UNIVERSIDADE FEDERAL DO PARANÁ

ACÁCIA DOS CAMPOS DA TERRA

INVESTIGANDO A IMPLEMENTAÇÃO DO CONSENSO ESCALÁVEL SOBRE O VCUBE

CURITIBA PR

2020

ACÁCIA DOS CAMPOS DA TERRA

INVESTIGANDO A IMPLEMENTAÇÃO DO CONSENSO ESCALÁVEL SOBRE O VCUBE

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Informática no Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Prof. Dr. Elias Procópio Duarte Jr.
Coorientador: Prof. Dr. Edson Tavares de Camargo.

CURITIBA PR

2020

Catalogação na Fonte: Sistema de Bibliotecas, UFPR
Biblioteca de Ciência e Tecnologia

T323i Terra, Acácia dos Campos da
Investigando a implementação do consenso escalável sobre o VCUBE [recurso eletrônico] / Acácia dos Campos da Terra. – Curitiba, 2020.

Dissertação - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática, 2020.

Orientador: Elias Procópio Duarte Junior.
Coorientador: Edson Tavares de Camargo.

1. Algorítmos computacionais. 2. Falhas de sistemas de computação. 3. Sistemas operacionais distribuídos (Computadores). I. Universidade Federal do Paraná. II. Duarte Junior, Elias Procópio. III. Camargo, Edson Tavares de. IV. Título.

CDD: 005.4476

Bibliotecária: Vanusa Maciel CRB- 9/1928

MINISTÉRIO DA EDUCAÇÃO
SETOR DE CIENCIAS EXATAS
UNIVERSIDADE FEDERAL DO PARANÁ
PRÓ-REITORIA DE PESQUISA E PÓS-GRADUAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO INFORMÁTICA - 40001016034P5

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da dissertação de Mestrado de **ACÁCIA DOS CAMPOS DA TERRA** intitulada: **INVESTIGANDO A IMPLEMENTAÇÃO DO CONSENSO ESCALÁVEL SOBRE O VCUBE**, sob orientação do Prof. Dr. ELIAS PROCÓPIO DUARTE JUNIOR, que após terem inquirido a aluna e realizada a avaliação do trabalho, são de parecer pela sua APROVAÇÃO no rito de defesa.
A outorga do título de mestre está sujeita à homologação pelo colegiado, ao atendimento de todas as indicações e correções solicitadas pela banca e ao pleno atendimento das demandas regimentais do Programa de Pós-Graduação.

CURITIBA, 17 de Abril de 2020.

Assinatura Eletrônica
17/04/2020 16:28:58.0
ELIAS PROCÓPIO DUARTE JUNIOR
Presidente da Banca Examinadora

Assinatura Eletrônica
19/04/2020 18:46:07.0
LUIZ CARLOS PESSOA ALBINI
Avaliador Interno (UNIVERSIDADE FEDERAL DO PARANÁ)

Assinatura Eletrônica
20/04/2020 03:07:19.0
FERNANDO PEDONE
Avaliador Externo (UNIVERSITà DELLA SVIZZERA ITALIANA)

Assinatura Eletrônica
17/04/2020 11:06:48.0
EDSON TAVARES DE CAMARGO
Avaliador Externo (UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ)

AGRADECIMENTOS

Agradeço primeiramente ao meu orientador, professor Elias, e ao meu coorientador, Edson: sem eles o trabalho não teria sido possível. O trabalho evoluiu demais desde a primeira conversa que tive com o professor Elias, ainda durante a disciplina de Sistemas Distribuídos, e tivemos um grande avanço com a reunião que fizemos em conjunto com o Edson, onde finalmente entendemos como o Paxos funciona! Meu muito obrigada, vocês são incríveis e admiro muito o trabalho de vocês.

Agradeço também aos membros da minha banca de qualificação e de defesa do trabalho, os quais trouxeram excelentes sugestões e observações para o trabalho. São eles: Giovanni Venâncio, Fernando Pedone e Luiz Carlos Albini.

Um muito obrigada ao professor Wagner Zola, que foi o primeiro professor da UFPR a acreditar em mim, e me aceitar inicialmente no programa de pós-graduação como sua orientanda.

Alguns amigos foram essenciais no andamento deste trabalho, me ajudando com a adaptação em Curitiba, apoio emocional ou até mesmo com algumas dúvidas de programação. Agradeço demais a vocês, João Pedro, Gabriel Galli, Hugo Takiuchi, Priscila Delabetha, José Wilson, Felipe Chabatura (Ninja), Leandro Zatesko e Giullia Stefanini.

Agradeço aos meus pais, Telma e Antonio Carlos, e também aos meus tios Vera Lúcia e Sérgio. Vocês me apoiaram em um momento onde a educação não está sendo muito valorizada, e nós todos acreditamos no poder transformador da educação. Muito obrigada por me ajudarem na empreitada de tornar o mundo melhor através da educação.

Um muito obrigada muito especial ao meu namorado, Guilherme Konopatzki, que me apoiou de todas as formas possíveis, presenciou alguns picos de ansiedade e todas as etapas que precisei ultrapassar para chegar ao final deste mestrado. Obrigada pelo seu apoio e carinho durante todo esse tempo. Obrigada por me ajudar a perseverar.

Por fim, agradeço a Capes pelo apoio financeiro que recebi através da bolsa de mestrado.

RESUMO

Um dos algoritmos essenciais na construção de sistemas distribuídos é o consenso, que garante que todos os processos corretos do sistema decidem por um mesmo valor dentre valores que foram propostos. O Paxos é um dos mais importantes algoritmos de consenso existentes. O Paxos tolera falhas por parada, garantindo a propriedade da segurança (*safety*) e, sob condições de assincronia fraca, garante o progresso (*liveness*). O Paxos foi proposto originalmente para a replicação máquina de estado. Variações do algoritmo, com destaque para o Ring Paxos, foram propostas com o intuito de melhorar o desempenho do sistema. O Ring Paxos executa o Paxos dispondo os processos em um anel lógico direcional. Neste trabalho é proposto um algoritmo de consenso escalável que implementa uma instância do Paxos. No algoritmo os processos *acceptors* são organizados na topologia virtual para sistemas distribuídos conhecida como VCube. O VCube organiza os processos em um hipercubo quando todos estão sem-falha, sendo um algoritmo que apresenta diversas propriedades logarítmicas mesmo quando há processos falhos. No algoritmo proposto, os *acceptors* são agrupados em *clusters*, e o coordenador executa o Paxos utilizando uma difusão de mensagens de melhor esforço sobre o VCube. Inicialmente o algoritmo tenta alcançar a maioria apenas no maior *cluster*. Caso haja processos falhos o algoritmo continua nos demais *clusters* até que obtenha uma maioria de *acceptors*. Conforme um *acceptor* encaminha a mensagem recebida para o próximo *acceptor* no *cluster*, concatena junto à mensagem a sua resposta para o coordenador. A mensagem percorre o ramo da árvore gerada pelo VCube e, sempre que um *acceptor* for folha, envia uma mensagem de resposta para o coordenador contendo as respostas concatenadas de todos os *acceptors* daquele ramo. Desta forma o coordenador sabe quando atingiu uma maioria entre os *N acceptors*. O algoritmo foi implementado através de simulação e foram obtidos resultados positivos para a quantidade de mensagens trocadas durante a execução do consenso, quando em comparação com uma versão do Paxos em anel inspirada no Ring Paxos. Nesta dissertação é apresentada uma especificação, implementação e os resultados obtidos através de simulação.

Palavras-chave: VCube. Sistema Distribuído. Consenso. Paxos. Tolerância a Falhas.

ABSTRACT

Consensus is one of the most important building blocks for building fault-tolerant distributed systems. The execution of consensus ensures that all the correct processes in the system decide the same value among a set of values initially proposed. Paxos is one of the most important consensus algorithms. Paxos tolerates crash faults, guarantees safety always, and liveness under weak synchrony assumptions. Paxos was originally proposed for state machine replication. Variations of the algorithm, in particular Ring Paxos, have been proposed in order to improve the performance of the system. Ring Paxos executes Paxos by organizing the processes on a directional logical ring. This work proposes a scalable consensus algorithm that implements an instance of Paxos. In the algorithm, acceptors are organized in a virtual topology for distributed systems known as VCube. The VCube is a hypercube when every process is correct, and presents several logarithmic properties even when processes fail. In the proposed algorithm, the acceptors are grouped in clusters, and the coordinator runs Paxos using a best effort broadcast algorithm that sends messages across the VCube. Initially the algorithm tries to reach a majority only in the largest cluster. In case there are faulty processes, the algorithm continues in the other clusters until it reaches a majority of acceptors. As an acceptor forwards the received message to the next acceptor in the cluster, it concatenates its response in the message that is sent up to the coordinator. The message travels through branches of the tree generated on the VCube and, whenever an acceptor is a leaf, it sends a reply message to the coordinator containing the concatenated responses of all acceptors of that branch. In this way as the coordinator receives the message it knows whether a majority among the N acceptors has replied. The algorithm was implemented through simulation and positive results were obtained for the amount of messages exchanged during the execution of the consensus, when compared to an equivalent Paxos version inspired on Ring Paxos. In this dissertation we present a specification of the algorithm, as well as an implementation and results obtained through simulation.

Keywords: VCube. Distributed System. Consensus. Paxos. Fault Tolerance

LISTA DE FIGURAS

2.1 Exemplo de execução do algoritmo de difusão de melhor esforço (Cachin et al., 2011) . . . 14
2.2 Exemplo de execução do algoritmo de difusão confiável. . . . 14

3.1 Exemplo de execução da fase 1 do algoritmo (de Camargo e Duarte Jr, 2017). . . 17
3.2 Exemplo de execução da fase 2 do algoritmo (de Camargo e Duarte Jr, 2017). . . 17
3.3 Execução da Fase 2 no M-Ring Paxos. . . . 19
3.4 Execução da Fase 2 no U-Ring Paxos. . . . 21

4.1 Exemplo de execução do algoritmo *Adaptive-DSD* com processos falhos.. . . . 24
4.2 Hipercubo com dimensão $d = 3$. . . . 25
4.3 Organização dos *clusters* para um sistema com 8 processos.. . . . 26
4.4 Organização dos *clusters* para um sistema com 8 processos e processos 1 e 7 falhos.28
4.5 Mensagens enviadas pelo processo 0.. . . . 30
4.6 Mensagens enviadas pelos demais processos. . . . 30
4.7 Mensagens enviadas quando os processos dois e quatro estão falhos. . . . 31

5.1 Exemplo com todos os processos sem-falha e $n = 8$.. . . . 34
5.2 Exemplo com um processo falho (4) e $n = 8$.. . . . 34
5.3 Exemplo com três processos falhos (2, 4, 7) e $n = 8$.. . . . 35

6.1 Quantidade total de mensagens trocadas no cenário sem-falhas.. . . . 37
6.2 Quantidade total de mensagens trocadas no cenário com $n/4 - 1$ falhas. . . . 38
6.3 Quantidade total de mensagens trocadas no cenário com máximo de falhas permitidas. . . . 38

SUMÁRIO

1 INTRODUÇÃO 8

2 SISTEMAS DISTRIBUíDOS: DEFINIÇÕES E ALGORITMOS FUNDAMENTAIS 10

2.1 MODELOS TEMPORAIS 10

2.2 MODELOS DE FALHA 10

2.3 CONSENSO 11

2.4 DETECTORES DE FALHAS 12

2.5 DIFUSÃO DE MELHOR ESFORÇO E DIFUSÃO CONFIÁVEL 13

3 PAXOS E RING-PAXOS 16

3.1 PAXOS 16

3.2 RING-PAXOS 18

4 VCUBE: UMA VISÃO GERAL 24

4.1 OS ALGORITMOS ADAPTIVE-DSD E HI-ADSD 24

4.2 O ALGORITMO DO VCUBE 26

4.2.1 Difusão Sobre o VCube 28

5 UMA INSTÂNCIA DO PAXOS SOBRE O VCUBE 32

6 IMPLEMENTAÇÃO E RESULTADOS 36

6.1 CENÁRIOS SIMULADOS 36

6.2 RESULTADOS EXPERIMENTAIS 36

7 CONCLUSÃO 40

REFERÊNCIAS 41

1 INTRODUÇÃO

Grande parte dos sistemas computacionais atuais são de natureza distribuída, ou seja, consistem de um conjunto de componentes distintos que se comunicam através da troca de mensagens e realizam cooperativamente tarefas diversas (Kshemkalyani e Singhal, 2008; Cachin et al., 2011; Mullender, 1993). Uma boa parte destes sistemas executa na própria Internet, uma rede em escala mundial com forte presença tanto para organizações como para indivíduos. A Internet está presente em, virtualmente, tudo: escritórios, escolas, empresas, governos, conectando aparelhos diversos como telefones, televisões e até relógios. Os sistemas e dispositivos estão se conectando cada vez mais uns com os outros, mas essa não é uma tarefa fácil, especialmente pela heterogeneidade e a complexidade de cada um deles. Apesar disso, a correta cooperação é de importância vital, pois muitas vezes os sistemas são utilizados para realizar missões críticas.

