 aborda-se o conceito de exclusão mútua, que características um algoritmo deve apresentar para garantir a exclusão mútua seguido dos tipo de algoritmo, os principais algoritmos e o algoritmo escolhido para ser utilizado neste trabalho. O Capítulo 4 apresenta a topologia virtual VCube, um algoritmo gerador de quóruns majoritários nesta topologia e, por fim, o algoritmo de exclusão mútua proposto neste trabalho. O Capítulo 5 apresenta a avaliação experimental, as ferramentas utilizadas nesta avaliação e os resultados obtidos ao utilizar o algoritmo proposto com diferentes geradores de quórum. Por fim, o Capítulo 6 contém a conclusão e sugestões de trabalhos futuros.

Capítulo 2

Sistemas Distribuídos

Os sistema distribuídos já são a estrutura base de muitos dos grandes sistemas modernos como a World Wide Web (WWW), sistemas bancários e sistemas de controle de tráfego aéreo. Esta seção apresenta as definições de um sistema distribuído, os modelos de sistema existentes seguido da definição e apresentação dos modelos de falhas.

2.1 Definição

Muitas definições de sistemas distribuídos são dadas na literatura porém ainda não chegou-se a um consenso geral sobre todas as características que definem um sistema distribuído. Tanenbaum [Tanenbaum e Bos 2014] define um sistema distribuído como uma coleção de computadores independentes que aparecem para o usuário como um sistema coerente. Um sistema distribuído opera em uma camada de software que cria uma camada entre as aplicações de usuário e os sistemas operacionais de cada nó do sistema, à essa camada pode-se dar o nome de *middleware* [Tanenbaum e Bos 2014]. A camada de *middleware* oferece estruturas de dados e operações que permite que usuários e processos operem e interajam de uma maneira consistente. O *middleware* pode ainda ser visto, até certo ponto, como o sistema operacional do sistema distribuído [Tanenbaum e Bos 2014].

Coulouris et al. [Coulouris, Dollimore e Kindberg 2005] definem que sistema distribuído é um sistema no qual componentes de hardware ou software localizados em uma rede de computadores se comunicam e coordenam suas ações unicamente através da troca de mensagens. Podemos ainda definir as seguintes características para um sistema distribuído:

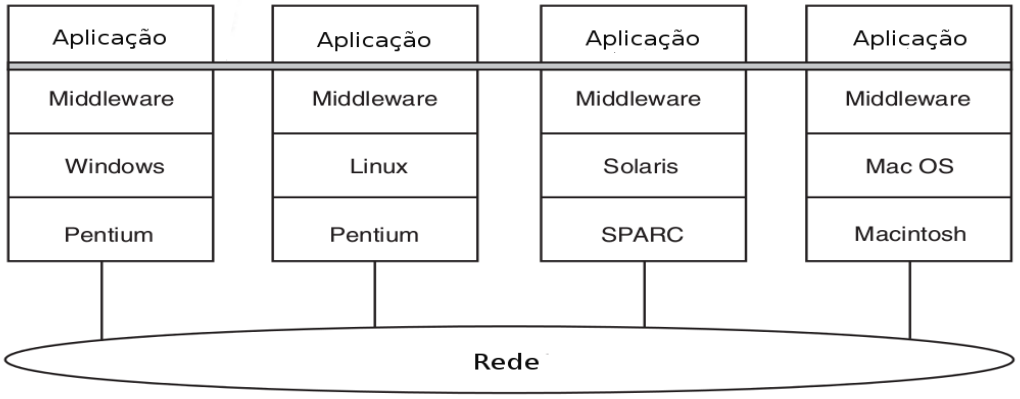


Figura 2.1: Camadas de um Sistema Distribuído [Tanenbaum e Steen 2007]

- **Concorrência:** a capacidade de um sistema lidar com recursos compartilhados.
- **Ausência de um Relógio Global:** cada nodo do sistema executa um relógio próprio.
- **Falhas Independentes:** cada componenente do sistema pode falhar de modo independente, permitindo que o resto do sistema continue em execução.

Kshemkalyani e Singhal [Kshemkalyani e Singhal 2008] definem um sistema distribuído como uma coleção de sistemas, em sua maioria, autônomos que se comunicam por uma rede e, além da ausência de um relógico global, definem as seguintes características:

- **Memória não compartilhada:** apesar de proverem memória, via abstração, para todo o sistema, cada nodo possui memória própria.
- **Separação Geográfica**: quanto mais distribuídos geograficamente, mais representativo é o sistema distribuído.
- **Autonomia e Heterogeneidade**: os processadores podem trabalhar em diferentes velocidades, com diferentes sistemas operacionais e ainda sim operar como um só sistema.

Segundo Raynal [M. 1991], um sistema distribuído pode ainda ser definido como um sistema composto de processos que comunicam-se entre si e supostamente cooperam por uma meta. Em este sistema, cada processo se comunica com os demais via troca de mensagens por um canal de comunicação assumidamente seguro. Em uma definição mais formal, partindo da

definição de que um sistema distribuído é composto por processos, define-se que o mesmo é composto por um conjunto $\prod$ de $n > 2$ processos $\{p_1, ..., p_n\}$ no qual cada $p_i, 1 \leqslant i \geqslant n$, representa um processo distinto e executa de forma sequencial.

2.2 Modelos de Sistema Distribuído

Sistemas distribuídos, como definido na Seção 2.1, são processos executando em diferentes nós de um sistema geograficamente distribuído que se comunicam por uma rede, e, uma vez que são composto de diferentes camadas, foram definidos diferentes modelos para esse tipo de sistema.

Em uma definição mais abrangente, um sistema distribuído possui três tipos de modelos: modelos físicos, modelos arquiteturais e modelos fundamentais [Coulouris, Dollimore e Kindberg 2005]. Um modelo físico especifica as particularidades dos elementos de hardware de um sistema, e não se preocupa quanto a computação e as formas de comunicação aplicadas. Em um modelo arquitetural são levadas em consideração características de projeto de um sistema distribuído, i.e., gerenciabilidade, custo, adaptabilidade, confiabilidade, etc. Os modelos fundamentais por sua vez apresentam propriedades mais específicas quanto as características, de comunicação, falha e risco de um sistema.

Jalote [Jalote 1994] apresenta uma definição sucinta de que sistemas distribuídos estão divididos em duas categorias: modelo físico (componentes físicos do sistema) e modelo lógico(definidos do ponto de vista de processamento). O modelo de sistemas síncronos, assíncronos e parcialmente síncronos são tipos de modelos lógicos.

2.2.1 Sistemas Distribuídos Síncronos

Um sistema distribuído é dito Síncrono quando em cada nó do sistema há um relógio local, cujas taxas de variação possuem limites conhecidos, que determina a ordem das operações a serem executadas. Em um Sistema Distribuído Síncrono, cada processo executa uma sequência de rodadas, sendo cada rodada um período específico do relógio global. As mensagens entre os processos são recebidas na mesma rodada em que são enviadas. Esse tipo de sistema é difícil de ser implementado em casos onde os nós estão geograficamente distantes, dada a dificuldade de se garantir os limites temporais, o que requer um controle rigoroso de aspectos de hardware

e software do sistema [Rodrigues 2014].

2.2.2 Sistemas Distribuídos Assíncronos

Um sistema distribuído assíncrono é um sistema sem uma uma taxa de variação de relógio, este por sua vez local, para determinar a ordem das operações e pode ser chamado de sistema *time-free* [Raynal 2013]. O progresso dos diferentes processos em um sistema distribuído assíncrono e as trocas de mensagens realizadas pelos mesmos estará relacionada ao processamento que acontecer em cada um dos diferentes nós do sistema, leve o tempo que for.

2.2.3 Sistemas Distribuídos Parcialmente Síncronos

Os sistemas apresentados nas Seções 2.2.1 e 2.2.2 possuem diferentes formas para entrar em consenso quanto a eventos que ocorrem no sistema. Em casos de falhas, por exemplo, um sistema dito síncrono possui o recurso de *timeouts*, e consegue definir trivialmente quando um nó do sistema falhou. Já em um sistema dito assíncrono, não há maneira trivial de determinar se um dado processo está demorando para responder ou se o mesmo falhou. Portanto, dada a dificuldade de implementação de sistemas síncronos e dos problemas de consenso em sistemas assíncronos [Fischer et al. 1985], novas soluções foram propostas que tentam atuar entre os dois modelos de interação, à estas soluções deu-se o nome de Sistemas Parcialmente Síncronos [Rodrigues 2014].

Sistemas parcialmente síncronos podem ser classificados em dois modelos: um modelo com limites temporais conhecidos somente após um tempo de estabilização (de duração desconhecida) e um modelo com limites temporais existentes mas desconhecidos [Dwork e Lynch 1988].

2.3 Modelos de Falhas

Na maioria dos sistemas distribuídos os esquemas de tolerância a falhas estão relacionados as falhas na parte física do sistema, i.e., nos nodos ou na rede de comunicação do sistema e podem receber a classificação da Figura 2.2 conforme o comportamento dos mesmos nos casos de falha [Jalote 1994].

Como ilustado na Figura 2.2, o tipo básico de falha em um sistema distribuído é a falha de colapso (*crash*). As falhas de colapso ocorrem quando um sistema para sua execução ou

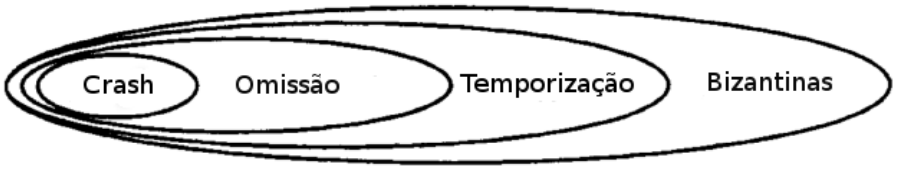


Figura 2.2: Modelos de Falha
[Jalote 1994]

perde seu estado intern o[Jalote 1994]. O fato do componente do sistema parar bruscamente sua execução torna este um tipo de falha de difícil detecção em sistemas assíncronos uma vez que não há meio de descobrir se o componente parou de funcionar ou se apenas demora a responder. Sistemas síncronos fazem o uso de *timeouts*, evitando assim o problema de detecção [Coulouris, Dollimore e Kindberg 2005].

As falhas por omissão ocorrem quando um sistema deixa de responder a algumas entradas [Jalote 1994], isto é, quando o processo ou o canal de comunicação deixa de executar o que se espera que seja executado [Coulouris, Dollimore e Kindberg 2005]. Como mostrado na Figura 2.2, as falhas de colapso (crash) são também classificadas como falhas por omissão.

Segundo Jalote [Jalote 1994], quando um processo responde mais rápido ou mais lento do que o esperado, ocorreu uma falha de temporização. Couloris et al [Coulouris, Dollimore e Kindberg 2005] especificam que falhas de temporização ocorrem somente em sistemas síncronos onde são definidos limites para execução de um processo, envio de mensagem e para taxa de variação de relógio. Em sistemas assíncronos não é possível confirmar que houve uma falha de temporização, uma vez que não há como garantir que esta falha ocorreu.

Quando um tipo de erro aleatório e inesperado ocorre em um sistema distribuído, dá-se a nome de falha bizantina ou arbitrária. Falhas bizantinas englobam todos os tipos de falha existentes, incluindo as maliciosas (Figura 2.2) isso deriva da arbitrariedade com a qual esse tipo de falha ocorre.

Singhal e Kshemkalyani [Kshemkalyani e Singhal 2008] separam ainda os modelos de falha pela origem da ocorrência delas, isto é, em qual parte do sistema distribúido uma dada falha ocorreu. Assim, classificam os modelos de falha em duas diferentes categorias: falhas de pro-

cesso e falhas de comunicação. Os modelos de falha de processo, em ordem de severidade, incluem *fail-stop*, *crash*, omissão de recebimento, omissão de envio e bizantinas. As falhas de processo ocorrem em sistemas síncronos e assíncronos, e para sistemas síncronos ainda são inclusas, as falhas de temporização como violação da taxa de variação do relógio e violação do tempo levado em um passo da execução. Por sua vez os modelos de falhas de comunicação incluem falhas de *crash*, falhas de omissão e falhas bizantinas. As falhas de comunicação também ocorrem tanto em sistemas síncronos quanto em sistemas assíncronos.

Capítulo 3

Exclusão Mútua

O problema da exclusão mútua ocorre quando diferentes processos em um dado Sistema Distribuído requerem acesso a um mesmo recurso do sistema. Esta sessão apresenta as três abordagens de implementação de exclusão mútua, começando com algoritmos de pedido de permissão, seguido de algoritmos de passagens de tokens e finalizando com algoritmos baseados em sistema de quóruns.

3.1 Definição

Os processos em um sistema distribuído precisam, frequentemente, coordenar suas atividades no sistema. Quando um dado número de processos disputam o acesso à um recurso compartilhado ou à uma coleção de recursos compartilhados, então é necessário que haja exclusão mútua para garantir que que os recursos serão acessados de forma consistente [Coulouris, Dollimore e Kindberg 2005].

Esse problema é o mesmo problema de concorrência que encontramos na área de sistemas operacionais, onde dois ou mais processos tentam obter acesso à chamada seção crítica. Porém, diferentemente de um sistema operacional, um sistema distribuído não pode utilizar de variáveis compartilhadas (semáforos) ou um kernel local para implementar a exclusão mútua [Kshemkalyani e Singhal 2008]. A solução é uma exclusão mútua distribuida que se baseia somente na passagem de mensagens entre os processos do sistema [Coulouris, Dollimore e Kindberg 2005].

Para que um dado algoritmo seja considerado um algoritmo de exclusão mútua, distribuída ou não, é preciso que uma série de propriedades sejam satisfeitas. Kshemkalyani e Singhal

[Kshemkalyani e Singhal 2008] definem três propriedades: *safety*, que garante que somente um processo por vez pode ter acesso a sessão crítica (recurso); *liveness*, que define a ausência de *deadlocks* e *starvation*; e *fairness*, que no contexto de algoritmos de exclusão mútua define que os pedidos de acesso à seção crítica serão executados em ordem de chegada. Coulouris et al. [Coulouris, Dollimore e Kindberg 2005] também definem as mesmas três propriedades para a exclusão mútua. Existem definições que não consideram a propriedade de *fairness* como um requisito, definindo *safety* e juntando, de certa forma, as características da propriedade de *fairness* à propriedade de *liveness*, enfatizando inclusive que garantir que o sistema seja *starvation-free* é mais importante do que garantir a ausência de *deadlocks* [Raynal 2013].

Quanto à classificação dos algoritmos de exclusão mútua, Raynal [M. 1991] define duas abordagens (podendo ser também três abordagens, como definido por Kshemkalyani e Singhal) [Kshemkalyani e Singhal 2008]: algoritmos baseados em pedido de permissão (*permission-based*) e algoritmos baseados em passagem de *token* (*token-based*). Embora algoritmos baseados em quóruns (*quorum-based*) façam parte dos algoritmos baseados em passagem de permissão [Maekawa e Mamoru 1985], aborda-se esse tipo de algoritmo na seção 3.4.

3.2 Algoritmos Baseados em Pedido de Permissão

A ideia dos algoritmos baseados em pedido de permissão é bem simples: quando um processo quer ter acesso a seção crítica ele envia uma mensagem aos outros processos. Se os outros processos não estiverem concorrendo por este acesso, eles enviam uma mensagem dando permissão ao requisitante, do contrário, algum tipo de fila de prioridades é formada para gerenciar as várias requisições [M. 1991]. Esta seção apresenta os principais algoritmos baseados em pedidos de permissão.

3.2.1 Algoritmo de Lamport

Lamport [L. 1978] especificou o que veio a ser o primeiro algoritmo de exclusão mútua distribuído. Este algoritmo foi criado com o propósito de ilustrar um esquema de sincronização de relógio [Kshemkalyani e Singhal 2008] e define que as requisições de acesso a seção crítica deve ser processada conforme a ordem crescente do *timestamp*, ou a marca de tempo, definida conforme o relógio lógico do sistema.

Todos os processos possuem uma fila local de requisições. Quando um dado processo p_i quer ter acesso a seção crítica, este processo envia uma mensagem REQUEST(p_i, ts_i) para todos os outros processos via *broadcast* e coloca sua própria requisição em sua fila local. Quando um processo recebe uma mensagem de REQUEST, ele a coloca na fila de requisições, atualiza o relógio local e responde com uma mensagem de REPLY(p_j, ts_j) contendo seu *timestamp* atualizado. Para que um dado processo p_i consiga acesso a seção crítica é necessário que sua requisição seja a primeira em sua fila local e que todos os REPLY contenham *timestamp* maior do que o seu. Ao sair da sessão crítica, um processo p_i, envia por *broadcast* uma mensagem de RELEASE(p_i, ts_i) para todos os processos do sistema. Quando um processo p_j recebe uma mensagem de RELEASE de um processo p_i ele remove o REQUEST de p_i da fila de requisições. O número total de mensagens enviadas em uma rodada requisição-resposta-liberação é de $3(n-1)$ mensagens.

3.2.2 Algoritmo de Ricart-Agrawala

O algoritmo proposto por Ricart e Agrawala [Ricart e Agrawala 1981] faz o uso de apenas dois tipos de mensagem: REQUEST e REPLY, e aprimora a solução proposta por Lamport reduzindo o número total de mensagens por rodada para $2(n-1)$. Esta solução também utiliza-se de um relógio lógico e atribui um *timestamp* as mensagens enviadas entre os processos. Aqui, cada processo mantém, ao invés de uma fila de requisições, uma lista de requisições pendentes.

Uma rodada de execução deste algoritmo se dá da seguinte forma: se um dado processo p_i quer acessar a seção crítica, ele envia por *broadcast* uma mensagem de REQUEST(p_i, ts_i) para todos os outros processos. Quando um processo p_j receber uma mensagem de REQUEST de um processo p_i e ele não estiver requisitando ou executando a seção crítica ou se ele estiver requisitando mas o *timestamp* de p_i for menor, ele envia uma mensagem de REPLY para p_i, caso contrário ele retém a resposta. Um processo p_i consegue acesso a seção crítica se receber um REPLY de todos os processos para o qual ele enviou REQUEST. Ao sair da seção crítica p_i envia uma mensagem de REPLY para todos os processos da lista de requisições pendentes, permitindo assim que um próximo processo acesse a seção crítica.

3.2.3 Algoritmo de Singhal

O algoritmo proposto por Singhal [Singhal 1992] trabalha de maneira dinâmica e, diferente dos algoritmos propostos por Lamport e Ricart-Agrawala, explora a mudança nos estados dos processos ao longo do tempo de execução.

Singhal propõe um conjunto de estruturas que irão armazenar as informações necessárias para que o algoritmo seja dinâmico. Cada processo p_i do conjunto de processos $\{p_1, ..., p_n\}$ de um sistema distribuído contém dois conjuntos: o conjunto de requisições R_i que contém os processos dos quais p_i precisa obter permissão para acessar a seção crítica, e o conjunto I_i (*inform-set*) que contém os processos para os quais o processo p_i deve, após ter acessado a seção crítica, enviar sua permissão de acesso à seção crítica. Os processos possuem também um relógio local C_i que é atualizado segundo as regras definidas por Lamport e é utilizado para definir o *timestamp* das mensagens de requisição. Assim como no algoritmo de Lamport, quanto menor o valor do *timtestamp* de uma requisição, maior será a prioridade da mesma. Os processos possuem ainda três variáveis booleanas referente ao estado atual do mesmo: *requesting* e *executing* que são verdadeiras somente se um processo estiver, respectivamente, requisitando ou executando a seção crítica e *My_priority* que é verdadeira se a prioridade da requisição do próprio processo for maior do que a das requisições que chegam.

No início da execução todo processo tem suas variáveis de requisição e execução com valor falso, seu conjunto R_i contendo todos os processos do sistema que tenham um identificador i menor do o seu e tem o conjunto I_i contendo somente ele mesmo. Se um processo p_i quer acessar a seção crítica ele muda seu estado para *requesting*, envia um REQUEST para todos os processos em R_i. Caso sua prioridade seja maior do que a prioridade dos REQUEST de p_j que chegam ele adiciona p_j à I_i. Caso a prioridade do REQUEST de p_j seja maior, p_i irá enviar um REPLY para p_j, e se $p_j \notin R_i$, enviará também um REQUEST e adicionará o processo p_j à R_i. Quando um processo p_i requisitante recebe uma mensagem de REPLY do processo p_j, p_j é removido do conjunto R_i. No caso do processo p_i estar executando a seção crítica, todo REQUEST de p_j que chegar durante este tempo será adicionado à I_i. Ao sair da seção crítica o processo p_i irá enviar uma mensagem de REPLY para cada processo p_j no conjunto I_i, removendo-o após o envio e adicionando-o em seguida à R_i. Por fim, um processo p_i que não estiver nem requisitando nem executando a seção crítica irá responder com um REPLY a

qualquer REQUEST de p_j, adicionando p_j à R_i.

O número de mensagens deste algoritmo pode variar de 0 a $3(n-1)/2$, a depender do número de processos requisitando acesso a seção crítica simultaneamente, e da posição dos mesmos no conjunto R_i.

3.3 Algoritmos Baseados em Passagem de Tokens

Nos algoritmos baseados em passagem de tokens, a permissão de acesso a seção crítica é materializada em um *token*, que é um objeto único dentro do sistema. Existem duas abordagens para gerenciar a passagem dos tokens. Na primeira o *token* é passado de processo em processo, percorrendo uma estrutura lógica em anel, se o processo não requer acesso a seção crítica, ele passa o *token* adiante. Na segunda abordagem, o processo deve requisitar o *token* aos outros processos do sistema, e em caso de requisições conflitantes, diversas maneiras de gerenciar a prioridade foram propostas [M. 1991]. Esta seção apresenta dois dos principais algoritmos de passagem de *token*.

3.3.1 Algoritmo de Suzuki-Kassami

O algoritmo de Suzuki-Kassami [Suzuki e Kasami 1985] tem uma ideia bem simples como base. Se um processo quer acesso a seção crítica e não está em posse do *token*, ele envia um REQUEST a todos os outros processos. Se um processo recebe esta mensagem e não está acessando a seção crítica, ele passa o *token*, do contrário, ele aguarda o final de sua execução da seção crítica para passar o *token*.

Nesta solução, cada processo p_i possui um número de sequência de requisições sn e mantém um vetor RN_i com o último número sn_i das requisições de outros processos recebidas previamente. Esta estrutura permite diferenciar se uma dada requisição é antiga ou recente. Define-se ainda que a estrutura do *token* é composta por uma fila Q dos processos requisitantes e um vetor $LN[1, ...1N]$ onde $LN[i]$ é o número de sequência da última requisição atendida pelo processo p_i, com isso é possível que o algoritmo determine se uma dada requisição ainda é válida.

Dado um proceso p_i que requer acesso a seção crítica, este processo incrementa seu número de sequência $R_i[i]$ e envia por *broadcast* uma mensagem REQUEST(i, sn) para todos os outros processos do sistema. Quando um processo p_j recebe esta mensagem, ele compara sn com

$R_j[i]$ e armazena em $R_j[i]$ o maior valor. Se p_j estiver com o token e verificar que $RN_j[i] = LN[i] + 1$, isto é, que p_i tem uma requisição pendente, ele envia o *token* para p_i. Ao receber o *token* o processo p_i podem por fim acessar a seção crítica. Ao sair da seção crítica p_i atualiza $LN[i] = RN_i[i]$ e coloca cada p_j, que não está na fila do *token*, na fila se houver uma requisição de p_j pendente. Se após estes passos a fila do *token* não estiver vazia, p_i deleta o identificador do topo da fila e envia o *token* para o processo indicado por este identificador.

3.3.2 Algoritmo de Raymond

O algoritmo baseado em passagem de *token* proposto por Raymond [Raymond 1989] utiliza-se de *spannig trees*, ou árvores geradoras, mínimas e sem raíz definida para reduzir o número total de mensagens enviadas por acesso a seção crítica. O algoritmo envia aproximadamente quatro mensagens com uma carga leve de trabalho.

Este algoritmo trabalha de forma similar aos algoritmos baseados em passagem de *token*, porém utiliza um conceito de privilégio na definição de quem terá acesso a seção crítica. O nó que possui o privilégio é chamado de nó-privilegiado e somente um nó pode estar em posse do privilégio em qualquer momento da execução. A única exceção a esta regra é quando uma mensagem PRIVILEGE é enviada para efetuar a transferência do privilégio para outro processo do sistema.