Neste trabalho, um sistema distribuído é definido como um conjunto de processos que se comunicam trocando mensagens através de canais de comunicação. Os processos podem falhar por parada (*crash*). Sistemas tolerantes a falhas devem manter o correto funcionamento mesmo após a falha de alguns processos. Garantir uma correta realização das tarefas é um desafio (Avizienis et al., 2004). Um dos problemas mais fundamentais de sistemas distribuídos é o consenso. Informalmente, o consenso precisa garantir que os processos entrem em acordo sobre um único valor dentre valores propostos, apesar de possíveis falhas. Em 1985 foi provado que o consenso é impossível em sistemas assíncronos sujeitos a falha por parada, ou seja, sistemas em que nenhum limite de tempo é conhecido (Fischer et al., 1985). Concretamente, a execução do consenso pode não terminar corretamente nestes tipos de sistema. A raiz deste problema é o fato de que em sistemas assíncronos é impossível distinguir quando um processo está falho de quando está apenas executando lentamente.

Posteriormente, em 1996, foram propostos os detectores de falhas não-confiáveis, uma abstração que permite investigar o consenso levando em conta que há como obter informação sobre as falhas. Os detectores de falhas são ditos não confiáveis porque podem cometer enganos. Os detectores de falhas atuam localmente em cada processo e fornecem informações sobre o estado de outros processos. Foram investigadas as propriedades dos detectores de falhas que interferem no consenso. Uma classificação para os detectores foi proposta baseada nestas propriedades. Os autores (Chandra e Toueg, 1996) mostraram que mesmo usando a classe de detectores de falhas mais fraca, o consenso em sistemas assíncronos sujeitos a falhas por parada é possível.

Uma forma dos processos determinarem os estados uns dos outros é se monitorando através da aplicação de testes. Um processo precisa responder corretamente a um teste dentro de um tempo limite pré-definido para ser considerado sem-falha, em caso contrário é considerado falho. O VCube é uma topologia virtual para sistemas distribuídos que provê um serviço de detecção de falhas. O VCube organiza os processos em um hipercubo quando todos estão sem-falha (Duarte Jr et al., 2014). Essa topologia é criada e mantida de acordo com as informações de monitoramento obtidas através dos testes executados entre os processos. Os processos são organizados em *clusters* e os testes são realizados em *clusters* progressivamente maiores. O algoritmo do VCube apresenta diversas propriedades logarítmicas.

Foi proposto em (Rodrigues et al., 2014) um algoritmo de difusão confiável hierárquica sobre o VCube. É muito comum em sistemas distribuídos um processo enviar uma mensagem para todos os outros processos do sistema. Esse tipo de comunicação é chamado *broadcast* ou

difusão. Em um sistema em que processos falham a difusão confiável pode ser utilizada, ela garante que todos os processos corretos entregam a mensagem transmitida mesmo que houver processos falhos. Outra difusão largamente utilizada é a difusão de melhor esforço, nela a entrega não é garantida caso o processo fonte falhe. Um algoritmo de difusão de melhor esforço sobre o VCube também foi definido em (Rodrigues et al., 2014). Os algoritmos de difusão sobre o VCube, tanto de difusão confiável quanto de melhor esforço, utilizam a estrutura oferecida pela topologia para permitir que a difusão seja escalável.

O Paxos é um algoritmo de consenso tolerante a falhas para sistemas assíncronos sujeitos a falhas por parada e recuperação (*crash-recovery*) e é um dos algoritmos de consenso mais importantes, sendo muito utilizado por empresas como o Google (Chandra et al., 2007) e a Microsoft (Isard, 2007), entre diversas outras. O Paxos garante a propriedade de segurança (*safety*), e sob condições de assincronia fraca, garante a propriedade de progressão (*liveness*). No Paxos, os processos assumem papéis, que podem ser: *proposer*, *acceptor* e *learner*. Um processo *proposer* propõe valores para o consenso, processos *acceptors* aceitam um valor e *learners* aprendem o valor decidido pelo consenso.

Um processo também é selecionado como *coordenador*, este processo define a proposta a ser executada. Embora um processo coordenador seja importante para manter a progressão do consenso, causa um gargalo na execução do algoritmo. Isso porque o coordenador troca mensagens ponto a ponto com cada processo em ambas as fases do Paxos. Algumas variações do Paxos foram propostas para reduzir a carga do coordenador, uma delas é o Ring Paxos (Jalili Marandi et al., 2017), que propõe a disposição dos processos em um anel lógico direcional. A estratégia apresenta bons resultados ao executar várias instâncias do Paxos ao mesmo tempo.

Neste trabalho é proposto um algoritmo de consenso escalável sobre o VCube. O algoritmo executa uma instância do Paxos e, nele, apenas os processos *acceptors* são organizados na topologia VCube. Os *acceptors* são agrupados em *clusters*, e o coordenador executa o Paxos utilizando a difusão de mensagens de melhor esforço sobre o VCube. Inicialmente o algoritmo tenta alcançar a maioria apenas no maior *cluster* (de tamanho $\log n$). Caso não haja processos falhos neste *cluster*, o algoritmo tem a maioria, pois o próprio coordenador é também um *acceptor*. Caso haja processos falhos no *cluster*, o algoritmo parte para o *cluster* de tamanho $\log n - 1$. O algoritmo continua até que obtenha uma maioria de *acceptors*. Conforme um *acceptor* encaminha a mensagem recebida para o próximo *acceptor* no *cluster*, concatena junto à mensagem a sua resposta para o coordenador. A mensagem percorre o ramo da árvore gerada pelo VCube e, sempre que um *acceptor* for folha, envia uma mensagem de resposta para o coordenador contendo as respostas concatenadas de todos os *acceptors* daquele ramo. Desta forma o coordenador sabe quando atingiu uma maioria entre os N *acceptors*. O algoritmo foi implementado através de simulação e foram obtidos resultados positivos para a quantidade de mensagens trocadas durante a execução do consenso, quando em comparação com uma versão do Paxos baseada em anel inspirada no Ring-Paxos. Nesta dissertação é apresentada uma especificação, implementação e os resultados obtidos através de simulação.

O restante deste trabalho está organizado da seguinte forma. O Capítulo 2 apresenta definições e algoritmos fundamentais de sistemas distribuídos que serão utilizados ao longo do trabalho. O Capítulo 3 apresenta o Paxos e o Ring Paxos. O Capítulo 4 descreve o VCube, seguido do Capítulo 5 que apresenta o algoritmo proposto neste trabalho. Por fim, o Capítulo 6 apresenta os resultados e comparações obtidos e o Capítulo 7 apresenta a conclusão.

2 SISTEMAS DISTRIBUÍDOS: DEFINIÇÕES E ALGORITMOS FUNDAMENTAIS

Um sistema distribuído é definido como um conjunto de processos que cooperam entre si para realizar uma determinada tarefa (Cachin et al., 2011; Kshemkalyani e Singhal, 2008). A comunicação entre os processos se dá ou por meio do envio e recebimento de mensagens ou através de memória compartilhada. Neste trabalho, consideramos apenas o paradigma de troca de mensagens. Neste capítulo são abordadas as definições de modelos temporais, modelos de falha, consenso, detectores de falhas e difusão. Estas definições serão posteriormente utilizadas ao longo deste trabalho.

2.1 MODELOS TEMPORAIS

Uma característica muito importante de um sistema distribuído é relacionada ao comportamento e gestão de tempo por parte dos processos (Cachin et al., 2011). Em um sistema síncrono existe um limite superior conhecido em que um processo irá executar uma determinada tarefa e também para o tempo de transmissão de uma mensagem. Na prática, esses limites permitem o uso de *timeouts*, que são parâmetros que estabelecem um tempo máximo de resposta para uma mensagem. *Timeouts* são frequentemente utilizados para gerenciar o tempo em sistemas síncronos. Já um sistema assíncrono é caracterizado pela ausência de qualquer conhecimento temporal. Há também os sistemas parcialmente síncronos, onde são definidos níveis intermediários de sincronismo (Dwork et al., 1988a).

Um sistema síncrono é mais restrito e não reflete a realidade da maioria dos ambientes reais, pois conhecer limites de tempo não é uma tarefa simples. Os sistemas assíncronos são mais fáceis de implantar em ambientes reais, porém, devido à falta de informação em relação a tempo de resposta dos processos, se torna impossível, por exemplo, distinguir se um processo está muito lento ou se está falho. Em certos contextos, um engano desta natureza pode causar inconsistências.

Entre os diversos modelos de sistemas parcialmente síncronos, há um destaque para o modelo baseado em GST (*Global Stabilization Time*), proposto em (Dwork et al., 1988b), onde os limites temporais são conhecidos, mas somente após um tempo de estabilização GST, que tem duração finita, mas desconhecida para os processos.

2.2 MODELOS DE FALHA

Uma aplicação distribuída é especificada para realizar determinadas tarefas, mas quando uma falha ocorre em algum processo, comportamentos inconsistentes surgem e podem gerar resultados incorretos. Apesar disso, é essencial garantir que o sistema continue operacional, mesmo que alguns processos falhem. Por isso, os sistemas devem satisfazer a propriedade de tolerância a falhas (*dependability*), de forma a garantir a execução correta do sistema em qualquer situação (Avizienis et al., 2004).

A falha (*fault*) de algum processo é uma possível causa de um erro (*error*), e o erro é a manifestação da falha. Um erro pode vir a causar uma falha no serviço (*failure*), que é identificada através de um resultado final incorreto, tendo em vista a especificação do sistema (Avizienis et al., 2004). Os processos podem falhar em diversos cenários diferentes, por isso são encontrados na literatura vários modelos de falha que especificam o modo pelo qual os processos falham (Cachin

et al., 2011). Alguns dos principais modelos de falha são: parada, omissão e bizantina, descritos a seguir.

No modelo de falha por parada (*crash*) há uma parada completa do processo, que não executa nenhuma computação local e também não envia mensagens para os outros processos. No modelo de falha *crash* simples, após a falha, o processo nunca se recupera. Uma variação é o modelo de falha por parada e recuperação (*crash-recovery*), onde os processos podem se recuperar após uma falha. Uma característica essencial deste modelo é o uso de memória secundária, onde são armazenadas as informações de estado que o processo possui enquanto ele está sem-falha e ao sofrer uma falha, essa informação sobre o estado interno não é perdida. Quando efetuar a recuperação, as informações são atualizadas de acordo com o que estava armazenado na memória secundária. No modelo de falha por omissão o processo não envia (ou recebe) uma ou mais das mensagens que deveria. Já a falha bizantina apresenta comportamento arbitrário, o processo pode enviar qualquer tipo de resposta. Esse comportamento pode ser causado por um erro na implementação do sistema ou por alguma ação maliciosa. A falha bizantina é a mais abrangente, ela engloba também as falhas por parada e por omissão. Analogamente, a falha por omissão engloba a por parada.