Cada processo mantém uma variável HOLDER, responsável por prover informações sobre a localização do privilégio em relação ao próprio processo. A variável HOLDER armazena quem o processo pensa que está ou quem pode levar ao nó privilegiado. Cada processo mantém também uma variável REQUEST_Q, que armazena as identidades de processos vizinhos que requisitarem o privilégio mas ainda não o receberam. Uma variável booleana ASKED indica se um dado processo já enviou ou não uma requisição. Por fim há uma variável booleana USING que indica se um processo está ou não a executar a seção crítica.

Um processo que possua um REQUEST_Q não vazio, irá sempre enviar mensagens REQUEST para tentar adquirir o privilégio e resolver as requisições pendentes. Após receber o privilégio por uma mensagem PRIVILEGE, irá checar quem está no topo de REQUEST_Q. Se for seu identificador, irá retirá-lo e entrar na seção crítica, se não for, irá encaminhar o privilégio para o processo referente ao identificador do topo, e remover este identificador de

REQUEST_Q.

O número de mensagens enviadas a cada solicitação de acesso a seção crítica é igual a duas vezes o caminho mais longo da árvore, e depende da topologia adotada.

3.4 Algoritmos Baseados em Sistema de Quóruns

Os algoritmos baseados em sistema de quóruns são uma variação dos algoritmos baseados em pedido de permissão. Nesse tipo de algoritmo, um processo não pede permissão à todos os outros processos do sistema, mas a processos que fazem parte de um subconjunto, do qual ele também faz parte, de processos do sistema. Todo processo do sistema pode dar permissão à somente um processo por vez, isto é, quando recebe um pedido de permissão ele entra em um estado que não permite que ele conceda permissão a mais nenhum processo até que o requisitante anterior o libere. Desta forma, um processo bloqueia todos os pedidos dos processos de seu subconjunto antes de acessar a seção crítica [Kshemkalyani e Singhal 2008].

A criação de subconjuntos dentro do sistema é baseada na noção de *coteries* e quóruns. Uma *coterie* é definida como um conjunto de conjuntos no qual cada conjunto $g \in C$ é chamado de quórum. As seguintes propriedades são definidas para quóruns em uma *coterie* [Kshemkalyani e Singhal 2008]:

- **Propriedade de interseção:** Para cada quórum $g, h \in C, g \cap h \neq \emptyset$, onde C é a *coterie*.
- **Propriedade de minimalidade:** Não devem existir quóruns g, h em C tal que $g \supseteq h$

A propriedade de interseção define que todo subconjunto terá um elemento em comum com os outros subconjuntos, então quando um processo requisita acesso a seção crítica e pede permissão a todos os membros de seu subconjunto, ele consegue a permissão de todos os outros subconjuntos, garantindo assim a exclusão mútua. Já a propriedade de minimalidade está ligada a eficiência uma vez que os quóruns podem ser construidos de maneiras diferentes e costumeiramente contém a maioria dos processos do sistema [Kshemkalyani e Singhal 2008]. Esta seção apresenta os principais algoritmos de exclusão mútua baseados em sistema de quóruns.

3.4.1 Algoritmo de Maekawa

Maekawa [Maekawa e Mamoru 1985] propôs o primeiro algoritmo de exclusão mútua baseado em sistema de quóruns. O algoritmo proposto usa da teoria de planos projetivos para geração de quóruns de tamanho $\sqrt{n}$, que estão sujeitos as seguintes condições:

I $\forall i \forall j : i \neq j, 1 \leqslant i, j \leqslant n : C_i \cap C_j \neq \emptyset$

II $\forall i : 1 \leqslant i \leqslant n : p_i \in C_i$

III $\forall i : 1 \leqslant i \leqslant n : |C_i| = k = \sqrt{n} + 1$

IV $\forall i, j : 1 \leqslant i, j \leqslant n$: todo p_i está presente em k conjuntos C_j

A propriedade I é necessária para para que os requisitos da exclusão mútua possam ser resolvidos e a propriedade II reduz o número de mensagens enviadas e recebidas por um processo. As propriedades III e IV são necessárias para garantir um algoritmo verdadeiramente distribuído e garantem, respectivamente, que um processo p_i envia e recebe a mesma quantidade de mensagens para conseguir acesso a seção crítica e que todo p_i tem o mesmo grau de responsabilidade no processo de exclusão mútua.

Assim como no algoritmo proposto por Lamport, os tipos de mensagem que podem ser enviadas entre os processos são: REQUIRE, REPLY e RELEASE. Quando um processo p_i deseja acessar a seção crítica ele envia uma mensagem REQUEST(i) a todos os processos p_j em seu quórum. Ao receber uma mensagem de REQUEST(i) um processo p_j, dado que não tenha enviado mais nenhum REPLY desde que recebeu seu último RELEASE, enviar um REPLY(j) para o processo p_i. Do contrário ele coloca a requisição de p_i na fila para ser avaliado mais tarde. O processo p_i só ganha acesso à seção crítica após ter recebido um REPLY(j) de todo processo p_j de seu quórum. Após sair da seção crítica o processo p_i envia uma mensagem RELEASE(i) para todos os processos p_j em seu quórum. Quando o proceso p_j recebe uma mensagem RELEASE(i), ele remove a requisiçãod e p_i da fila de requisições e envia uma mensagem REPLY para o próximo processo requisitante. Se a fila se encontrar vazia ele atualiza seu estado para comunicar que não enviou mais nenhum REPLY desde o recebimento do último RELEASE.

A solução de Maekawa tem o número de mensagens enviadas pautado no tamanho dos quóruns, dado que o uso de planos projetivos gera quóruns de tamanho $\sqrt{n}$ e três mensagens são enviadas por acesso a seção crítica, o número total de mensagens enviadas é de $3\sqrt{n}$ mensagens. Porém este número tende a ser maior uma vez que, segundo Singhal [Singhal 1992], a solução de Maekawa tem tendência de gerar *deadlocks*, e solucionar esse problema acarreta em um aumento no número de mensagens enviadas [Kshemkalyani e Singhal 2008].

3.4.2 Algoritmo de Agarwal-El Abbadi

Proposto por Agarwal-El Abbadi [Divyakant e Amr 1991] este algoritmo simples, eficiente e tolerante à falhas introduz o conceito de árvore de quóruns. A idéia é gerar quóruns estruturados em árvore de modo a usar uma estrutura hierárquica de uma rede. Todos os processos do sistema são organizados em uma árvore binária completa.

O processo de geração dos quóruns estruturados em árvore faz o uso de duas diferentes funções: a função recursiva *GetQuorum* que recebe a raíz da árvore como parâmetro e a função *GrantsPermission(processo)*. A função *GetQuorum* é chamada no nó filho se a função *GrantPermission(processo)* for verdadeira, e esta por sua vez é verdadeira somente se o processo aceitar fazer parte do quórum, esta aceitação será negativa caso o processo tenha falhado. O algoritmo tenta gerar os quóruns de forma que cada um represente um caminho possível da raíz até as folhas da árvore. Caso a folha da árvore não esteja acessível, o algoritmo termina com um erro, já que cada caminha deve, obrigatoriamente, terminar em uma folha da árvore. O tamanho dos quóruns gerados por este algoritmo é de *O(log n)* e em cenários de falhas, como no caso de nó pai de uma folha falhar, ainda será de *O(log n)*. O algoritmo consegue tolerar falhas de até *n - O(log n)* processos e no pior caso gera quóruns de tamanho *O((n+1)/2)*.

Por exemplo, a árvore da Figura 3.1 pode gerar os seguintes quóruns: $\{1, 2, 4\}$, $\{1, 2, 5\}$, $\{1, 3, 6\}$ e $\{1, 3, 7\}$. Caso ocorra uma falha no nó 2 e 3, por exemplo, pegando as duas subárvores os quóruns gerados passam a ser: $\{1, 4, 5\}$ e $\{1, 6, 7\}$. Dada a forma que seus quóruns são gerados, existem cenários onde o algoritmo não tolera falhas de até *n - O(log n)* pois não consegue gerar um quórum válido. No exemplo da Figura 3.1, o algoritmo deve tolerar até pelo menos 4 falhas. Caso os procesos $2, 5, 3$ e 7 falhem, observamos que não é possível gerar nenhum quórum, assim, o algoritmo apresenta uma certa desvantagem em cenários onde muitas

falhas possam ocorrer.

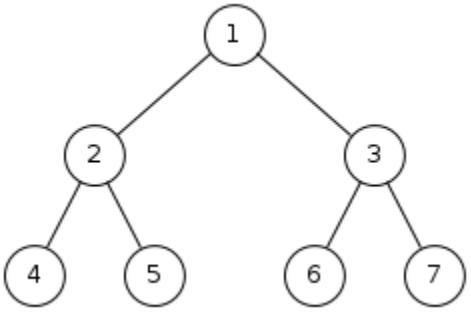


Figura 3.1: Árvore

Quando um processo p_i deseja acessar a seção crítica ele envia uma mensagem de *Request* para todos os outros processos p_j que pertencerem ao seu quórum. Cada processo em um quórum mantém uma fila de requisições ordenada pelo *timestamp* de cada requisição. Um processo p_j só responderá com um Reply se o processo p_i se encontrar no topo da fila de requisições, e caso um Request de menor prioriridade chegar à p_j, uma mensagem *Inquire* é enviada ao p_i do topo da lista. Se p_i recebe uma mensagem Inquire e não obteve o número de Reply's necessários para acessar a seção crítica, ele responde p_j com uma mensagem Yield. Se o processo p_i obter permissão de todos os processos em seu quórum ele acessa a seção crítica. Ao sair da seção crítica o processo p_i envia uma mensagem Relinquish pra todos os processos p_j em seu quórum, que por sua vez removem o Request de p_i do topo de suas filas.

O total de mensagens enviadas por solicitação de seção crítica é de 3 mensagens por processo em um dado quórum, totalizando $O(logn)$ no melhor caso e $O((n+1)/2)$ no pior caso.

Capítulo 4

Um Algoritmo de Exclusão Mútua no VCube

Os algoritmos baseados em quóruns apresentados no Capítulo 3, com exceção do de Maekawa, apresentam tolerança a falhas na geração dos quóruns, porém a ocorrência de falha não é abordada no algoritmo de exclusão mútua. Para que um algoritmo de exclusão mútua possa ser tolerante a falhas é preciso um sistema que monitore o estado de cada processo do sistema e informe o algoritmo para que ele envie as mensagens corretas e garanta a exclusão mútua, para tanto será utilizado o VCube. O VCube é um sistema distribuido de diagnóstico que usa de uma topologia virtual de hypercubo. A estrutura do VCube oferece um ambiente de monitoramento que permite que os processos sempre sejam informados sobre as ocorrências de falhas no sistema, o que permitiu também que fosse desenvolvido um algoritmo gerador de qúoruns tolerante a falhas [Rodrigues 2014]. Esta seção apresenta a topologia virtual VCube, seguida do algoritmo gerador de quóruns no VCube e da proposta de algoritmo de exclusão mútua com tolerância a falha.

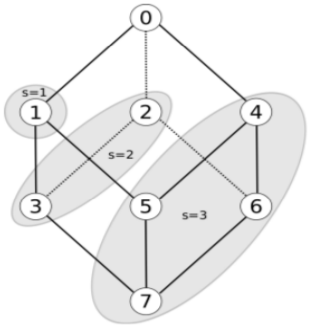
4.1 A Topologia Virtual VCube

Na topologia de virtual VCube os processos do sistema são organizados em clusters de diferentes tamanhos. Cada cluster $s = 1, ..., \log_2 n$ é composto por 2^{s-1} elementos onde n é o número total de processos executando no sistema. Na topologia de hipercubo virtual, cada processo do sistema executando VCube é capaz de testar outros processos no sistema, verificando se estes por sua vez estão corretos ou falhos. Todo processo que responda ao teste dentro do tempo esperado é considerado correto. Os testes são executados em rodadas. Durante uma ro-

dada cada processo i testa o primeiro processo j sem falhas na lista de processos de cada cluster s do sistema, obtendo assim informações sobre todos os processos do sistema. A definição de quais processos i pertencem a um cluster s é dado por:

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

Todo processo (falho ou correto) é testado uma única vez por rodada o que garante uma latência de diagnóstico média de $\log_2 n$ rodadas. A Figura 4.1, abaixo, exemplifica a organização hieráquica do VCube.



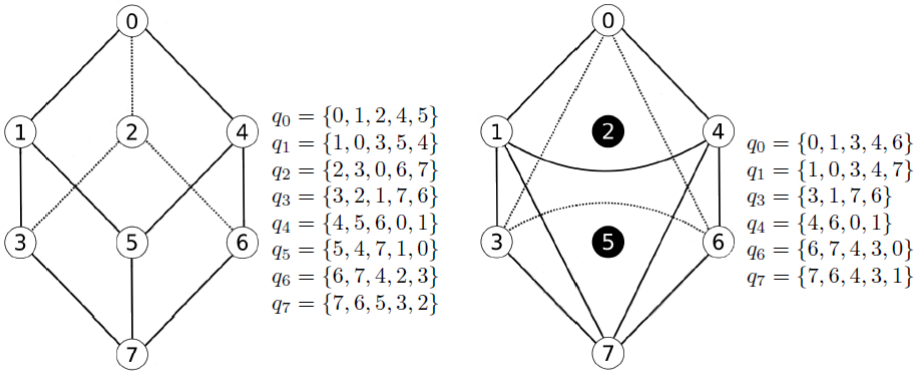
s	$c_{0,s}$	$c_{1,s}$	$c_{2,s}$	$c_{3,s}$	$c_{4,s}$	$c_{5,s}$	$c_{6,s}$	$c_{7,s}$
1	1	0	3	2	5	4	7	6
2	2 3	3 2	0 1	1 0	6 7	7 6	4 5	5 4
3	4 5 6 7	5 4 7 6	6 7 4 5	7 6 5 4	0 1 2 3	1 0 3 2	2 3 0 1	3 2 1 0

Figura 4.1: Exemplo da Topologia Virtual VCube

Para propagar as mensagens de requisição, um algoritmo de exclusão mútua pode fazer uso de um mecanismo de *broadcast*. No caso de sistemas baseados em quóruns, um algoritmo de multicast pode ser utilizado. Neste sentido, em Rodrigues[Rodrigues, Jr. e Arantes 2015], um algoritmo de *multicast* para os quóruns do VCube foi proposto. O algoritmo constrói árvores geradora sob demanda. Essa construção é feita com base na topologia VCube à partir da raiz principal, e sua autonomia vem de sua capacidade de organizar automaticamente os processos do sistema e reconstruir a árvore sempre que uma falha for detectada, mantendo assim uma árvore mínima. Com base na função $c_{s,i}$ define-se: Seja i um processo executando o algoritmo de árvore geradora e seja a dimensão $d = \log_2 n$ do VCube com 2^n processos. A lista de processos testados como corretos por i é armazenada no conjunto de processos não falhos $correct_i$. Considerando uma execução inicialmente correta, a propagação das mensagens de teste é iniciada. Uma mensagem é enviada aos $\log_2 n$ vizinhos (em outro cluster) sem falha do processo i. Cada processo ao receber uma mensagem do processo j, verifica se o processo $j \in correct_i$ e retransmite a mensagem aos vizinhos internos ao cluster em que se encontram. Caso um processo

falhe, este é excluido do conjunto $correct_i$, e o processso i envia a mensagem para o próximo vizinho sem falha k (pertencente ao mesmo cluster do processo falho j) que por usa vez propaga a mensagem internamente no seu cluster.

No exemplo da Figura 4.2 temos os quóruns de cada processo em dois momentos: (a) antes das falhas dos processos p_2 e p_5 e (b), após a falha dos processos p_2 e p_5. Como exemplo, vejamos como é calculado o novo quórum do proceso p_0 após a ocorrência das duas falhas. O quórum de p_0, tanto antes como depois da falha, é composto por ele mesmo mais a metade absoluta dos clusters $c_{0,1}$ = (1), $c_{0,2}$ = (2,3), $c_{0,3}$ = (4,5,6,7). Após calcular os clusters, calcula-se $FF\_cluster_0$ para se obter a lista dos elementos considerados corretos em cada cluster s. Logo temos, $FF\_cluster_0(1)$ = (1), $FF\_cluster_0(2)$ = (3) e $FF\_cluster_0(3)$ = (4,6,7). Por fim, calcula-se FF_mid_0 para se obter a metade absoluta dos processos considerados corretos em cada cluster. Assim obtemos: $FF\_mid_0(1)$ = (1), $FF\_mid_0(2)$ = (3) e $FF\_mid_0(3)$ = (4,6). Como podemos ver na Figura 4.2, o novo quórum do processo p_0 deixa de conter os elementos falhos e os substituiu pelos processos p_3 e p_6.



(a) Sem falhas (b) Processos p_2 e p_5 falhos

Figura 4.2: Exemplo do Gerador de Quórum com falhas em p_2 e p_5.

4.2 Algoritmo de Geração de Quóruns no VCube

O algoritmo de geração de quóruns majoritários proposto por Rodrigues [Rodrigues 2014] utiliza-se de algumas funções e estruturas auxiliares para gerar quóruns tolerantes a falhas sobre a topologia do VCube.

Um conjunto $correct_i$ é definido para armazenamento de informações sobre um processo i tem sobre o estado dos outros processos do sistema, essas informações são obtidas pelo monitoramento feito pelo algoritmo de diagnósticos do VCube. Problemas de latência podem acarretar em informações desatualizadas neste conjunto mas são corrigidas tão logo dados mais recentes cheguem ao processo i. Seja a função $c_{i,s}$ definida no VCube, a lista $(a_1, a_2, ..., a_m), m = 2^{s-1}$ resultante desta função define todos os processos no cluster s do processo i. Define-se uma função $\textit{FF\_cluster}_i(s) = (b_1, b_2, ..., b_{m'}), b_k \in (c_{i,s} \cap correct_i), k = 1..m', m' \leqq m$ onde m' é uma lista de elementos considerados corretos por i em um dado cluster $c_{j,s}$. Rodrigues define ainda uma função $\textit{FF\_mid}_i(s) = (b_1, b_2, ..., b_{[m'/2]})$ responsável por gerar um conjunto com a maioria absoluta dos processos considerados corretos por um processo i.

Cada processo i no sistema irá gerar seu próprio quórum adicionando a si mesmo e a metade dos processos que considerar correto (sem-falha) em cada um dos clusters $c_{i,s}$ do VCube. Os quóruns permanecem inalterados por todo o tempo de execução até que o monitoramento do VCube informe a ocorrência de um processo falho. Ao receber a informação que um dado proceso j está falhou, o processo i irá remover o processo j do conjunto $correct_i$ e recriará todos os quóruns com os processos não falhos restantes.

4.3 Um Algoritmo de Exclusão Mútua no VCube

Para que o algoritmo de exclusão mútua seja compatível com o algoritmo de geração de quóruns de Rodrigues[Rodrigues 2014], é necessário que o mesmo também seja tolerante a falhas. Para tanto, algumas estruturas auxiliares se fazem necessárias. Cada processo p_i terá uma fila de requisições, esta fila será ordenada por ordem de prioridade (menor timestamp tem maior prioridade e em casos de empate, o processo com menor id tem maior prioridade). O algoritmo utiliza um Relógio de Lamport [L. 1978] para contar o tempo do sistema. O processo p_i mantém também um conjunto $locked_i$, que armazena de quais processos p_j o processo p_i espera uma mensagem de REPLY. Por fim cada processo p_i mantém um conjunto $members_i$ que armazena quais processos fazem parte do mesmo quórum que p_i.

As interações entre os processos são feitas através da troca de mensagens. As seguintes mensagens foram definidas para este algoritmo:

- **REQUEST**: Esta mensagem é enviada sempre que um processo deseja ter acesso à seção

crítica.

- **REPLY**: Quando um processo dá permissão de acesso à seção crítica para um processo requisitante, uma mensagem REPLY é enviada.
- **FAILED**: Qunado um processo não pode dar permissão de acesso à seção crítica para um processo requisitante, uma mensagem FAILED é enviada.
- **INQUIRE**: Caso um processo bloqueaod receba uma mensagem me maior prioridade, ele envia INQUIRE para quem o bloqueou.
- **YIELD**: Ao receber INQUIRE, se um processo verificar que não irá obter todas as permissões, ele responde com uma mensagem YIELD.
- **RELEASE**: Um processo que obteve acesso à seção crítica envia mensagens RELEASE após sair da mesma.
- **CANCEL**: Quando ocorre mudanças nos quóruns (casos de falha), um processo deve cancelar as requisições feitas à processos que não estão mais em seu quórum, assim ele envia uma mensagem do tipo CANCEL.

Quando um processo p_i deseja acessar a seção crítica, ele envia uma mensagem de REQUEST para todos os processos $p_j \in members_i$. Ao receber uma requisição, um processo p_j irá colocá-la em sua fila de requisições, e enviará uma mensagem de REPLY, dando permissão de acesso a seção crítica, para o processo p_i que estiver no topo da fila, não enviando mais nenhuma mensagem de REPLY até que receba uma mensagem RELEASE ou YIELD do processo p_i. Um processo p_j que estiver bloqueado e receber uma requisição, irá verificar se a prioridade da mesma é maior do que a prioridade do processo que o bloqueara anteriormente (requisições com *timestamp* menor tem maior prioridade e em casos de empate o requisitante com menor identificador tem maior prioridade). Se a prioridade for maior, ele colocará a requisição na fila de requisições e enviará uma mensagem INQUIRE para o processo bloqueante. O processo p_i ganha acesso a seção crítica se receber um REPLY para cada REQUEST enviado. Ao sair da seção crítica p_i irá enviar uma mensagem RELEASE para cada $p_j \in locked_i$, e irá processar a requisição no topo de sua fila.

Quadro 4.1 Algoritmo de Exclusão Mútua no VCube

Requisição:

1 - p_i envia REQUEST via multicast para todo o seu quórum

2 - p_i aguarda REPLY de todos os procesos em seu quórum

3 - p_j recebe requisição e a coloca na fila de requisições

4 - Se p_j está bloqueado, ele verifica se a requisição de p_i tem maior prioridade do que a requisição p_k que o bloqueara. Se tem, p_j envia INQUIRE para p_k e coloca p_i na fila. Se não tem, p_j envia FAILED para p_i.

5 - Se p_j não estiver bloqueado, envia REPLY para o primeiro p_i da fila

7 - Ao receber uma mensagem INQUIRE de p_j, o processo p_k verifica se recebeu anteriormente uma mensagem FAILED. Caso tenha recebido, p_k envia uma mensagem YIELD para p_j.

8 - Ao receber uma mensagem YIELD de p_k, p_j envia uma mensagem de REPLY para o primeiro p_i de sua fila de requisições.

Execução:

1 - p_i recebe REPLY de todo seu quórum

2 - p_i muda seu estado para bloqueado e acessa a seção crítica

Release:

1 - p_i sai da seção crítica e envia RELEASE via multicast para todos os processos que bloqueou

2 - p_j recebe mensagem RELEASE e volta a responder requisições pendentes na fila.

Em casos de falha:

Falha em processo bloqueado:

1 - p_i é informado que um p_j falhou

2 - p_i verifica se houve mudança em seu quórum

3 - p_i envia uma mensagem REQUEST para todo processo novo em seu quórum

Falha em processo que obteve seção crítica:

1 - p_j é informado que o processo p_i que o bloqueara falhou.

2 - p_j muda seu estado.

3 - p_j volta a responder requisições pendentes na fila.

Mudança no quórum com a saída de um proceso não falho:

1 - p_i é informado que houve mudança em seu quórum e verifica que um processo p_j não falho não pertence mais ao mesmo.

2 - Caso p_i tenha enviado um REQUEST para p_j anteriormente, p_i envia uma mensagem CANCEL para p_j.

3 - p_j recebe uma mensagem CANCEL de p_i e remove a requisição de p_i que estiver em sua fila de requisições.

Ao receber uma mensagem de INQUIRE, um processo p_k irá verificar se já recebeu uma mensagem do tipo FAILED. Caso tenha recebido, irá enviar uma mensagem YIELD para o processo p_j que enviou o INQUIRE e irá remover o REPLY que recebera de p_j de sua fila de respostas. Caso ainda não tenha recebido nenhuma mensagem FAILED, p_k irá postergar a resposta até que receba ou até que obtenha uma seção crítica (caso o qual p_k desbloqueará p_j ao enviar uma mensagem RELEASE).