2.3 CONSENSO

Processos de um sistema distribuído muitas vezes precisam entrar em acordo em relação a alguma informação ou ação executada no sistema. Assuma um conjunto de processos que pode propor valores. O algoritmo de consenso garante que todos os processos corretos *decidem* por um único valor dentre os valores que foram inicialmente propostos (Lamport, 2001). Essas ações são disparadas através das primitivas *propose* e *decide*.

O algoritmo de consenso precisa satisfazer as seguintes propriedades (Cachin et al., 2011):

- Terminação: todo processo correto irá, em algum momento, decidir por um valor;
- Validade: se um processo decide pelo valor v, então v foi proposto por algum processo;
- Integridade: nenhum processo decide duas vezes;
- Acordo: dois processos corretos não decidem por valores diferentes.

A terminação é uma propriedade de progressão (*liveness*), a validade, a integridade e o acordo são propriedades de segurança (*safety*).

Uma classificação do consenso é em consenso regular e consenso uniforme (Cachin et al., 2011). No caso do consenso uniforme, processos corretos devem decidir por um valor que seja consistente com o valor decidido por processos que possam ter decidido antes de falhar. Enquanto o consenso regular garante que dois processos corretos não decidem por valores diferentes, o consenso uniforme garante que dois processos quaisquer não decidem por valores diferentes, estejam eles corretos ou não.

Em 1985 foi publicado um dos mais importantes resultados da teoria de sistemas distribuídos. Esse resultado (Fischer et al., 1985), conhecido como a impossibilidade FLP, provou que é impossível alcançar o consenso em um sistema assíncrono sujeito a falhas por parada. Isso porque é impossível neste tipo de sistema que um processo faça a distinção se um determinado processo falhou ou se está apenas executando muito lentamente. Sem ter certeza do estado de um processo, o sistema assíncrono não pode garantir que são satisfeitas a terminação (*liveness*) e o acordo (*safety*) ao mesmo tempo.

A partir de então vários algoritmos passaram a ser propostos para um modelo parcialmente síncrono ou assumindo a presença de um detector de falhas. Em modelos parcialmente síncronos o consenso é possível, desde que haja uma maioria de processos sem-falha.

2.4 DETECTORES DE FALHAS

Algoritmos de consenso são imprescindíveis para o desensolvimento de aplicações distribuídas. Como mencionado anteriormente, o consenso não pode ser alcançado em sistemas assíncronos sujeitos a falhas por parada. De forma a resolver esse impasse, outro resultado importante foi publicado em 1996, onde é estudada a possibilidade de utilizar detectores de falha para resolver o problema do consenso em sistemas assíncronos (Chandra e Toueg, 1996). É apresentada a abstração de detectores de falha, onde cada processo tem acesso a um módulo local com informações sobre quais processos estão falhos e quais estão sem-falha. Para determinar se um processo está falho ou sem-falha geralmente são utilizados *timeouts*, se o processo não se comunicar com o detector de falhas dentro do período de *timeout* estabelecido, ele é considerado falho.

Os detectores de falhas apresentados são definidos através de duas propriedades básicas, a completude (*completeness*) e a precisão (*accuracy*). A completude garante que há um tempo após o qual todo processo falho será suspeito de ter falhado. A precisão garante que há um tempo após o qual todo processo correto não é suspeito de ter falhado. A completude pode ser ainda forte ou fraca, e tanto a completude forte quanto a fraca é classificada como *eventual*, palavra do inglês que indica que existe um momento após o qual algo vai acontecer, neste caso os processos falhos são suspeitos de estarem falhos. Por sua vez, a precisão pode ser forte, fraca, forte *eventual* ou fraca *eventual*, estas características são apresentadas a seguir.

- Completude forte *eventual*: a partir de algum momento, todos os processos falhos são suspeitos de estarem falhos por todos os processos corretos;
- Completude fraca *eventual*: a partir de algum momento, todos os processos falhos são suspeitos de estarem falhos por algum processo correto;
- Precisão forte: processos corretos nunca são suspeitos de estarem falhos por processos corretos;
- Precisão fraca: algum processo correto nunca é suspeito de estar falho por nenhum processo correto;
- Precisão forte *eventual*: a partir de algum momento, nenhum processo correto é suspeito de estar falho por outro processo correto;
- Precisão fraca *eventual*: a partir de algum momento, algum processo correto não é suspeito de estar falho por nenhum processo correto.

Através da combinação destas propriedades foram definidas oito classes diferentes de detectores de falhas. Cada classe é definida pela combinação de uma propriedade de completude e uma propriedade de precisão. A classe $\mathscr{P}$ representa o detector de falhas perfeito, que tem completude e precisão fortes, é a classe mais forte de todas. Por sua vez, a classe mais fraca de todas é a $\diamond\mathscr{W}$, que satisfaz completude fraca e precisão eventual fraca. Foi provado que mesmo a classe mais fraca de detectores de falhas é suficiente para que o consenso seja alcançado em um sistema assíncrono (Chandra e Toueg, 1996).

2.5 DIFUSÃO DE MELHOR ESFORÇO E DIFUSÃO CONFIÁVEL

A troca de mensagens entre os processos muitas vezes se dá através de um algoritmo de difusão (*broadcast*). Nesta situação, um processo fonte de uma mensagem *m* envia a mensagem para todos os processos presentes no sistema distribuído.

A difusão é baseada em duas primitivas *broadcast(m)* e *deliver(m)*, onde *m* é uma mensagem. A primitiva *broadcast(m)* é utilizada para enviar a mensagem *m* a todos os processos. Ao receber a mensagem, cada processo utiliza a primitiva *deliver(m)* para entregá-la para a aplicação. Cada mensagem *m* é identificada de forma única através do identificador do processo que emitiu a mensagem e o número de sequência local da mensagem. Dentre os vários algoritmos de difusão encontrados na literatura, dois são pertinentes neste trabalho: a difusão de melhor esforço (*best effort broadcast*) e a difusão confiável (*reliable broadcast*).

A difusão de melhor esforço garante a entrega de uma mensagem apenas uma única vez por cada processo correto. Contudo, se o processo fonte da mensagem falhar enquanto envia mensagem, a entrega não é garantida. Três propriedades são satisfeitas pela difusão de melhor esforço (Cachin et al., 2011), são elas:

- Não-duplicação: para toda mensagem *m* enviada por um processo, todos os processos corretos entregam *m* no máximo uma vez;
- Exclusão de mensagens espúrias: para toda mensagem *m* enviada por um processo, todos os processos corretos entregam *m* somente se *m* foi previamente enviada por algum outro processo;
- Validade: se a fonte da mensagem *m* não falhar, *m* é recebida e entregue por todos os processos.

Um algoritmo da difusão de melhor esforço sobre canais de comunicação confiáveis e um exemplo de execução são apresentados a seguir, no Algoritmo 1 e Figura 2.1, respectivamente. Na Figura 2.1 há quatro processos denominados P0, P1, P2 e P3. Neste exemplo, o processo 0 faz a difusão de uma mensagem, o envio desta mensagem para todos os processos é representado através de setas direcionadas. Cada processo, inclusive o próprio P0, ao receber a mensagem, entrega para a aplicação apenas uma única vez. Considerando o mesmo exemplo, caso o processo 0 falhasse após enviar a mensagem para P1, então os processos 2 e 3 não entregariam a mensagem.

Algoritmo 1: DIFUSÃO DE MELHOR ESFORÇO

```
1 broadcast(m): para todo processo no sistema envia m
2 receive(m): faz deliver(m)
```

Durante o desenvolvimento de aplicações tolerantes a falhas é frequentemente necessária a utilização da difusão confiável (*reliable broadcast*) (Cachin et al., 2011; Mullender, 1993). A difusão confiável, ao contrário da difusão de melhor esforço, garante que todos os processos corretos da rede entregam uma mensagem transmitida, mesmo que qualquer processo falhe durante a execução, inclusive o processo origem da mensagem (Cachin et al., 2011; Rodrigues et al., 2015).

As primitivas utilizadas por este algoritmo são as mesmas do anterior: *broadcast(m)* e *deliver(m)*. As propriedades que devem ser satisfeitas pela difusão confiável são (Rodrigues et al., 2015):

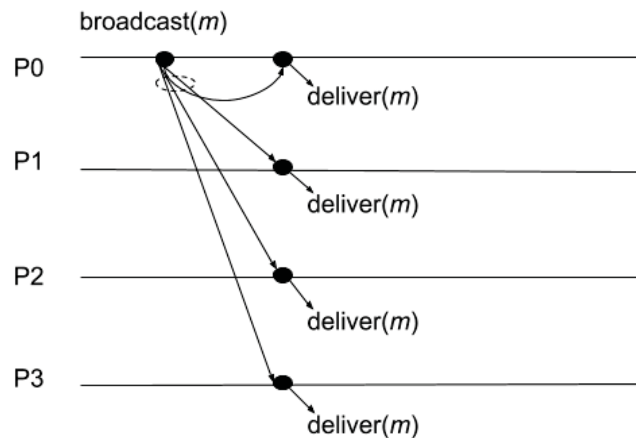

Figura 2.1: Exemplo de execução do algoritmo de difusão de melhor esforço (Cachin et al., 2011)

- Acordo: se um processo correto ou falho entregou a mensagem *m*, então todo processo correto entrega *m* em um tempo finito;
- Validade: se um processo faz *broadcast* da mensagem *m*, então todos os processos corretos irão, em algum momento, entregar *m*;
- Integridade: para toda mensagem *m* enviada por um processo, todos os processos corretos entregam *m*

O Algoritmo 2 apresenta a difusão confiável em um sistema com canais de comunicação perfeitos. Um exemplo da sua execução é mostrado na Figura 2.2, onde há os processos P0, P1, P2 e P3. No exemplo, o processo 0 está fazendo a difusão e cada processo que recebe a mensagem (P1, P2 e P3) repassa a mensagem recebida para todos os processos antes de entregar a mensagem. Desta forma, se um processo entrega a mensagem, então todos os processos sem-falha recebem e entregam a mensagem. Por exemplo, quando o processo 0 falha, o processo 1 já recebeu a mensagem e a repassou para os demais (linha 6 do Algoritmo 2), que fizeram o mesmo.

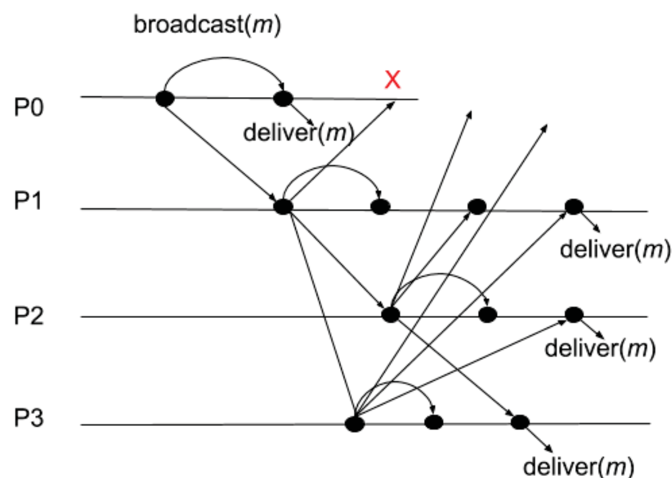

Figura 2.2: Exemplo de execução do algoritmo de difusão confiável.