Caso um processo p_i esteja aguardando mensagens de REPLY do processo p_j e o monitoramento do VCube informe o processo p_i que o processo p_j falhou, p_i irá, após a atualização do quórum, enviar uma nova mensagem de REQUEST para todo processo $p_j \in (members_i - locked_i)$, isto é, para todo processo que antes da falha não fazia parte do seu quórum, e portanto não havia recebido um REQUEST de p_i. Além disso, todo processo p_i que for informado que um processo p_j falhou, irá busca-lo em suas filas de requisições e respostas e remover as mensages de p_j. Podem ocorrer casos onde, no cálculo do novo quórum, um ou mais processos não falhos deixem de pertencer ao quórum de p_i. Caso isso ocorra e p_i tenha REQUESTs pendentes, p_i envia uma mensagem CANCEL para cada um desses processos.

Caso um proceso p_j esteja aguardando uma mensagem de RELEASE e o monitoramento do VCube informe um dado processo p_i que o bloqueara anteriormente, falhou, p_j irá mudar seu estado, removera de sua fila de requisições todos as requisições de processos que não se encontram mais em $member_j$ e passará a processar as requisições pendentes. A permissão de acesso dado anteriormente a p_i, agora falho, é revogada, e novos processos podem obter acesso a seção crítica.

4.3.1 Exemplo de Execução

Em um dado sistema tem-se o seguinte cenário: o processo p_0 possui quórum $\{p_0, p_1, p_2, p_3\}$, o processo p_3 possui quórum $\{p_0, p_1, p_2, p_3, p_6\}$ e o processo p_4 possui quórum $\{p_2, p_4, p_{10}, p_{11}\}$. Dada a propriedade de interseção, os quóruns do processo p_0 e p_3 se diferem por apenas um processo e p_2 faz parte também do quórum de p_4. Os processos p_0, p_3 e p_4 desejam obter acesso à seção crítica, então enviam mensagens de REQUEST para todos os membros de seu quórum e simulam um envio de REQUEST para si mesmos, colocando-se em sua própria fila e mudando seu estado para bloqueado. Os processos p_0 e p_4 enviam suas mensagens no mesmo instante e

o processo p_3 envia as suas alguns segundos depois (Figura 4.3).

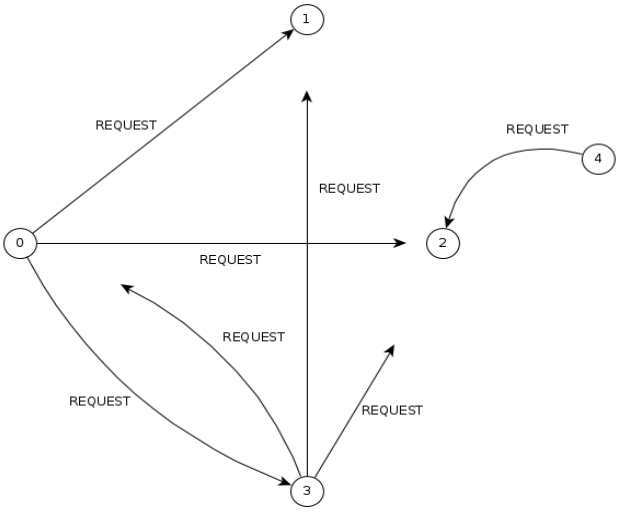


Figura 4.3: Requisições de p_0, p_3 e p_4.

O processo p_2 recebe primeiro a requisição de p_4, coloca-a em sua fila de requisições e envia uma mensagem de REPLY. Ao receber a requisição de p_0, o processo p_1 ainda se encontra desbloqueado, e assim, a coloca em sua fila de requisições e responde com uma mensagem de REPLY. O processo p_3 recebe a requisição de p_1 antes de enviar sua requisições, assim ele se bloqueia para p_1 e envia um REPLY. Os processos que já estavam bloqueados ao receber as mensagens REQUEST de p_3, respondem com uma mensagem FAILED (Figura 4.4)

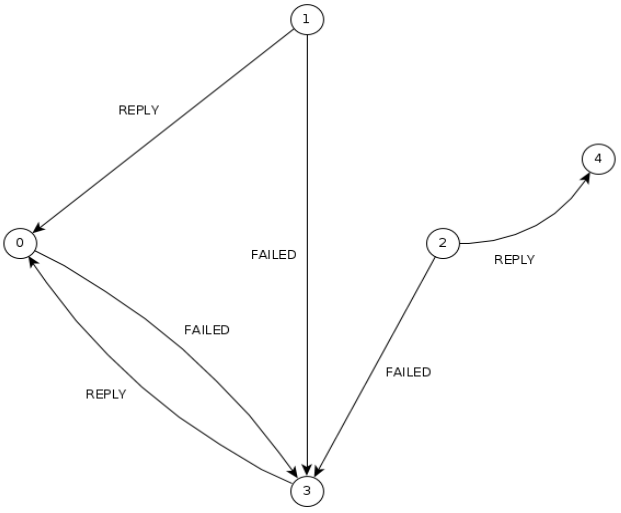


Figura 4.4: REPLYs e FAILEDs de p_0, p_1, p_3 e p_4.

Ao receber a requisição de p_0, o processo p_2, se encontrava bloqueado pela requisição de p_4.

Como a requisição de p_0 tem maior prioridade que a de p_4, o processo p_2 envia uma mensagem de INQUIRE ao processo bloqueante p_4. (Figura 4.5).

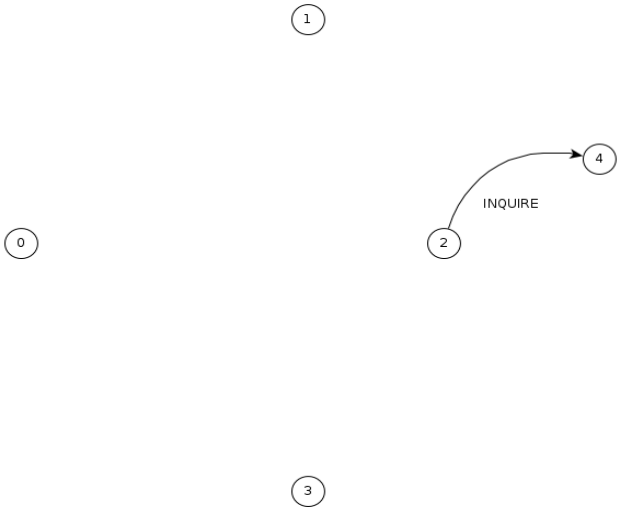


Figura 4.5: Inquire para p_4.

Ao receber o INQUIRE enviado pelo processo p_2, o processo p_4 verifica que não conseguirá obter acesso à seção crítica (i.e., recebeu uma mensagem FAILED de outro processo em seu quórum) e responde com uma mensagem de YIELD. O processo p_2 recebe a mensagem YIELD e envia uma mensagem de REPLY para p_0 (Figura 4.6). O processo p_0 obteve todos os REPLYs necessários e obtém assim acesso à seção crítica.

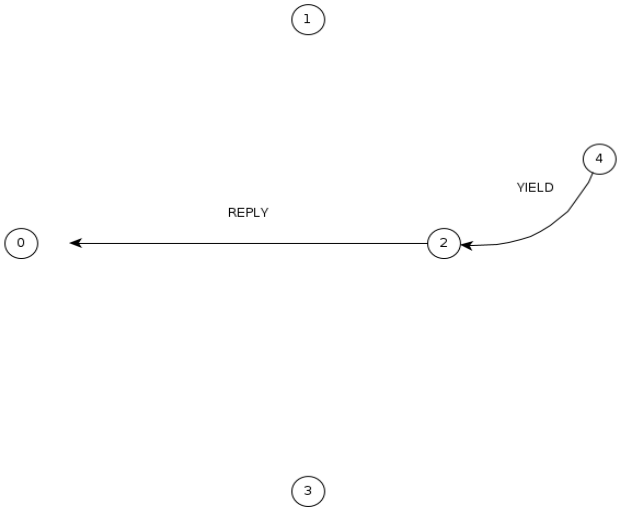
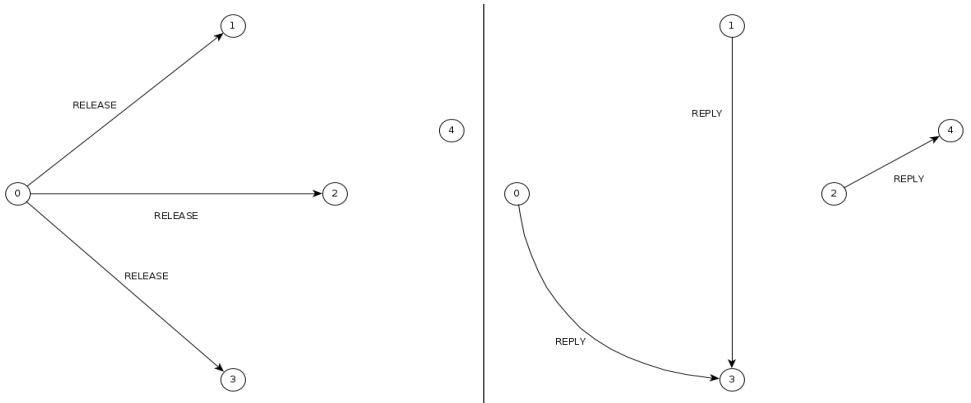


Figura 4.6: Último REPLY necessário para p_2.

Ao sair da seção crítica p_0 envia uma mensagem RELEASE para todo processo que havia

bloqueado (Figura 4.7-(a)). Em seguida envia uma mensagem de REPLY para o novo processo de maior prioridade em sua fila de requisições, que no caso é o proceso p_3. Os processos p_1 e p_2 recebem o RELEASE e enviam um REPLY para o processo de maior prioridade em suas filas de requisições (Figura 4.7-(b)).



(a) O processo p_0 saida seção crítica.

(b) Os processos voltam a processar suas filas.

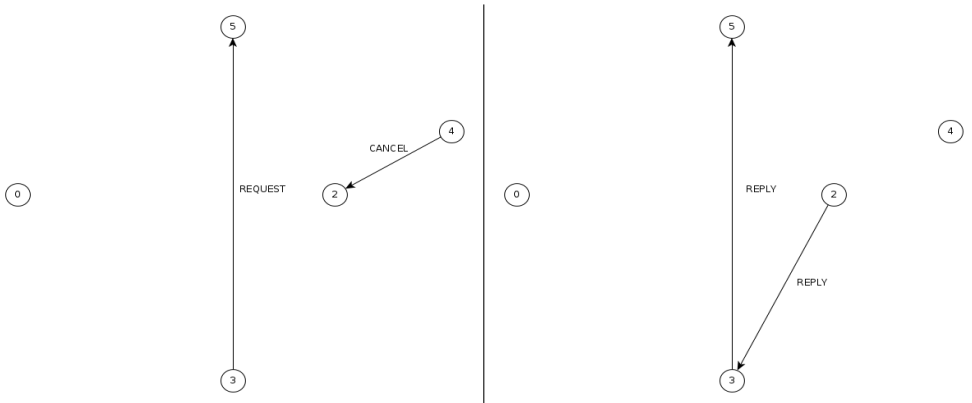
Figura 4.7: Saida da seção crítica

Suponha que uma falha ocorreu no processo p_1, isso causou uma alteração nos membros do quórum do processo p_0 que passou a ser $\{p_0, p_2, p_3, p_5\}$ e no quórum de p_3 que passou a ser $\{p_0, p_5, p_2, p_3, p_6\}$e fez com que o processo p_2, apesar de não estar falho, não fizesse mais parte do quórum de p_4. Como há um novo processo em seu qúorum o processo p_3, que teve de retirar o REPLY de p_1 de sua fila de respostas, envia uma nova mensagem REQUEST. Para cada processo não falho que saiu de seu quórum e para o qual havia sido enviada uma mensagem de REQUEST, p_4, envia uma mensagem CANCEL (Figura 4.8-(a)).

Ao receber uma mensagem CANCEL o processo p_2 remove a requisição de p_4 da fila e envia REPLY para o processo de maior prioridade na mesma (Figura 4.8-(b)).

4.3.2 Propriedades do Algoritmo

Para que se possa garantir o funcionamento do algoritmo proposto, é preciso garantir algumas das propriedades de algoritmos de exclusão mútua citadas na Seção 3. Primeiramente deve-se garantir que um e apenas um processo por vez obterá acesso à seção crítica (*safety*). Deve-se garantir também que um processo que aguarda acesso à seção crítica irá obter este acesso em um tempo finito (*liveness*) e, por fim, deve-se garantir a ausência de impasse (*dea-*



(a) Nova mensagens são enviadas

(b) O processo 3 obtém os REPLYs que faltavam.

Figura 4.8: Falha com alteração nos quóruns

dlocks).

Segurança (*safety*)

Considere que cada processo envie mensagens de REQUEST para os membros de seu quórum e os quóruns possuem interseções entre si. Considere que cada processo pode estar em dois estados diferentes: bloqueado e não bloqueado. Um processo não bloqueado pode responder à uma requisição e caso o faça, seu estado muda para bloqueado. Um processo bloqueado não responde requisições de menor prioridade e só responde à requisições de maior prioridade se o processo que o bloqueara liberá-lo, abdicando assim da permissão que havia obtido. Assim, um processo só dá permissão à um requisitante por vez. Se cada processo responde apenas um requisitante por vez, garante-se que um processo pertencendo a dois quóruns diferentes irá responder à apenas um processo e que dentro do qúorum, apenas um processo terá todas as respostas necessárias para acessar a seção crítica. Se um processo falhar, os processos bloqueados serão informados da falha e irão revogar a permissão dada ao processo falho, respondendo então o processo de maior prioridade em sua fila.

Progressão (*liveness*)

Se um ou mais processos disputam o acesso à seção crítica, cada um terá sua requisição atendida conforme a prioridade da mesma nas filas de requisição. Como a prioridade é definida pelo *timestamp* e, em caso de empates, pelo identificador, a propriedade de progressão é

garantida pela ordenação por da fila de requisições de cada processo.

Impasse (*deadlock*)

Considere que em um dado sistema tenhamos três processos: p_0, p_1 e $p2$ e que haja uma situação de espera circular onde p_0 está esperando por um REPLY de p_1, este está esperando por um REPLY de p_2 e este está esperando um REPLY de p_1. Situações assim são evitadas pelo algoritmo, dado que:

1. As filas de requisições são ordenadas por ordem de prioridade, esta prioridade se dá primeiro pelo *timestamp* de cada requisição e em casos de empate pelo menor identificador. Como cada processo possui identificador único, a fila estará sempre ordenada.

2. Assim sendo, em qualquer cenário deverá haver pelo menos um processo cujas requisições tenham maior prioridade do que todas as outras.

3. Independentemente da prioridade das requisições enviadas, não há nada que garante que elas cheguem na mesma ordem em que foram enviadas, logo, uma requisição de menor prioridade que chega em um processo já bloqueado é respondida com um FAILED. Se o processo não se encontra bloqueado, responde com um REPLY.

4. Um processo que recebeu um FAILED irá responder mensagens de INQUIRE com um YIELD, já que, isso significa que não obterá seção crítica.

5. Ao receber uma requisição de menor prioridade um processo irá enviar uma mensagem INQUIRE ao processo bloqueante. O processo bloqueante irá responder com YIELD, quebrando a espera circular, caso não consiga seção crítica ou irá postergar a resposta até que receba FAILED ou que consiga seção crítica. Se conseguiu seção crítica, não há espera circular, do contrário, irá responder ao INQUIRE, o que permite que seja enviado um REPLY para o REQUEST de maior prioridade, quebrando a espera circular.

Já que não há como dois processos diferentes possuirem, ao mesmo tempo, requisições com maior prioridade que todos os outros e como há maneiras de um processo ceder a permissão obtida para um com maior prioridade, não há cenários em que possa ocorrer um impasse.

4.3.3 Desempenho

O número de mensagens que um dado processo envia para obter seção crítica está diretamente relacionado ao tamanho n de seu quórum.

Em um cenário sem falhas onde todos os processos disputam por um recurso. um processo p_i que deseja acessar a seção crítica envia, em média n mensagens REQUEST, n mensagens RELEASE e n mensagens REPLY. O número de mensagens FAILED, YIELD e INQUIRE dependem do número de processos requisitantes e variam conforme a ordem que as requisições chegam em cada processo p_j. Assim, em um cenário sem falhas, um processo envia pelo menos $3n$ mensagens por seção crítica obtida.

Em um cenário com falhas o número de mensagens continua diretamente relacionado ao tamanho do quórum, porém, passa a depender também de variáveis como o número de processos novos em um quórum após a ocorrência de uma falha e o número de processos não falhos que deixam de pertencer ao qúorum após uma falha. Esses fatores aumentam o número de mensagens enviadas de cada tipo de mensagem (REQUEST, REPLY, RELEASE, YIELD, FAILED, REPLY, INQURIE) e estão diretamente ligados à forma com que cada gerador de quóruns lida com a ocorrência de falhas.

Capítulo 5

Avaliação Experimental e Resultados

Para avaliar o comportamento do algoritmo proposto, foram elencados mais dois geradores de quóruns (além do gerador para o qual este algoritmo foi proposto) com o intuito de comparar os resultados obtidos e analisar como as diferentes características de cada gerador influencia no desempenho do algoritmo de exclusão mútua proposto. Este capítulo apresenta uma introdução a esturural geral do simulador, as adaptações necessárias, quais parâmetros foram avaliados e quais fram os resultados obtidos.

5.1 Estrutura Geral

Para implementar o algoritmo e avaliar seu comportamento com diferentes geradores de quóruns utilizou-se o *framework* Neko [Urban, Defago e Schiper 2001] e a linguagem de programação Java. Cada processo do sistema executa uma instância do algoritmo de exclusão mútua, do algoritmo de monitoramento do VCube e do algoritmo de *broadcast* para envio de mensagens. O algoritmo de exclusão mútua envia e recebe suas mensagens através do algoritmo de *broadcast*, que por sua vez usa a rede do Neko para simular os envios e recebimentos. O algoritmo do VCube usa da rede do Neko para se comunicar e monitorar os processos e, caso uma falha ocorra, informa o algoritmo de exclusão mútua da falha (Figura 5.1).

O algoritmo de exclusão mútua foi testado com três geradores de qúorum: Quóruns em Grade [Maekawa e Mamoru 1985], Quóruns em Árvore [Divyakant e Amr 1991] e Quóruns baseado no VCube [Rodrigues 2014]. Os quóruns em árvore e baseados no VCube possuem um certo grau de tolerância à falhas. No algoritmo de Maekawa a tolerância a falha foi feita usando-se da uma técnica de substituição semelhante à sugerida para casos em que o número

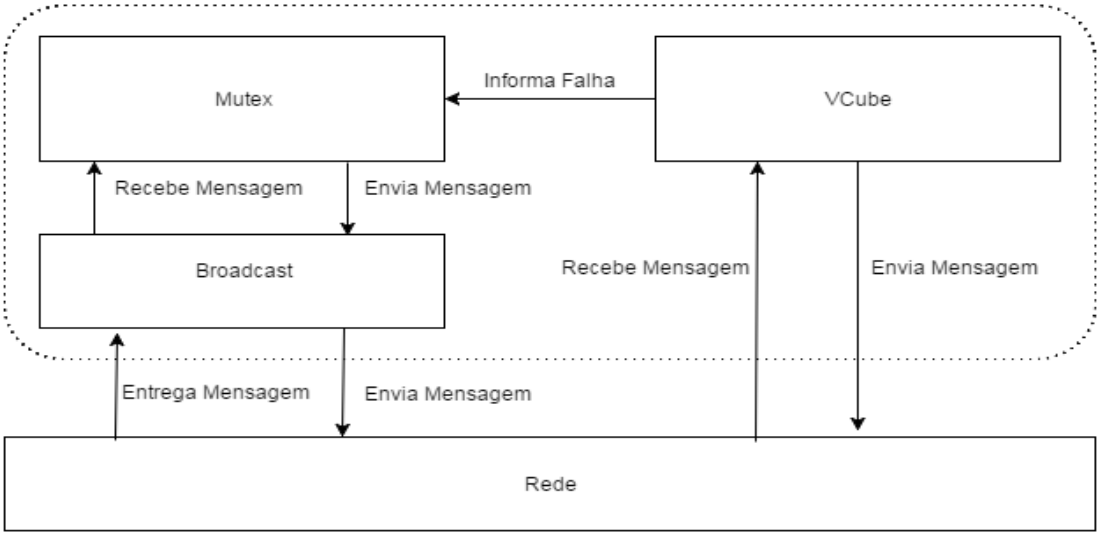


Figura 5.1: Módulos do Experimento

total de processo não é o quadrado de um inteiro (situação em qua sobram espaços vazios no grid, preenchidos por algum valor de outra linha ou coluna, durante a geração dos quóruns) como podemos ver na Figura 5.2.

Figura 5.2: Grid de Maekawa

5.1.1 Parâmetros

Dois diferentes cenários foram avaliados: com um processo requisitante e com N processos requisitantes. Em cada cenário foram avaliados casos com uma, duas, e $n/2$ falhas. A avaliação foi feita contando o número médio de mensagens enviadas por um processo até que o mesmo consiga acessar a seçao crítica. Os casos de teste foram gerados de forma que número s de processos que falham fossem escolhidos aleatoriamente e o tempo t de simulação em que a falha ocorre também fosse aleatório. O mesmo caso de teste foi utilizado com cada gerador de quóruns.

5.1.2 Cenários Sem Falhas

Em um cenário sem falhas e com apenas um processo requisitando acesso à seção crítica, observamos que o número de mensagens enviadas, nos três geradores de quórum, é tal qual descrito na sub-seção 4.3.3, isto é, diretamente relacionado ao tamanho dos quóruns gerados. Como não há processos concorrendo por um mesmo recurso não há envio de mensagens INQUIRE, YIELD ou CANCEL e cada processo envia n REQUESTs, recebe n REPLYs e ao sair da seção crítica envia n RELEASES (Tabela 5.1). O desvio padrão σ se deve ao fato de que a maioria dos processos não enviam mensagem alguma (com exceção dos membros do quórum do processo requisitante, que enviam uma mensagem REPLY).

Tabela 5.1: Um requisitante

	Maekawa		Agrawal		Rodrigues	
Processos	$\bar{x}$	σ	$\bar{x}$	σ	$\bar{x}$	σ
8	15	38.40	12	30.79	15	38.40
16	21	80.00	15	57.22	27	102.83
32	33	182.23	18	99.46	51	281.55
64	45	355.70	21	166.05	99	782.42
128	66	742.22	24	269.94	195	2192.80
256	62	988.65	27	430.73	387	6173.21
512	90	2033.01	30	677.82	514	11610.63

Em um cenário com n processos tentando obter seção crítica (Tabela 5.2) o número médio de mensagens enviadas aumenta, uma vez que agora todos os processos concorrem com todos os processos por um único recurso. Os processos passam a enviar mensagens de FAILED, INQUIRE e YIELD, o que causa um aumento no envio de todos os tipos de mensagem. Nota-se que o desvio padrão diminui drásticamente já que agora a maioria dos processos envia em média uma quantidade semelhante de mensagens. Para cada número de processos no sistema, pode ocorrer um desvio menor como acontece entre 128 e 256 processos no algoritmo de Maekawa. Essa diferença se deve a forma com que o algoritmo de Maekawa gera os qúoruns, e, é possível observar que para número de processos que são quadrados de inteiros (onde não há substituição no final do grid e todos os qúoruns possuem o mesmo tamanho) o desvio padrão é menor. Para o quórum em árvore de Agrawal-El-Abaddi o desvio padrão se mantém alto, isso está relacionado à forma com que os quóruns dos processos são construídos. Cada quórum é composto por um caminho percorrido da raíz até uma folha, adicionado do próprio processo. Assim, dada

a ausência de falhas, todos os processos possuem quóruns quase que idênticos, já que o primeiro caminho encontrado até uma folha forma um quórum, diferenciando-se em apenas um processo, o que faz com que alguns processos enviem um número de mensagens muito superior ao da maioria dos processos do sistema. Para situações sem a ocorrência de falhas, o algoritmo de exclusão mútua, utilizando do gerador de qúoruns em árvore de Agrawal-El-Abbadi envia em média um número muito menor de mensagens por seção crítica obtida, porém a carga de mensagens (envio e recebimento) não é muito bem distribuída, já que alguns processos enviam um número muito maior de mensagens (evidenciado pelo desvio padrão). Usando dos geradores de qúorum de Maekawa e Rodrigues o número de mensagens aumenta consideravelmente, principalmente no caso de Rodrigues que gera quóruns de maior tamanho. Entretanto, olhando-se para o desvio padrao de ambos nota-se uma menor variação em relação a média de mensagens enviadas por cada processo, o que indica uma melhor distribuição de carga entre os processos.