Algoritmo 2: DIFUSÃO CONFIÁVEL

```
1 broadcast(m): para todo processo no sistema execute envia(m)
2 receive(m): para todo processo no sistema execute envia(m)
3               deliver(m)
```

Neste algoritmo, são transmitidas N^2 mensagens para completar a difusão.

3 PAXOS E RING-PAXOS

A replicação máquina de estados é uma técnica para mascarar falhas de processos. Para isso, é criada uma coleção de réplicas que recebem as mesmas sequências de operações, então passam pela mesma sequência de transições de estados, terminam no mesmo estado e produzem a mesma saída. Para garantir a disponibilidade, se assume que ao menos uma das réplicas nunca falha (Schneider, 1990).

O Paxos é um algoritmo de consenso tolerante a falhas e também um protocolo para replicação máquina de estados em um ambiente assíncrono sujeito a falhas por parada com recuperação (modelo *crash-recovery*) (Lamport, 2001; Van Renesse e Altinbuken, 2015). Este capítulo aborda o algoritmo Paxos e o Ring-Paxos, que utiliza a topologia em anel para melhorar o *throughput*.

3.1 PAXOS

O Paxos, comumente chamado de Paxos clássico, é um dos mais importantes algoritmos de consenso, e é utilizado em grandes sistemas, de grandes organizações (Chandra et al., 2007). Sendo muito utilizado no contexto de replicação, o Paxos é comumente executado em várias instâncias separadas (Lamport, 2001), e cada instância está associada a um valor decidido pelo consenso. Em cada instância, o Paxos é executado em duas fases distintas, descritas adiante.

No Paxos, os processos podem assumir os seguintes papéis: *proposers*, *acceptors* e *learners*. Os *proposers* são responsáveis por propor valores para o consenso, os *acceptors* por escolher um valor e os *learners* aprendem o valor decidido. Um processo pode assumir qualquer um desses papéis e também pode assumir múltiplos papéis simultaneamente. As fases do Paxos podem ser visualizadas no Algoritmo 3 e também são explicadas com mais detalhes a seguir.

Fase 1. (a) Um *proposer* seleciona uma proposta numerada *n* e envia um *prepare request* com número *n* para uma maioria de *acceptors*. Esse passo pode ser observado na Figura 3.1, onde o processo 0 (P0) é o *proposer* e envia o *prepare request* de número *n* para os *acceptors* A0, A1 e A2. Esse passo é também descrito como Tarefa 1 no Algoritmo 3.

(b) Se um *acceptor* recebe um *prepare request* com número *n* maior do que qualquer *prepare request* que já tenha respondido, então responde ao *prepare request* com um *prepare response* com a promessa de não aceitar nenhuma outra proposta com número menor do que *n* e, além disso, retorna o maior número de proposta que já aceitou, caso haja uma. Na Figura 3.1 os *acceptors* A0, A1 e A2 respondem ao *prepare request* recebido com uma mensagem *prepare response* (*prepareResp*) contendo o número da proposta *n*, o valor *v* da proposta de maior número que aceitou (caso ainda não tenha aceitado nenhuma, esse parâmetro é nulo) e a promessa de não aceitar nenhuma outra proposta com número menor que *n*. Esse passo pode ser visto no Algoritmo 3 como Tarefa 2.

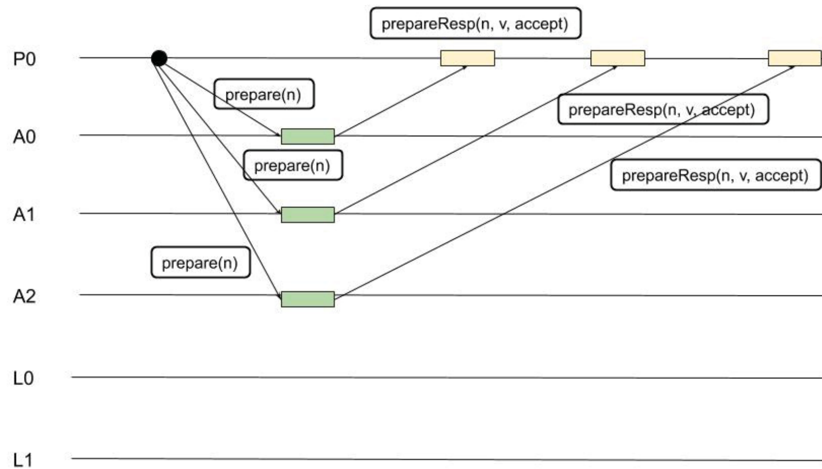

Figura 3.1: Exemplo de execução da fase 1 do algoritmo (de Camargo e Duarte Jr, 2017).

Fase 2. (a) Se o *proposer* recebe respostas ao seu *prepare request* de número n de uma maioria de *acceptors*, então ele envia um *accept request* para cada um desses *acceptors* para a proposta com número n e o valor v, onde v é o valor da proposta de maior número dentre as *prepare responses*. Na situação em que as respostas não informaram propostas já aceitas, o *proposer* envia o *accept request* com o seu próprio valor v. Essa fase pode ser visualizada na Tarefa 3 (Algoritmo 3) e na Figura 3.2, onde o *proposer* P0 envia o *accept request* para os *acceptors* A0, A1 e A2 contendo o número n e o valor v, na fase anterior esses *acceptors* enviaram respostas ao *prepare request* de P0 com número n.

(b) Se um *acceptor* recebe um *accept request* de uma proposta numerada n, ele aceita a proposta a menos que já tenha respondido a um *prepare request* com número ainda maior que n. Na Figura 3.2, ao aceitar a proposta, os *acceptors* A0, A1 e A2 respondem ao *accept request* com *accept responses* informando o número de proposta n e o valor v que foi aceito. Em seguida, os *acceptors* enviam o valor aceito para os *learners* (L0 e L1). Esse passo é descrito como Tarefa 4 no Algoritmo 3.

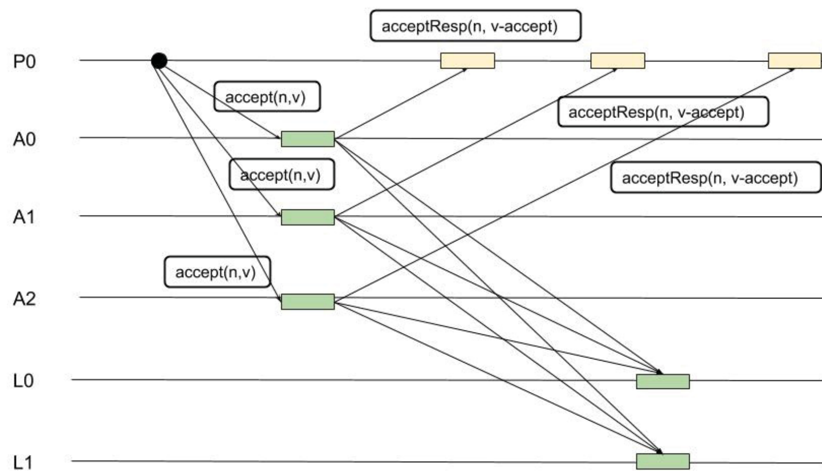

Figura 3.2: Exemplo de execução da fase 2 do algoritmo (de Camargo e Duarte Jr, 2017).

Algoritmo 3: PAXOS

```
1  Tarefa 1 (coordenador):
2  Quando recebe valor V de um proposer P
3      enumera a proposta n
4      envia prepare-request(n) para todo acceptor pertencente ao quorum
5  Tarefa 2 (acceptor):
6  Quando recebe prepare-request(n) do coordenador
7      se n > maior-numero-recebido então
8      |   maior-numero-recebido ← n
9      fim
10     envia prepare-response(n, v, accept) ao coordenador
11 Tarefa 3 (coordenador):
12     Quando recebe prepare-response (n, v, accept) de uma maioria de acceptors
13     seleciona valor V da proposta de maior numero dentre as prepare-responses
14     para cada acceptor da maioria envia accept-request(n, V)
15 Tarefa 4 (acceptor):
16     Quando recebe accept-request(n, V) do coordenador
17     se nao respondeu a um prepare-request com numero maior que n então
18     |   aceita o valor V
19     fim
20     envia accept-response(n, V, accept) ao coordenador
```

É fácil acontecer uma situação onde dois ou mais *proposers* fiquem enviando propostas com números crescentes, de forma que nenhum deles nunca seja escolhido e o consenso nunca termine. Assim, para garantir o progresso, um dos *proposers* pode ser classificado como coordenador. Os demais *proposers* enviam seus valores a ser propostos para o processo coordenador, que então executa a primeira e a segunda fases do algoritmo. Em caso de falha do coordenador, outro é escolhido. A consistência é garantida mesmo com múltiplos *proposers* concorrentes, e a terminação também é garantida na presença de um único coordenador (Lamport, 2001).

O coordenador é o único que executa as fases do Paxos, enviando e recebendo mensagens para os *acceptors* e por isso, quanto maior o sistema, mais sobrecarregado o coordenador fica. Algumas alternativas são propostas a fim de reduzir a carga do coordenador, uma delas é o Ring Paxos, apresentado a seguir.

3.2 RING-PAXOS

O Ring-Paxos é proposto no contexto de *atomic broadcast* (ou difusão atômica), que garante a entrega das mensagens na mesma ordem por todos os processos. A difusão atômica pode ser implementada através da execução consecutiva de várias instâncias de consenso. O Ring-Paxos é uma versão do Paxos que organiza os processos *acceptors* e *learners* em um anel lógico unidirecional e é uma versão otimizada para execução em redes locais (Jalili Marandi et al., 2017). O algoritmo tira proveito da topologia em anel principalmente ao executar a Fase 2 do Paxos, pois se forma um *pipeline* de execução quando várias instâncias são executadas. Isso porque, quando um *acceptor* recebe uma mensagem da Fase 2, ele encaminha essa mensagem juntamente com a sua resposta para o próximo processo no anel, e isso se repete até que a mensagem chegue ao processo coordenador. São propostos dois protocolos: o M-Ring Paxos e o

U-Ring Paxos. O primeiro utiliza comunicação *multicast* e *unicast*, enquanto que o segundo utiliza apenas *unicast*. Os protocolos são apresentados a seguir.

No M-Ring Paxos, a primeira fase do consenso pode ser executada antecipadamente para múltiplas instâncias. A primeira fase é exatamente a mesma do Paxos. Na segunda fase, os *acceptors* são dispostos em um anel lógico unidirecional e as mensagens são propagadas entre os *acceptors* através do anel. A execução da Fase 2 do M-Ring Paxos pode ser vista na Figura 3.3 e também está descrita como Tarefa 3 e Tarefa 4 no Algoritmo 4. O coordenador inicia a Fase 2 disseminando as mensagens *accept-request* através de um *multicast*. Ao receber um *accept-request*, um *acceptor* verifica se pode aceitar o valor recebido. Em caso positivo, atualiza seus valores de rodada e valor aceito. A mensagem percorre o anel da seguinte forma. O primeiro *acceptor* no anel responde ao *accept-request* com uma mensagem *accept response* e encaminha a mensagem para o seu sucessor no anel através de um *unicast*. O próximo *acceptor* recebe o *accept-response* e verifica se recebeu previamente um *accept-request* referente àquele valor (enviado através do *multicast)*. O *acceptor* só pode aceitar um valor se tiver previamente recebido o valor em uma mensagem *accept-request*. Quando a mensagem atinge o último *acceptor* do anel, ou seja, o coordenador, este envia uma mensagem informando a decisão através de um *multicast*.