Tabela 5.2: N requisitantes

	Maekawa		Agrawal		Rodrigues	
Processos	$\bar{x}$	σ	$\bar{x}$	σ	$\bar{x}$	σ
8	18	2.66	16	6.00	19	1.17
16	26	1.60	20	12.04	33	2.27
32	40	5.70	24	21.19	61	4.59
64	55	3.39	28	34.86	119	8.08
128	85	10.51	32	55.13	236	17.79
256	118	6.59	36	85.12	469	36.61
512	170	14.49	40	129.36	937	72.62

5.1.3 Cenários Com Falhas

Assim como feito para os casos sem falha, foram feitos testes com um processo e com n processos requisistando acesso à seção crítica.

Usando-se o gerador de quóruns em árvore (Tabela 5.3 e Tabela 5.4) para uma e duas falhas, o número médio de mensagens se aproxima do tamanho do qúorum. Conforme o número de processos cresce o número de mensagens cresce substancialmente em relação ao caso sem falhas. Isso se dá porque quando uma falha ocorre em um dado nodo n da árvore, os quóruns gerados que continham esse nó, passam a conter um caminho para esquerda e para a direita deste nó falho, percorrendo até a folha. Assim, com novos processos pertencendo ao quórum,

o processo requisitante precisa enviar novas mensagens de REQUEST, que por consequência gera novas mensagens de REPLY, FAILED, INQUIRE e YIELD. Com mais de uma falha, dada à forma que esses quóruns são construídos, pode ocorrer ainda de nós não falhos deixarem o quórum, fazendo com que o nó requisitante tenha que enviar mensagens de CANCEL para esses nós.

Tabela 5.3: Agrawal - Um requisitante

Falhas	1		2		N/2	
Processos	$\bar{x}$	σ	$\bar{x}$	σ	$\bar{x}$	σ
8	12	30.79	11	28.44	14	36.05
16	16	61.09	19	72.49	15	57.42
32	22	121.58	22	121.58	18	99.46
64	28	221.39	28	221.39	43	340.01
128	34	382.41	34	382.41	57	640.90
256	40	638.11	40	638.11	87	1387.94
512	46	1039.32	46	1039.32	104	2349.67

Para um número de falhas de $N/2$ o número médio de mensagens enviadas chega a ser mais do que o dobro como nos casos com 64, 256 e 512 processos. Isso se deve ao fato de que com o gerador de quóruns em árvore, quanto maior o número de falhas, maiores tendem a ser os qúoruns (já que pra cada nó falho ele percorre um caminho para esquerda e para a direita) e de que com mais falhas, mais mensagens precisam ser enviadas para reorganizar o sisteam.

Tabela 5.4: Agrawal - N requisitantes

Falhas	1		2		N/2	
Processos	$\bar{x}$	σ	$\bar{x}$	σ	$\bar{x}$	σ
8	18	6.18	19	7.99	25	19.63
16	22	11.71	28	13.16	55	42.15
32	25	20.36	28	21.51	52	43.41
64	29	34.15	32	36.10	69	63.34
128	33	55.03	36	55.38	91	89.82
256	37	83.64	40	84.50	116	133.09
512	41	128.25	44	128.89	128	184.60

Com o gerador de qúoruns em Grade de Maekawa construindo os quóruns da forma que descrita no começo da Seção 4.4, o tamanho dos qúoruns podem vir a diminuir e variar consideravelmente conforme o número de falhas aumenta, uma vez que em uma mesma coluna da grade o mesmo processo pode aparecer mais de uma vez. Assim, conforme o número de falhas

cresce o algoritmo tende a uma situação na qual a propriedade de minimalidade (Seção 3.4) não é mais garantida pois os quóruns passam a conter todos os processos do sistema. Na Tabela 4.5, observamos que com um único requisitante, devido a essa tendência de diminuir o tamanho dos qúoruns, o número médio de mensagens enviadas varia pouco.

Tabela 5.5: Maekawa - Um requisitante

Falhas	1		2		N/2	
Processos	$\bar{x}$	σ	$\bar{x}$	σ	$\bar{x}$	σ
8	15	38.40	16	41.03	15	38.69
16	21	80.03	27	102.83	19	72.49
32	33	182.23	38	209.78	37	204.47
64	45	355.70	45	355.70	47	371.65
128	66	742.22	66	742.22	79	888.61
256	93	1483.52	129	2057.68	100	1595.28
512	135	3050.09	135	3004.34	164	3705.43

Já em casos com n processos requisitando acesso à seção crítica, o número médio de mensagens enviadas a cada falha cresce, isso se deve ao fato de que, para cada falha que ocorre, os processos enviam novas mensagens de REQUEST (aumentando o número de mensagens enviadas em geral) e passam a enviar também mensagens de CANCEL. Conforme o número de falhas vai aumentando, aumenta também o número de mudanças nos quóruns, acarretando assim em um acentuado crescimento no número de mensagens enviadas. Com um requisistante esse aumento acentuado não se evidencia, já que todas essas mudanças ocorrem em apenas um quórum.

Tabela 5.6: Maekawa - N requisitantes

Falhas	1		2		N/2	
Processos	$\bar{x}$	σ	$\bar{x}$	σ	$\bar{x}$	σ
8	22	4.84	25	8.25	32	24.26
16	30	4.83	42	10.85	58	43.94
32	52	7.69	59	10.52	164	126.67
64	68	7.44	65	10.05	352	269.72
128	103	14.11	123	19.17	1024	802.15
256	136	12.88	164	16.44	2286	1769.79
512	199	21.00	235	24.96	4572	3538.58

Usando o gerador de quóruns no VCube proposto por Rodrigues[Rodrigues 2014], assim como no caso do algoritmo de Maekawa, o tamanho dos quóruns podem vir a diminuir com

o aumento no número de processos falhos no sistema (Tabela 5.7). Porém, quando o número de requisitantes e falhas aumenta, o número de mensagens também aumenta. Diferentemente do algoritmo de Maekawa, o algoritmo de Rodrigues apresenta um número menor de mudanças nos quóruns. Isso se dá pela forma que ele se comporta em casos de falha. Na maioria dos casos, o algoritmo substitui um processo falho por um outro processo não falho. Ocorre também de apenas remover o processo falho dos quóruns aos quais ele pertencia, fazendo com que hajam quóruns de menor tamanho que a média. Há ainda casos onde processos não falhos deixam de fazer parte de um ou mais quóruns, causando um aumento no número médio de mensagens enviadas.

Tabela 5.7: Rodrigues - 1 requisitante

Falhas	1		2		N/2	
Processos	$\bar{x}$	σ	$\bar{x}$	σ	$\bar{x}$	σ
8	22	55.80	14	36.05	13	33.42
16	27	102.83	29	110.55	22	84.08
32	52	287.11	51	281.55	41	226.71
64	99	782.42	100	790.35	76	601.24
128	195	2192.80	197	2215.33	145	1631.34
256	388	6189.17	387	6173.21	283	4515.17
512	771	17419.25	564	12753.124	560	12653.49

O desvio padrão é grande para ambos os tipos de quórum (Maekawa e Rodrigues) pois em ambos, dadas às mudanças que ocorrem em alguns quóruns, e dado que processos falhos deixam de enviar mensagens, alguns processos terminam enviando um número um pouco maior de mensagens do que a maioria. Neste sentido, o algoritmo de Rodrigues possui um desempenho melhor que o de Maekawa, já que em um cenário com um grande número de falhas ele apresenta um número menor de mudanças em seus quóruns.

O algoritmo de exclusão mútua baseado em sistema de quóruns tem um número de mensagens diretamente dependente do tamanho dos quóruns gerados e das mudanças nesses quóruns quando falhas ocorrem no sistema. Em cenários com apenas um processo requisistante e apenas um processo falho no sistema se evidencia a diferença nos tamanhos dos quóruns gerados por cada um dos algoritmos (Figura 5.3). O algoritmo de Agrawal gera os menores quóruns, seguido pelo de Maekawa. Existem casos em que o algoritmo de Agrawal gera quóruns maiores [Rodrigues 2014], o que acarretaria em um gradativo aumento no número de mensagens envi-

Tabela 5.8: Rodrigues - N requisitantes

Falhas	1		2		N/2	
Processos	$\bar{x}$	σ	$\bar{x}$	σ	$\bar{x}$	σ
8	21	4.12	26	9.14	29	21.01
16	35	5.42	36	7.65	49	35.78
32	63	7.13	65	9.29	97	70.80
64	121	10.84	121	14.91	181	133.24
128	238	20.89	240	24.86	365	265.80
256	471	39.95	473	41.12	703	514.67
512	939	77.07	939	80.21	1447	1055.01

adas, porém, em geral e para poucas falhas, o tamanho dos quórum varia pouco, mantendo o número de mensagens enviadas baixo. Em cenários com um número muito grande de falhas (no caso, $n/2$ falhas) os quóruns gerados pelo gerador de quóruns em árvore passa a gerar quóruns maiores e como podemos observar na Figura 5.3, tem um aumento acentuado no número de mensagens enviadas. Já para os algoritmos de Maekawa e Rodrigues, que podem gerar quóruns de menor tamanho em cenários com mais falha,s observamos uma redução no número médio de mensagens enviadas.

Quando mais de um processo concorre por acesso a seção crítica a diferença entre os diferente geradores de quórum se acentua ainda mais (Figura 5.3). Com apenas um processo falho no sistema com o geradores de quóruns em árvore e em grade ainda mantem o baixo número de mensagens enviadas por seção crítica obtida. Por outro lado os quóruns no VCube tem um aumento no número de mensagens já que, com mais processos requisitando seção crítica, os processos passam a enviar um maior número de mensagens, uma vez que precisam resolver a concorrência pela seção crítica.

Com metade dos processos falhos o número de mensagens enviadas utilizando-se do gerador de quórum de Maekawa e Rodrigues aumenta ainda mais. Como comentado, para testes com um processo requisitante apenas, em ambos os algoritmos ocorrem mudanças nos quóruns que acarretam em um maior número de mensagens enviadas. Note que apesar do número de mensagens aumentar com os qúoruns no VCube, este é um aumento previsto, já que os quóruns gerados sempre foram maiores do que em relação aos outros algoritmos. No entanto, para este cenário o número de mensagens, quando usando o gerador de quóruns em grade de Maekawa, aumenta acentuadamente.

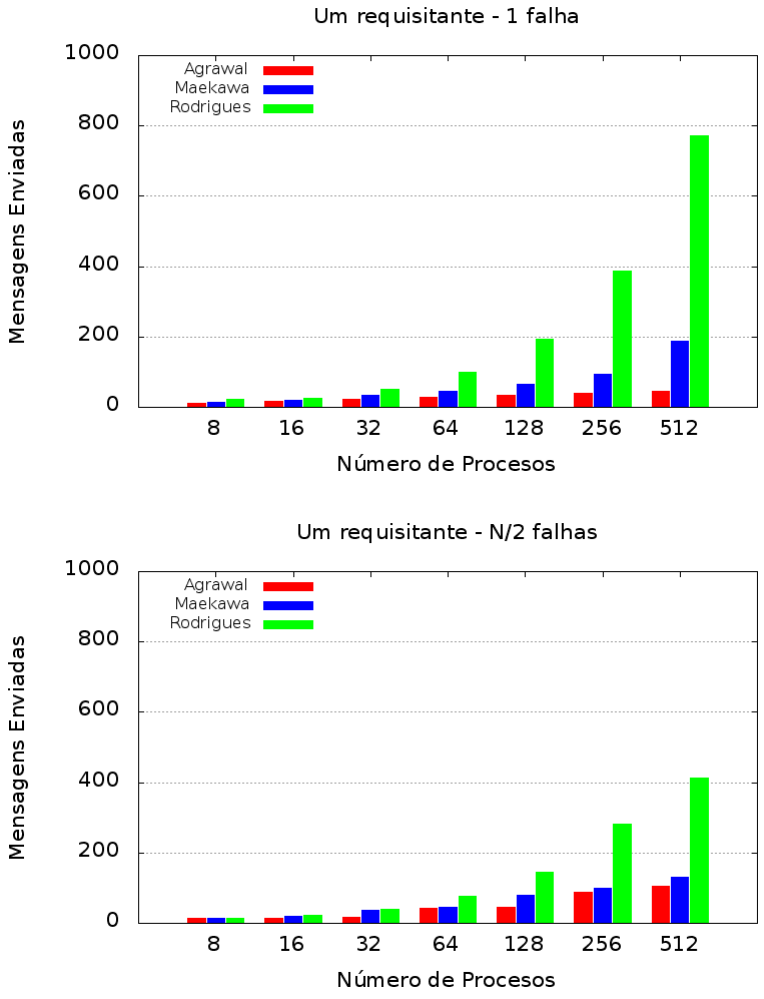


Figura 5.3: Um requisitante

Essa diferença enorme se explica com as mudanças que ocorrem nos quóruns e acarratam em novas mensagens de REQUEST, CANCEL e por consequência em um aumento no envio de todos os tipos de mensagem, principalmente as do tipo FAILED. Enquanto no gerador de quóruns baseado no VCube as mudanças que acarretam em muitas novas mensagens não ocorrem com tanta frequência, para o de Maekawa essas mudanças ocorrem a cada falha. Cada falha que acontece gera uma mudança de um ou mais processos, falhos e não falhos, no quórum de cada processo p_i. Em cenários anteriores onde havia apenas um processo essa mudanças no quóruns era menos evidente pois ocorria em apenas um quórum e dependendo de quais processos falhavam, um número consideravelmente menor de mudanças ocorriam no quorum do proceso requisitante. Já aqui, cada processo p não falho (no caso 256 processos) tem seu quórum alterado, alguns com mais alterações do que outros e precisa enviar novas mensagens para