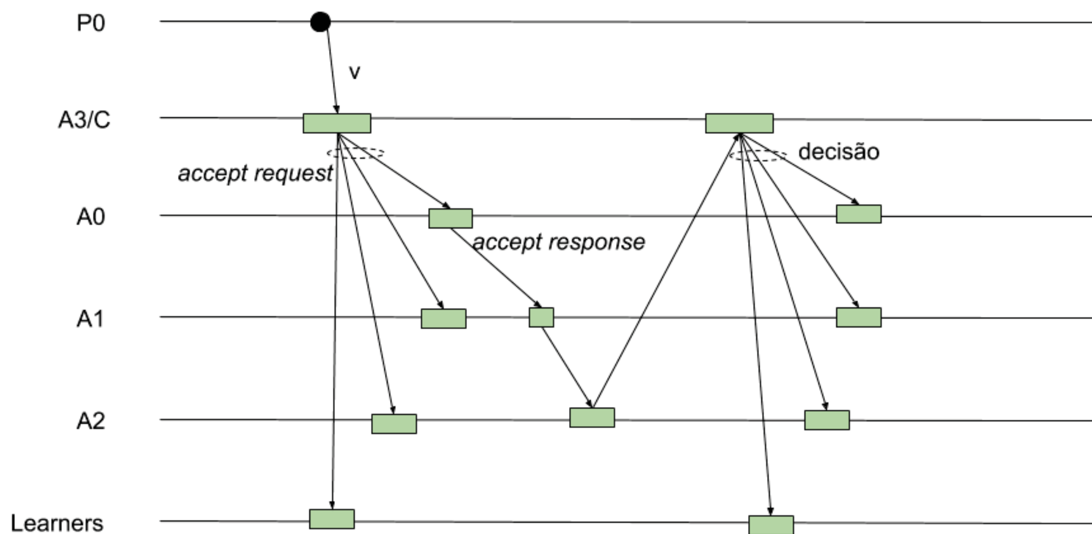

Figura 3.3: Execução da Fase 2 no M-Ring Paxos.

Algoritmo 4: M-RING PAXOS

```
1  Tarefa 1 e Tarefa 2: iguais ao Paxos
2  Tarefa 3 (coordenador):
3  Quando recebe de uma maioria de acceptors o prepare-response contendo número
     de proposta p
4     Envia accept-request(p, V) para os acceptors e learners
5  Tarefa 4 (acceptor):
6  Quando recebe accept-request(p, V) do coordenador
7     se processo pode aceitar o valor V então Atualiza seus valores
8     se processo é o primeiro no anel então
9     |   Envia accept-response(p, V, accept) para o sucessor no anel
10    fim
11 Tarefa 5 (coordenador e acceptor):
12 Quando recebe accept-response
13    se processo não é o último no anel então
14    |   encaminha accept-response(p, V, accept) para o acceptor sucessor no anel
15    senão
16    |   envia decisão para todos
17    fim
```

O U-Ring Paxos, ao contrário do M-Ring, prevê as Fases 1 e 2 propagadas em um anel lógico direcional apenas através de mensagens *unicast*. O U-Ring Paxos coloca os *proposers*, *learners* e o quorum majoritário de *acceptors* em um anel lógico direcionado. O pipeline de todos os processos em um anel é uma alternativa ao *multicast* com o objetivo de atingir alta taxa de transferência. Para reduzir a latência o coordenador pode ser posicionado no início do anel, porém não há restrições quanto a posição dos processos no anel, independente dos seus papéis. A execução pode ser vista na Figura 3.4 e no Algoritmo 5. Uma vez que um *proposer* propõe um valor, o valor é encaminhado ao longo do anel até chegar ao coordenador, que prosseguirá com a Fase 1, como no Paxos. Quando o coordenador recebe mensagens da Fase 1b de um quorum, o coordenador verifica qual valor pode ser proposto e atribui um identificador exclusivo ao valor a ser proposto, da mesma forma que no M-Ring Paxos. O coordenador envia as mensagens da Fase 2a e 2b (*accept request* e *accept response*) para o seu sucessor no anel. Da mesma forma que Paxos e M-Ring Paxos, o coordenador em U-Ring Paxos pode executar a Fase 1 antes que um valor seja proposto, reduzindo a latência do protocolo.

Ao receber uma mensagem da Fase 2, um *acceptor* verifica se pode votar no valor proposto. Nesse caso, ele atualiza suas variáveis, como no Paxos. Se o *acceptor* não preceder o último *acceptor* no anel, ele envia a mensagem da Fase 2 ao seu sucessor. Diferentemente do M-Ring, onde o coordenador é quem verifica se uma decisão foi tomada na instância, no U-Ring Paxos, isso é delegado ao último *acceptor* no anel. Após a decisão, o último *acceptor* envia a decisão, possivelmente junto com o valor escolhido, ao seu sucessor no anel. O valor escolhido precisa então percorrer o anel a partir do início até chegar ao predecessor do *acceptor* que emitiu a decisão.

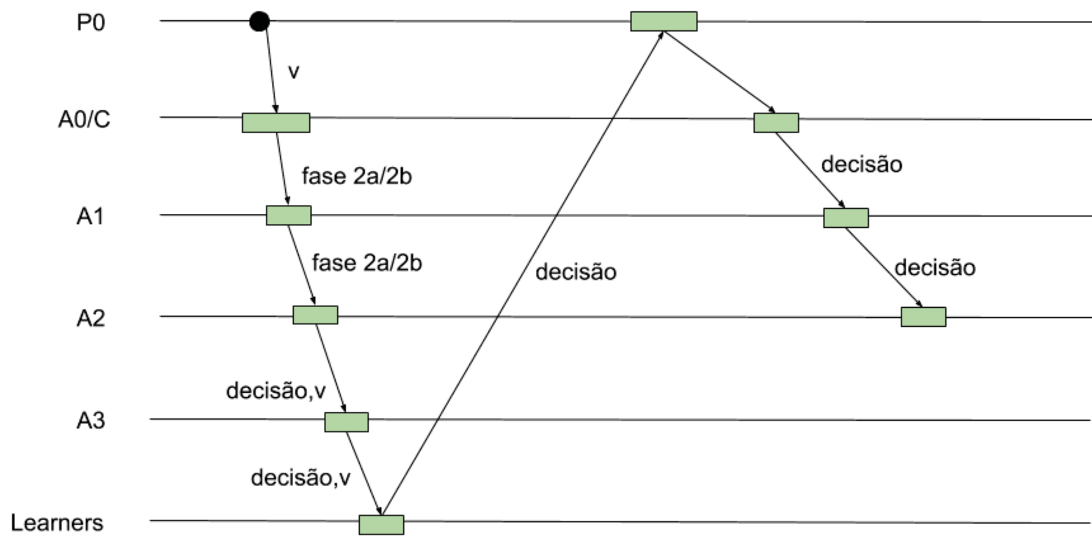

Figura 3.4: Execução da Fase 2 no U-Ring Paxos

Algoritmo 5: U-RING PAXOS

```
1  Tarefa 1 (todos os processos):
2  Quando recebe valor de um predecessor no anel um V proposto por um proposer P
3  se processo é coordenador então
4  |   Atualiza numero de rodada
5  |   para todo acceptor em quorum envia prepare-request
6  senão
7  |   envia V para o sucessor no anel
8  fim
9  Tarefa 2 (acceptor):
10 Quando recebe prepare-request do coordenador
11     Atualiza valores de rodada
12     Envia prepare-response ao coordenador
13 Tarefa 3 (coordenador):
14 Quando recebe prepare-response de um acceptor
15     se rodada recebida na mensagem for atual então
16 |       Atualiza valores
17     fim
18     Envia prepare-response/accept-response para o próximo acceptor no anel
19 Tarefa 4 (acceptor):
20 Quando recebe mensagem prepare-response/accept-response
21     se processo pode votar então
22 |       Atualiza valores
23     fim
24     se processo é o último acceptor no anel então
25 |       Envia decisão
26     senão
27 |       Encaminha mensagem para o sucessor no anel
28     fim
29 Tarefa 5 (todos os processos):
30 Quando recebe decisão
31     se processo não é predecessor do último acceptor no anel então
32 |       Encaminha decisão
33     fim
34     se processo é predecessor do proposer então
35 |       envia decisão contendo o valor para o sucessor no anel
36     senão
37 |       envia decisão sem valor para o sucessor no anel
38     fim
```

Um coordenador que suspeita da falha de um ou mais *acceptors* simplesmente tenta contactar todos os *acceptors* para reunir um quórum. Essa solução reduz o *troughput*, mas permite o progresso, apesar das falhas. Mensagens perdidas são resolvidas com retransmissão. Se o coordenador não recebe respostas das mensagens *prepare request* ou *accept request*, reenvia as mensagens possivelmente com um número de rodada maior. *Eventually*, o coordenador recebe uma resposta ou suspeita da falha de um processo, a suspeita pode ou não estar correta. Quando o coordenador suspeita da falha de um processo, estabelece um novo anel, excluindo o processo suspeito, e executa novamente a Fase 1.

O processo coordenador pode falhar, e neste caso os demais processos detectam a falha do coordenador e selecionam um novo. Assim como no Paxos, o Ring Paxos garante a segurança (*safety*), mesmo quando vários coordenadores executam ao mesmo tempo, embora não possa garantir a progressão (*liveness*). Contudo, após o GST (*Global Stabilization Time*), *eventually*, um único coordenador correto é selecionado.

Além do Ring Paxos, existem diversas outras variações do Paxos, como o Egalitarian Paxos (Moraru et al., 2013) e o Fast Paxos (Lamport, 2006).

4 VCUBE: UMA VISÃO GERAL

Este Capítulo apresenta o VCube e um algoritmo de difusão de melhor esfoço definido sobre o VCube. O algoritmo de difusão aproveita as propriedades do VCube para diminuir o número de mensagens enviadas. Também são apresentados os algoritmos da teoria de diagnóstico *Adaptive-DSD* e Hi-ADSD que inspiraram o VCube e sua evolução em um detector de falhas.

4.1 OS ALGORITMOS ADAPTIVE-DSD E HI-ADSD

Um dos algoritmos que proveêm o serviço de detecção de falhas é o *Adaptive-DSD* (*Adaptive Distributed System-Level Diagnosis*) (Bianchini e Buskens, 1991). neste algoritmo, a detecção de falhas é feita baseada em testes entre os processos. Quando um processo responde corretamente a um teste dentro de um determinado tempo, o processo é considerado sem-falha, Em caso contrário é considerado falho. O algoritmo prevê falhas e recuperações de processos, mas o modelo assume que não há falhas de enlaces e a topologia deve ser totalmente conectada. O número permitido de processos falhos é limitado a $N - 1$, isto é, ainda que haja apenas um processo sem-falha, este é capaz de determinar o estado de todos os demais processos do sistema. O algoritmo executa em rodadas de testes, que consistem no período de tempo necessário para que todos os processos sem-falha executem os testes que lhe foram atribuídos.

A estratégia de testes no *Adaptive-DSD* consiste em assinalar aos processos identificadores sequenciais e cada processo testar o processo seguinte, ou seja, um processo com identificador 1 testa o processo com identificador 2, que testa o próximo (identificador 3), e assim por diante. No caso de um processo identificar um processo falho, o processo testa o seguinte do seguinte, até que encontre um processo sem-falha, ou até que teste todos os processos no sistema. Um exemplo dos testes é mostrado na Figura 4.1; nela, os processos são numerados de 0 a 5, e os processos falhos são circulados em vermelho, enquanto os sem-falha são circulados em preto. Os testes são representados através de arcos direcionados. No *Adaptive-DSD*, uma rodada de testes é finalizada quando todos os processos sem-falha realizaram teste em outro processo sem-falha ou quando um processo testa todos os outros como falhos. Um testador que testa um processo sem-falha obtém informações sobre outros processos do sistema a partir do processo testado.

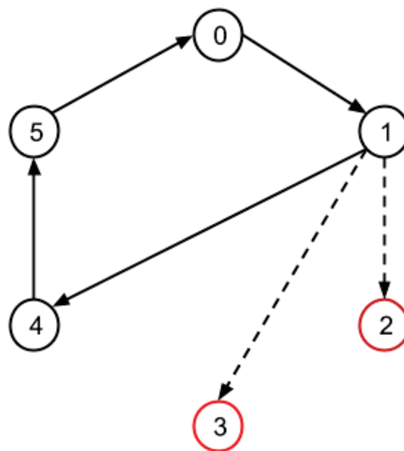

Figura 4.1: Exemplo de execução do algoritmo *Adaptive-DSD* com processos falhos.

Outro algoritmo, que além de distribuído e adaptativo, é também hierárquico, é o *Hierarchical Adaptive Distributed System-Level Diagnosis* (Hi-ADSD). A hierarquia é baseada em um hipercubo e quando todos os processos estão sem-falha, o grafo de testes forma um hipercubo completo. O hipercubo é escalável por definição, sua topologia apresenta características importantes: simetria, diâmetro logarítmico e boas propriedades para tolerância a falhas. No hipercubo, os processos são identificados unicamente. Quando há apenas um *bit* de diferença nas representações binárias dos identificadores de quaisquer dois processos i e j, esses processos são vizinhos. Um hipercubo com 2^d elementos tem dimensão d, e um caminho entre quaisquer dois processos tem no máximo d arestas. Um hipercubo de dimensão $d = 3$ é mostrado na Figura 4.2, nele há 8 processos.

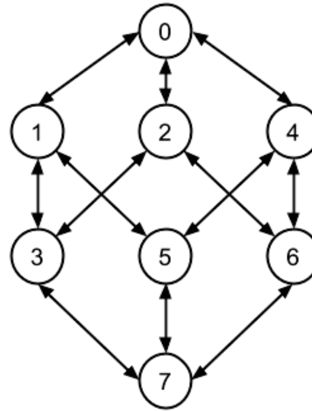

Figura 4.2: Hipercubo com dimensão $d = 3$.

O Hi-ADSD organiza os processos do sistema em *clusters* (conjuntos de processos) para a execução dos testes, o tamanho desses *clusters* é sempre uma potência de dois e os *clusters* são progressivamente maiores conforme a quantidade de processos no sistema. Os *clusters* são organizados e mantidos na estrutura do hipercubo.

Os testes executados pelos processos são definidos através de uma lista ordenada que é retornada pela função $C_{i,s}$, onde i é o processo que executa os testes e 2^{s-1} é o tamanho do *cluster* no qual o teste será feito. A função é apresentada a seguir.

$$C_{i,s} = i \oplus 2^{s-1} \mid\mid C_{i \oplus 2^{s-1},k} \mid k = 1, ..., s-1$$

Na Tabela 4.1 estão as listas de processos retornada pela função $C_{i,s}$ para um sistema com 8 processos. Na tabela, cada coluna representa um processo, que vai de 0 a $N - 1$. A quantidade de linhas é $\log_2 N$ e representa o *cluster* no instante s. Os primeiros processos sem-falha de cada *cluster* serão os processos testadores de um determinado processo i. Por exemplo, os processos testadores do processo 2, são: o processo 3 ($C_{2,1}$), o processo 0 ($C_{2,2}$) e o processo 6 ($C_{2,3}$).

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

Tabela 4.1: Processos organizados de acordo com a função $C_{i,s}$ para um sistema com 8 processos no Hi-ADSD.

No Hi-ADSD, quando um processo testador testa um processo que está sem-falha, ele obtém informações sobre os demais processos pertencentes ao *cluster* testado. Contudo, quando

o testador testa um processo que está falho, continua testando os outros processos no *cluster* até encontrar um processo sem-falha, ou até testar todos os processos do *cluster*. Concretamente, o algoritmo consiste em o processo testador executar testes sequencialmente em cada *cluster* até que encontre um processo sem-falha.

Embora a latência do algoritmo seja logarítmica ($(\log_2 N)^2$), assumir essa estratégia pode gerar um número quadrático de testes entre os processos. Imagine um sistema com N processos numerados de 0 a 7. Neste sistema os processos 0, 1, 2 e 3 estão sem-falhas e os processos 4, 5, 6 e 7 estão falhos. Ou seja, $N/2$ processos estão sem-falha e $N/2$ processos estão falhos, formando dois *clusters* de mesmo tamanho, sendo um *cluster* completo com processos sem-falha e o outro completo com processos falhos. Suponha que todos os processos do *cluster* sem-falha estão executando os testes no *cluster* falho. Neste exemplo cada processo do *cluster* sem-falha irá executar testes em todos os processos do *cluster* falho. Isso resulta em $N/2$ testes por cada processo sem-falha, ou seja, $N/2 * N/2 = \mathcal{O}(N^2)$ testes em uma rodada de testes. Esta mesma complexidade pode ser obtida com um algoritmo de força bruta. Neste contexto surge o VCube, um topologia virtual semelhante ao Hi-ADSD que mantém propriedades logarítmicas mesmo no pior caso.

4.2 O ALGORITMO DO VCUBE

O VCube, ou *Virtual Hypercube*, (Duarte Jr et al., 2014) é uma topologia virtual para sistemas distribuídos que provê o serviço de detecção de falhas. O VCube organiza os processos do sistema em um hipercubo quando todos os processos estão sem-falha (Rodrigues et al., 2015). A topologia é criada e mantida com base nas informações de monitoramento obtidas através de testes executados entre os processos. O VCube foi desenvolvido a partir do Hi-ADSD, apresentado na seção anterior. Diversos conceitos do Hi-ADSD são reapresentados nesta seção por completude, pois o algoritmo efetivamente usado na contribuição da dissertação é o VCube.

No algoritmo do VCube, os processos do sistema são agrupados em *clusters* para a realização de testes. Os *clusters* são conjuntos de processos com tamanhos que são sempre potências de 2. A Figura 4.3 mostra um sistema com $N = 8$ processos e a organização dos *clusters* para esse sistema, neste exemplo é considerado que todos os processos estão sem-falha.

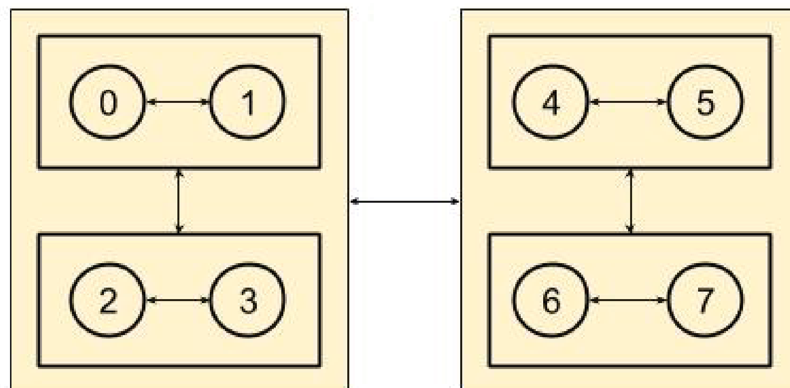

Figura 4.3: Organização dos *clusters* para um sistema com 8 processos.

Pode-se observar pela Figura 4.3 que cada processo sozinho compõe um *cluster* de tamanho 1 e que tem aresta com o processo adjacente, que tem apenas o *bit* menos significativo diferente na representação binária dos seus identificadores, por exemplo os processos 4 (100) e 5 (101). Estes dois processos também formam, juntos, um *cluster* de tamanho 2 que tem arestas

com outro *cluster* de igual tamanho. Por exemplo, o processo 4 (100) tem uma aresta com o processo 6 (110) e o processo 5 (101) tem uma aresta com o processo 7 (111). Esses 4 processos juntos formam um outro *cluster* de tamanho 4 que tem arestas com outro *cluster*, onde o processo 4 (100) tem aresta com o processo 0 (000), o processo 5 (101) com o processo 1 (001), o processo 6 (110) com o processo 2 (010) e o processo 7 (111) com o processo 3 (011). Na Figura 4.3 cada processo é representado através do círculo numerado, e os *clusters* que compõem dois ou mais processos são representados através de retângulos, e suas conexões (arestas) são representadas através dos arcos entre os *clusters*. Cada processo pertence a $\log N$ *clusters* em um sistema com N processos.

Em intervalos de tempo periódicos (por exemplo a cada 30 segundos) os processos executam testes em todos os seus $\log N$ *clusters*. Um processo mantém informações sobre o estado de todos os outros processos do sistema. Desta forma o VCube oferece um serviço de detecção de falhas. As informações de estado dos processos são armazenadas na forma de contadores. Cada processo mantém um vetor local $state[0...N-1]$ com as informações obtidas dos demais processos. Um processo i após obter a informação de um processo testado j como sem-falha, atualiza o valor no seu vetor de contadores apenas quando o valor correspondente obtido for maior do que o atual, ou seja, quando um novo evento tiver acontecido sobre aquele processo.

Um processo testador pode obter informações sobre um determinado processo através de vários outros processos e inicialmente todos os processos são considerados sem-falha, o contador correspondente é 0. Quando um evento é detectado, ou seja, um processo sem-falha se torna falho ou um processo falho se torna sem-falha, então o contador correspondente àquele processo é incrementado. Desta forma, quando o valor for par irá corresponder a um processo sem-falha e quando for ímpar corresponderá a um processo falho.

Antes de um processo i executar um teste em um processo $j \in C_{i,s}$, ele verifica se ele é o primeiro processo sem-falha na lista ordenada de processos que podem testar o processo j na rodada s. Se for, então executa o teste. Após a execução do teste, caso o processo j esteja sem-falha, então o processo i obtém novas informações de monitoramento a partir do processo j sobre os demais processos do sistema. Ao obter a nova informação, o contador correspondente é incrementado. Quando os testes são realizados por todos os processos do sistema, uma rodada de testes é finalizada. No algoritmo do VCube é garantido que serão executados no máximo $N \log N$ testes em $\log N$ rodadas de testes, onde N é a quantidade de processos do sistema.

A Figura 4.4 apresenta o mesmo sistema com 8 processos anteriormente apresentado (Figura 4.3), desta vez considerando os processos um (1) e sete (7) falhos. Estes processos estão marcados com um X vermelho na imagem. Quando um processo está falho, ele é testado, mas não responde corretamente dentro do tempo pré-determinado, por isso é considerado falho pelo processo que o testou. Além disso, o processo falho não realiza testes. Na Figura, os testes são representados por setas direcionadas entre os vértices, que representam cada processo. O processo testador, conforme descrito anteriormente, gera a lista ordenada de processos testadores de um determinado nodo e verifica se ele próprio é o primeiro sem-falha de um *cluster* de um determinado processo. Se estiver, testa o nodo. Quando há processos falhos, um processo testador pode executar mais testes, como pode ser visto na Figura 4.4, onde o processo 0 executa também testes no processo 3, além dos testes que faz quando os processos estão todos sem-falha.