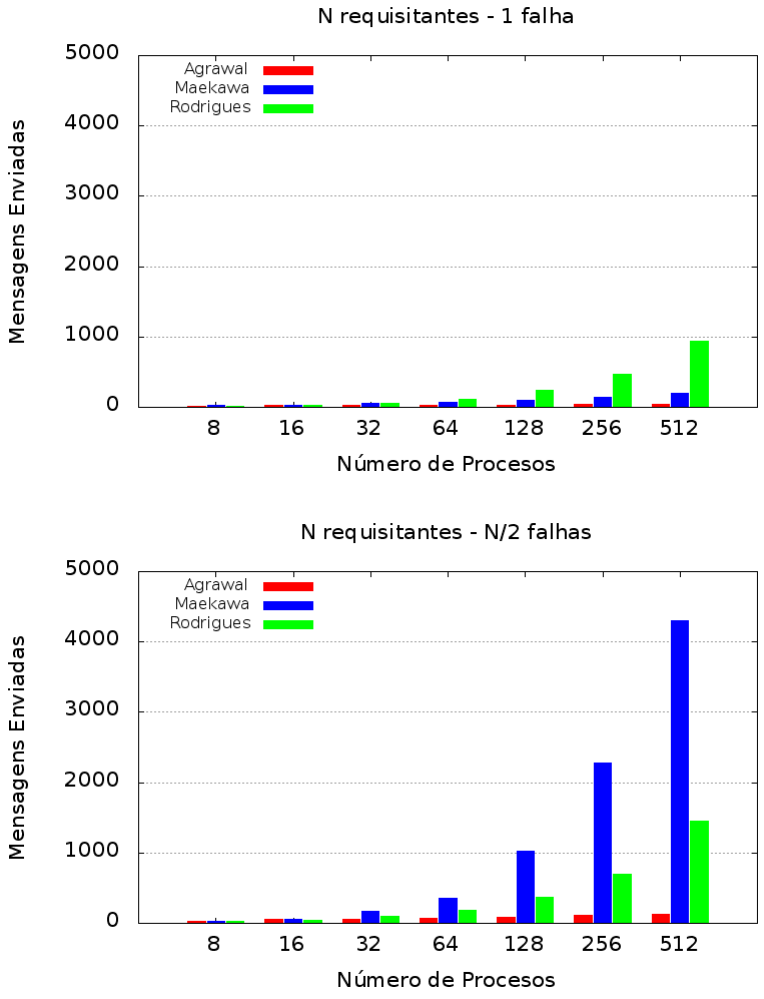


Figura 5.4: N requisitantes

reorganizar o sistema.

Esse comportamento se deve exclusivamente a forma com o gerador de quóruns em Grade foi adaptado para tolerar falhas. Esta foi a maneira encontrada para, com um grande número de falhas aleatórias, garantir que as propriedades definidas na Seção 3.4. fossem mantidas até pelo menos $n/2$ falhas.

5.1.4 Considerações Finais

Avaliando-se o número total de mensagens enviadas por seção crítica fica evidente que o algoritmo de quóruns em árvore envia um número total de mensagens bem menor do que com os quóruns gerados pelos outros algoritmos. Apesar disso, há de se considerar que, mesmo havendo casos em que o algoritmo suporte até $n - |log\, n|$ falhas, em geral, conforme aumenta o

número de processos falhos no sistema cresce, crescem também as chances dos processos falhos serem folhas na árvore, e por consequência, crescem as chances do algoritmo não conseguir formar nenhum quórum. Mesmo para o propósito deste trabalho, onde foram geradas falhas aleatórias em tempos aleatórios, foi preciso encontrar um cenário funcional para $n/2$ processos falhos já que, dada a aleatoriedade dos testes gerados, era muito comum ocorrerem cenários onde após um número de falhas, mais nenhum quórum era gerado. O número menor de processo em um quórum termina se refletindo em uma menor tolerância a falhas. Por outro lado, apesar de gerar os maiores quóruns, o algoritmo de quóruns no VCube não apresenta este problema, tolerando falhas em até $n-1$ processos, sendo então mais adequado para uso em situações onde uma maior tolerância a falhas seja mais crítica do que o número total de mensagens trocadas pelos nodos do sistema.

Capítulo 6

Conclusão

Neste trabalho foi proposto um algoritmo de exclusão mútua com quóruns no VCube. O algoritmo proposto usa do VCube para fazer o monitoramento do estado de cada processo e utiliza da estrutura e propriedades dos qúoruns para garantir a exclusão mútua entre processos que disputam um mesmo recurso. Para que a tolerância a falha do algoritmo de exclusão mútua funcione foi preciso que o gerador de quóruns suportasse também a tolerância à falhas.

Dois dos algoritmos escolhidos possuem tolerância a falhas e um deles foi adaptado de forma a tolerar falhas sem perder as propriedades necessárias em um sistema de qúoruns. O algoritmo foi testado para cenários de até $n/2$ falhas com n processos requisitando acesso à seção crítica e foi capaz de garantir a exclusão mútua em todos os cenários de teste.

Dos geradores de quóruns utilizados, o gerador de quóruns em árvore de Agrawal-El-Abbadi resultou em um menor número de mensagens enviadas por seção crítica, porém tem uma tolerância a falhas menor e muito dependente de cenários específicos onde as falhas não ocorram em processos específicos. O gerador de quóruns em grade de Maekawa precisou ser adaptado e, apesar de ser o segundo gerador com os menores quóruns, dada a adaptação feita para que o mesmo fosse tolerante a falhas, o número de mensagens aumenta acentuadamente com o aumento de falhas e de processos concorrendo por um recurso no sistema. Por fim, utilizando-se do gerador de qúoruns no VCube de Rodrigues, apesar de serem gerados os maiores quóruns, permite-se uma maior tolerância a falhas que Agrawal.

Observou-se também que apesar de existirem diferentes algoritmos de exclusão mútua usando sistema de quóruns, a maior influência no desempenho (número de mensagens trocadas) e na tolerância à falhas dependem dieretamente da forma com que os quóruns são gerados, do tamanho dos quóruns e do grau de tolerância a falhas do sistema de quóruns. Assim sendo,

apesar da necessidade da exclusão mútua se adaptar à sistemas com falhas, os verdadeiros ganhos vem do gerador de quóruns.

Para trabalhos futuros propõe-se que avalie a possibilidade de remoção da mensagem de CANCEL nos casos em que um process não falho deixa um dado quórum. A remoção desta mensagem levaria a uma redução considerável no número de mensagens enviadas em sistemas com nodos falhos, principalmente utilizando-se os geradores de quórum de Maekawa e de Rodrigues.

Referências Bibliográficas

[Coulouris, Dollimore e Kindberg 2005]COULOURIS, G. F.; DOLLIMORE, J.; KINDBERG, T. *Distributed systems : concepts and design*. Harlow, England, New York: Addison-Wesley, 2005. (International computer science series). ISBN 0-321-26354-5. Disponível em: <http://opac.inria.fr/record=b1102391>.

[Divyakant e Amr 1991]DIVYAKANT, A.; AMR, E. A. An efficient and fault-tolerant solution for distributed mutual exclusion. *ACM Trans. Comput. Syst.*, ACM, New York, NY, USA, v. 9, n. 1, p. 1–20, fev. 1991.

[Dwork e Lynch 1988]DWORK; LYNCH. Consensus in the presence of partial synchrony. *Journal of the ACM(JACM)*, New York, USA, v. 35, n. 2, p. 288–323, Abril 1988.

[Fischer et al. 1985]FISCHER et al. Impossibility of distributed consensus with one faulty process. *J. ACM*, ACM, New York, NY, USA, v. 32, n. 2, p. 374–382, abr. 1985. ISSN 0004-5411. Disponível em: <http://doi.acm.org/10.1145/3149.214121>.

[Jalote 1994]JALOTE, P. *Fault Tolerance in Distributed Systems*. 1. ed. Reading: Prentice Hall, 1994.

[Kshemkalyani e Singhal 2008]KSHEMKALYANI, A.; SINGHAL, M. *Distributed Computing: Principles, Algorithms and Systems*. 1. ed. Reading: Cambridge, 2008.

[L. 1978]L., L. Time, clocks and the ordering of events in a distributed system. In: *Commun. ACM*. [S.l.: s.n.], 1978. p. 558–565.

[M. 1991]M., R. A simple taxonomy for distributed mutual exclusion algorithms. *SIGOPS Oper. Syst. Rev.*, v. 25, p. 47–50, Abril 1991.

[Maekawa e Mamoru 1985]MAEKAWA; MAMORU. A n algorithm for mutual exclusion in decentralized systems. *ACM Trans. Comput. Syst.*, ACM, New York, NY, USA, v. 3, n. 2, p. 145–159, maio 1985.

[P., E. e Ruoso 2014]P., D. J. E.; E., L. C.; RUOSO. Vcube: A provably scalable distributed diagnosis algorithm. In: *Proceedings of the 5th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*. Piscataway, NJ, USA: IEEE Press, 2014. (ScalA '14), p. 17–22. ISBN 978-1-4799-7562-4. Disponível em: <http://dx.doi.org/10.1109/ScalA.2014.14>.

[Raymond 1989]RAYMOND. A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems (TOCS)*, New York, USA, v. 7, n. 1, p. 61–77, Fevereiro 1989.

[Raynal 2013]RAYNAL, M. *Distributed algorithms for message-passing systems*. [S.l.]: Springer, 2013.

[Ricart e Agrawala 1981]RICART; AGRAWALA. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, New York, USA, v. 24, n. 1, p. 9–17, Janeiro 1981.

[Rodrigues, Jr. e Arantes 2015]RODRIGUES; JR., D.; ARANTES e. Um serviço multicast confiável hierárquico com o vcube. In: *XXXIII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos, Anais do WTF*. Vitória - BR: [s.n.], 2015. p. 71–84.

[Rodrigues 2014]RODRIGUES, L. A. *Uma Solução Autônoma para K-Exclusão Mútua em Sistemas Distribuídos*. Tese (Tese de Doutorado) — Universidade Federal do Paraná, Curitiba, PR, Agosto 2014.

[Rodrigues, Jr e Arantes]RODRIGUES, L. A.; JR, E. P. D.; ARANTES, L. Uma soluç ao autonômica para a criaç ao de quóruns majoritários baseada no vcube.

[Singhal 1992]SINGHAL. A dynamic information-structure mutual exclusion algorithm for distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, Piscataway, NJ, USA, v. 3, n. 1, p. 121–125, Janeiro 1992.

[Suzuki e Kasami 1985]SUZUKI; KASAMI. A distributed mutual exclusion algorithm. *ACM Transactions on Computer Systems (TOCS)*, New York, USA, v. 3, n. 4, p. 344–349, Novembro 1985.

[Tanenbaum e Bos 2014]TANENBAUM; BOS. *Modern operating systems*. [S.l.]: Prentice Hall Press, 2014.

[Tanenbaum e Steen 2007]TANENBAUM, A.; STEEN, M. *Distributed Systems: Principles and Paradigms*. 2. ed. Reading: Prentice Hall, 2007.

[Urban, Defago e Schiper 2001]URBAN, P.; DEFAGO, X.; SCHIPER, A. Neko: A single environment to simulate and prototype distributed algorithms. In: *In Proc. of the 15th Int'l Conf. on Information Networking (ICOIN-15*. [S.l.: s.n.], 2001. p. 503–511.