A fim de acelerar a disseminação de informações sobre novos eventos quando um processo i testa um processo j recebe apenas as novas informações, desde o último teste realizado do processo j pelo processo i. Utilizando esta estratégia, a latência média do algoritmo é melhorada, pois um processo testador pode obter novas informações de diagnóstico sobre todos os processos do sistema através de qualquer processo testado. O VCube mantém a mesma latência

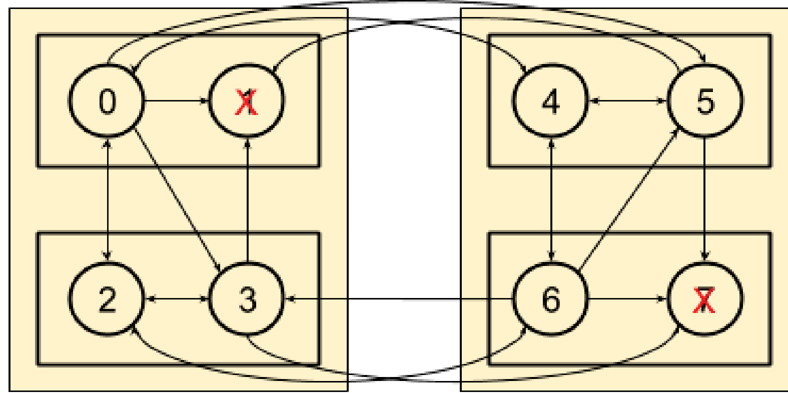

Figura 4.4: Organização dos *clusters* para um sistema com 8 processos e processos 1 e 7 falhos.

do algoritmo Hi-ADSD ($(\log_2 N)^2$), mas o número de testes realizados reduz significativamente, executando no máximo $N \log_2 N$ testes no pior caso.

4.2.1 Difusão Sobre o VCube

Em (Rodrigues et al., 2014) foram propostos dois algoritmos de difusão sobre o VCube, sendo um algoritmo de difusão de melhor esforço e um algoritmo de difusão confável. Como neste trabalho é utilizada a difusão de melhor esforço, este algoritmo é explicado a seguir. Nele, se a origem da mensagem falha, a entrega não é garantida pelos demais processos. O processo fonte gera uma mensagem *m*, que contém dois parâmetros, o identificador da origem (processo que gerou a mensagem) e o *timestamp*, que é um contador local de mensagens transmitidas pelo processo. Juntos os dois parâmetros permitem a identificação única de cada mensagem gerada. O processo fonte *i* inicia a difusão invocando o procedimento `broadcast(m)`, que pode ser observado no Algoritmo 6. O algoritmo aguarda que uma difusão feita anteriormente termine antes de iniciar uma nova. Primeiramente o algoritmo faz uma entrega local da mensagem (linha 4) e em seguida envia a *m* para todos os vizinhos sem-falha do VCube. Para cada mensagem enviada, um *ack* é incluído na lista de *acks* pendentes $ackSet_i$.

Quando um processo *i* recebe uma mensagem de um processo *j* (linha 14), ele verifica se a origem de *m* e se *j* estão falhos. Caso estejam, encerra o método (linha 15). Em caso contrário, o processo determina se a mensagem é nova, comparando os *timestamps* da última mensagem recebida. Caso *m* seja uma nova mensagem, atualiza a última mensagem recebida e entrega para a aplicação através da primitiva *deliver(m)*. Em seguida, o processo retransmite a mensagem para outros processos de forma descrita com mais detalhes posteriormente. Por sua vez, quando um processo *i* recebe uma mensagem que seja do tipo *ack* de um processo *j* (procedimento `receiveAck`, linha 26), o conjunto $ackSet_i$ é atualizado e, caso não existam mais *acks* pendentes para a mensagem *m*, envia um *ack* para o processo *x* do qual *i* recebeu a mensagem anteriormente. No entanto, se $x = i$, a mensagem *ack* ou alcançou a fonte de *m* ou o processo que retransmitiu a mensagem após a falha do processo fonte. Neste caso, a mensagem *ack* não precisa mais ser propagada.

A detecção de um processo falho *j* é tratada no procedimento `falha(j)`. Primeiramente a lista de processos corretos é atualizada (linha 32), em seguida remove os *acks* pendentes para mensagens que foram retransmitidas por *j* e aqueles em que a mensagem *m* foi originada em j. A terceira ação é reencaminhar para um novo vizinho *k* no *cluster* de *j* as mensagens que anteriormente foram transmitidas para j.

Algoritmo 6: DIFUSÃO DE MELHOR ESFORÇO SOBRE O VCUBE PARA PROCESSO i

```
1  BROADCAST(m)
2  espere ate terminar o broadcast anterior, caso haja
3      atualiza última mensagem recebida
4      faz deliver(m)
5  para todo j vizinho ao cluster log n execute
6      acrescenta a mensagem ao ackSet_i
7      envia m para j
8  VERIFICAACKS(j, m)
9  se não existem acks pendentes para a mensagem m recebida de j então
10 |   se fonte(m) e j estão sem-falha então
11 |   |   envia ack para j
12 |   fim
13 fim
14 RECEIVE(m)
15 se fonte(m) e j estao falhos então return
16 se m for uma nova mensagem para i então
17 |   atualiza última mensagem recebida
18 |   faz deliver(m)
19 fim
20 para todo k vizinho de cluster(j) − 1 execute
21     se j ainda nao enviou m para k então
22     |   acrescenta mensagem ao ackSet_i
23     |   envia m para k
24     fim
25 VERIFICAACKS(j, m)
26 RECEIVEACK(ACK, m)
27 remove do conjunto ackSet_i a mensagem m recebida por j de um processo x
28 se x ≠ i então
29 |   VERIFICAACKS(k, m)
30 fim
31 FALHA(j)
32 retire j do conjunto de processos sem-falha
33 selecione k, tal que k é o primeiro processo sem-falha no cluster de j
34 para todas as mensagens pendentes em ackSet_i execute
35     se fonte(m) e x estão falhos, onde x é o processo que enviou uma mensagem
         então
36     |   remove ACKs pendentes para mensagens enviadas por j e de qualquer
     |     processo para o qual foi enviada m, sendo fonte(m) = j
37     senão se mensagem foi transmitida para j por um outro processo p então
38     |   se não há registro de envio de p para k em ackSet_i então
39     |   |   acrescenta mensagem ao conjunto ackSet_i
40     |   |   retransmite mensagem para novo vizinho k
41     |   fim
42     |   VERIFICAACKS(x, m)
43     fim
```

As Figuras 4.5 e 4.6 mostram o caminho percorrido por uma mensagem m gerada e enviada pelo processo 0 em um sistema com 8 processos onde todos estão sem-falha. O processo 0 verifica de acordo com sua função $C_{i,s}$ para quais processos deve enviar a mensagem. Envia para o primeiro vizinho sem-falha do primeiro *cluster* ($C_{0,1}$), o processo 1, depois para o primeiro vizinho sem-falha do segundo *cluster* ($C_{0,2}$) e assim por diante. O processo que recebe a mensagem repassa para os seus vizinhos nos seus *clusters*. O número de mensagens transmitidas depende do *cluster* em que o processo está, como descrito a seguir. Quando todos os processos corretos entregam a mensagem, o *broadcast* termina.

Ao disparar a mensagem para o *broadcast*, uma árvore é construída através do VCube, sendo o processo fonte a raiz da árvore. Cada processo, ao receber uma mensagem, verifica a qual *cluster* ele pertence para o processo de quem recebeu a mensagem. O identificador do *cluster*-1 indica para quantos outros processos a mensagem deve ser enviada por aquele processo. Por exemplo, se ele pertence ao *cluster* 3 do processo que enviou a mensagem, então precisa enviar a mensagem para outros dois *clusters*, os primeiros processos sem-falha dos seus *clusters* 1 e 2, respectivamente. Se ele pertence ao *cluster* 1 do processo que enviou a mensagem, então é um nodo folha da árvore e não é preciso repassar a mensagem para ninguém.

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

Figura 4.5: Mensagens enviadas pelo processo 0.

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

Figura 4.6: Mensagens enviadas pelos demais processos.

Um exemplo da difusão em um sistema com processos falhos é mostrado na Figura 4.7. Neste exemplo, os processos 2 e 4 estão falhos e estão representados por círculos vermelhos.

Os processos falhos não são inclusos nas listas ordenadas retornadas pela função $C_{i,s}$ e estão representados com um X vermelho na tabela apresentada. O processo 0 inicia a difusão e envia a mensagem para o processo 1 seguindo a mesma lógica do exemplo anterior. Contudo, o processo fonte sabe que o processo 2 está falho, então verifica nas listas ordenadas para qual outro processo ele envia mensagem no *cluster* s = 2. Envia então para o processo 3, que está sem-falha, e que não envia mensagem para nenhum outro processo. Da mesma forma, o processo não envia mensagem para o processo 4, que está falho, e envia para o próximo processo da sua lista de *cluster* 3, o processo 5, que envia a mensagem para o processo 7, o qual por sua vez envia para o processo 6, e assim a difusão é terminada.

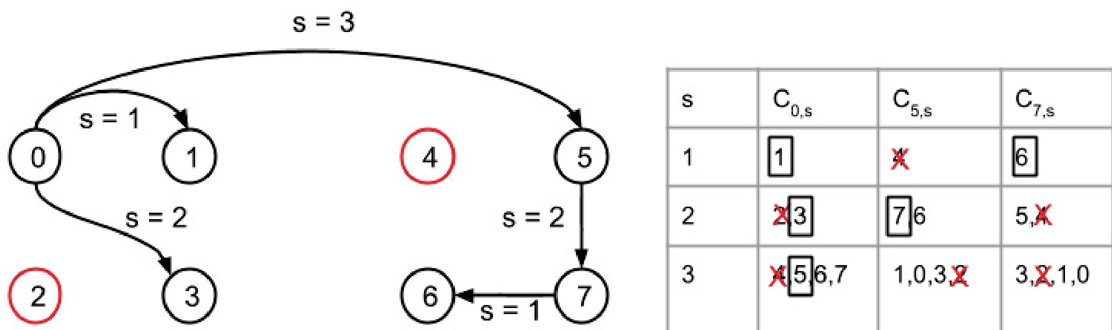

Figura 4.7: Mensagens enviadas quando os processos dois e quatro estão falhos.

Além de ser escalável em termos do número de mensagens que utiliza, o algoritmo também é considerado autonômico, pois há uma adaptação a falhas. As mensagens são transmitidas para o primeiro processo sem-falha de cada *cluster*, independente de quem seja. Para cada mensagem emitida através da difusão, o total de mensagens trocadas entre os processos em uma execução sem-falhas do algoritmo 6 é $2 * (n - 1)$. Se um processo j que recebeu a mensagem de um processo i falha antes de confirmar o recebimento, o total de mensagens extra depende da quantidade de processos no *cluster* de j.

5 UMA INSTÂNCIA DO PAXOS SOBRE O VCUBE

A seguir é apresentado o algoritmo que implementa uma instância do Paxos tirando proveito da topologia virtual hierárquica do VCube. O modelo de sistema é parcialmente síncrono com GST (*Global Stabilization Time*), isto é, após um período de instabilidade o sistema passa a respeitar limites de tempo de transmissão de mensagens e processamento. Os canais de comunicação são confiáveis.

O algoritmo tem as duas fases do Paxos. A Fase 1 é descrita abaixo. Nessa fase, o *proposer* dissemina o *prepare-request* no VCube com o objetivo de obter uma maioria de *prepare-responses*. Após receber respostas de uma maioria de *acceptors*, o *proposer* utiliza um algoritmo de difusão de melhor esforço baseado no algoritmo apresentado no Capítulo 4 para enviar para os *acceptors* o *accept-request* com o maior número de rodada recebido na Fase 1 e o valor correspondente.

A Fase 1 do algoritmo proposto tira proveito da organização dos processos em *clusters* para acelerar sua execução. O *proposer* é também *acceptor*, desta maneira são necessários $n/2$ *prepare-responses* para obter uma maioria. O *proposer* envia o *prepare-request* apenas para o maior *cluster*, de índice $\log n$ que consiste de $n/2$ nodos. Se nenhum nodo deste *cluster* estiver falho ou incorretamente suspeito, o *proposer* já terá a maioria. Caso contrário, segue para o *cluster* de índice $\log n - 1$, e assim por diante, até conseguir $n/2$ *prepare-responses*. Neste ponto a Fase 1 está completa.

O algoritmo `VCubeProposerFase1` executado pelo *proposer* é mostrado como Algoritmo 7. O algoritmo `VCubeAcceptorsFase1` executado pelos *acceptors* é mostrado como Algoritmo 8.

No algoritmo `VCubeProposerFase1` o *proposer* envia o *prepare-request* para o maior *cluster* de índice $logN$. O número de respostas inicialmente é 1, apenas a do próprio *proposer*. No algoritmo `VCubeAcceptorFase1` os *acceptors* vão concatenando seus *prepare-responses*. Ao receber o *prepare-request*, o *acceptor* verifica se o número de rodada recebido é maior que aquele que mantém. Neste caso adota o número de rodada recebido, e concatena seu *prepare-response* na mensagem recebida, re-encaminhando para seus *clusters*, alterando os índices de *clusters* adequadamente. O algoritmo usado para determinar quais processos se comunicam é exatamente a difusão descrita no Capítulo 4, com uma diferença: um nodo folha envia a mensagem que concatena todos os *prepare-responses* para o *proposer*. O *proposer* aguarda respostas de todos os nodos folha esperados. Como o VCube funciona como um detector de falhas o *proposer* sabe quantas mensagens deve receber. Caso haja um engano e um processo correto seja indevidamente considero falho, a única consequência é que o *proposer* não vai considerar as respostas deste processo. O *proposer* então processa as mensagens concatenadas,

determinando o maior número de rodada e se há um valor já decidido. Se o número de *prepare-responses* recebido for maior que $n/2$ passa para a Fase 2.

Algoritmo 7: VCubeProposerFase1

```
1  Cluster ← log n // Maior cluster
2  NumPrepResp ← 1 // Proposer já tem seu prepare-response
3  repita
4  |  VCubeAcceptorsFase1(Proposer, NumeroRodada, Valor, Cluster, NULL)
5  |  Aguarde mensagens dos processos folhas do Cluster com as prepare-responses
   |    concatenadas
6  |  Selecione o maior NumeroRodada e o Valor correspondente, se houver
7  |  Atualize NumPrepResp com o número de prepare-responses recebidos
8  |  Cluster ← Cluster − 1
9  até (NumPrepResp > n/2) ou (Cluster = 0)
```

Um *acceptor* executando algoritmo `VCubeAcceptorsFase1` recebe cinco parâmetros: o nodo que está fazendo o broadcast, o número da rodada, o valor, o seu próprio índice do *cluster* e as *prepare responses* concatenadas. Uma chamada ao `VCubeAcceptorsFase1` retorna o número de confirmações obtidas naquele *cluster*. Este algoritmo é apresentado a seguir.

Algoritmo 8: VCubeAcceptorsFase1

```
   Entrada: Proposer, NumeroRodada, Valor, Cluster, mensagens-concatenadas
1  se numRodadaLocal < NumeroRodada então
2  |  numRodadaLocal = NumeroRodada
3  fim
4  Concatena prepare response a mensagens-concatenadas
5  enquanto Cluster > 1 faça
6  |  VCubeAcceptorsFase1(Proposer, NumeroRodada, Valor, Cluster − 1)
7  fim
8  se Cluster = 1 então
9  |  envia mensagens-concatenadas para o Proposer
10 fim
```

É possível que nodos sem-falha sejam incorretamente suspeitos de terem falhado e por isso não recebem o *prepare-request*. Neste algoritmo se o número de falsas suspeitas for menor que $n/2$, a Fase 1 não é afetada. Por outro lado, com $n/2$ ou mais falsas suspeitas o *proposer* não consegue a maioria, e a progressão (*liveness*) do algoritmo fica comprometida. Por lado, a segurança (*safety*) não é comprometida: a decisão ocorre uma única vez devido à maioria de *acceptors* necessária.

A Fase 2 do algoritmo é executada de forma idêntica à Fase 1. O *proposer* encaminha uma mensagem *accept-request* para os *acceptors* exatamente da mesma forma que o *prepare-request* é enviado na Fase 1, atingindo os mesmos *acceptors*, que então podem aceitar o valor enviado. Se entre a Fase 1 e a Fase 2 algum *acceptor* da maioria recebida falhar, uma nova Fase 1 deve ser executada. Ao final, os *learners* aprendem a decisão.

As Figuras 5.1, 5.2 e 5.3 apresentam o caminho que as mensagens percorrem durante a Fase 1 do Paxos utilizando os algoritmos apresentados nesta seção. Nas figuras, a representação é a seguinte. Processos sem-falha que recebem mensagem *prepare-request* são representados por círculos pretos, enquanto processos sem-falha que não recebem mensagem *prepare-request* são representados por círculos cinzas, os processos considerados falhos são representados por círculos vermelhos. Mensagens: *prepare-request* é representada por setas pretas, *prepare-response* por setas verdes.

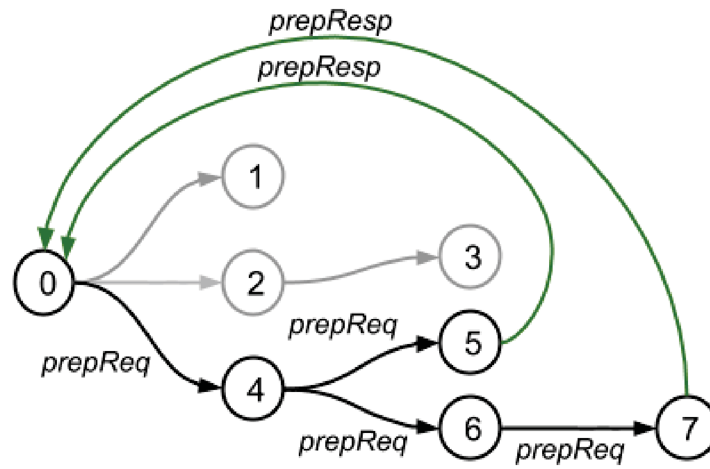

Figura 5.1: Exemplo com todos os processos sem-falha e $n = 8$.

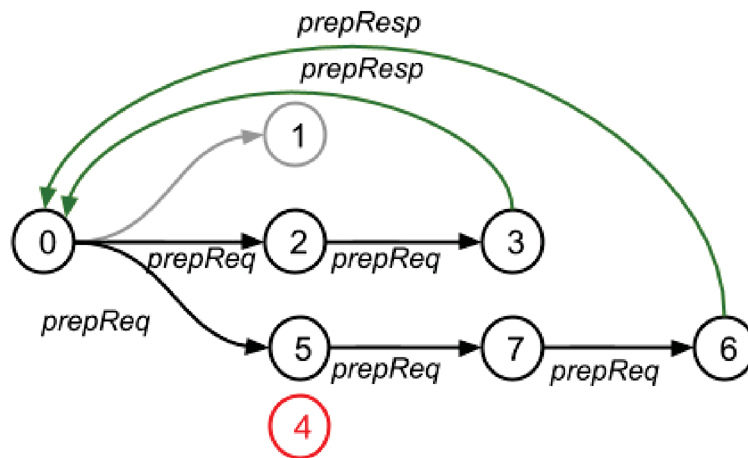

Figura 5.2: Exemplo com um processo falho (4) e $n = 8$.

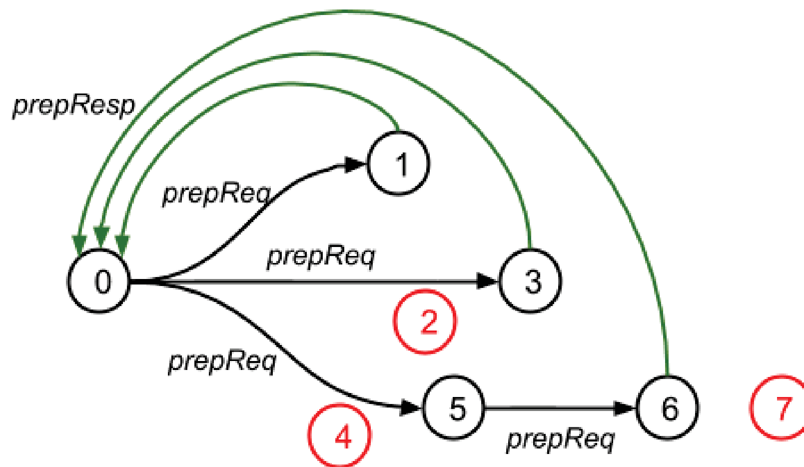

Figura 5.3: Exemplo com três processos falhos (2, 4, 7) e $n = 8$.

São apresentados três exemplos, todos com $n = 8$ e consideram o processo 0 como *proposer* coordenador. Na Figura 5.1 todos os processos estão sem-falha e o *proposer* obtém a maioria enviando a *prepare-request* apenas para seu maior *cluster* ($\log n$). Os *acceptors* folhas 5 e 7 enviam para o *proposer* a mensagem *prepare-response*. Na Figura 5.2 o processo 4 está falho, então não basta enviar a *prepare-request* apenas para o maior *cluster*, assim o processo 0 envia também para o processo 2, que está no *cluster* $\log n - 1$. Neste ponto, a quantidade de respostas já é suficiente para a maioria, mas como o *acceptor* 2 não é folha, ainda encaminha a mensagem para o *acceptor* 3, que também responde a mensagem e, como é folha, envia o *prepare response* para o *proposer*. Na Figura 5.3 há $n/2 - 1$ processos falhos, o número máximo de falhas suportadas pelo Paxos. Nela, os processos 2, 4 e 7 estão falhos, então o *proposer* envia a *prepare request* para todos os *clusters* do VCube até atingir a maioria e recebe três *prepare responses*, sendo neste caso uma de cada *cluster*.

O algoritmo apresentado foi implementado através de simulação, a sua implementação e resultados são apresentados a seguir.

6 IMPLEMENTAÇÃO E RESULTADOS

A implementação do Paxos sobre o VCube foi feita utilizando as linguagens de programação C/C++ e a biblioteca de simulação SMPL (*Simulation Programming Language*) (MacDougall, 1987). Neste capítulo são apresentados os resultados obtidos, incluindo uma comparação com uma versão do Paxos baseada em anel inspirada no M-Ring Paxos.

6.1 CENÁRIOS SIMULADOS

Foram realizados experimentos com número de processos no sistema igual a 8, 16, 32, 64 e 128. O processo escolhido para ser coordenador, em todos os cenários, é o processo de identificador igual a $n/2 - 1$, ou seja, o processo que está no centro da topologia. A escolha do coordenador não interfere nos resultados. Em todos os cenários apresentados adiante, todos os processos são *acceptors*, inclusive o processo coordenador (que é também *proposer*).

São apresentados três cenários: (1) o primeiro com todos os processos sem-falhas. (2) Outro cenário com o número máximo de falhas ($n/2 - 1$), no qual ainda há uma maioria de processos para executar o consenso. (3) E ainda um cenário com apenas $n/4 - 1$ falhas. Nos dois últimos cenários, o *propose* é feito após a falha dos processos programados.

O cenário (2) com o número máximo de falhas foi programado para falharem os processos de identificador ímpar (exceto o processo coordenador). Por exemplo, no sistema com $n = 8$, os processos 1, 5 e 7 estão falhos, e o processo 3 é o coordenador. Esta escolha foi feita para que a mensagem precise percorrer todos os *clusters* de tamanho $s > 1$, sendo este o pior caso do algoritmo, onde são enviadas mais mensagens de resposta para o coordenador e ainda assim demonstrar a eficiência do algoritmo.

No cenário (3) que apresenta $n/4 - 1$ falhas, por sua vez, as falhas dos processos foram organizadas em ordem sequencial. Por exemplo, no cenário com $n = 8$, o processo falho é o processo com identificador igual a 0. No cenário com $n = 32$, os processos falhos são: 0, 1, 2, 3, 4, 5 e 6. Nestes cenários, os processos falhos estão mais agrupados nos *clusters*, então provavelmente há um número menor de envio de mensagens-resposta para o coordenador.

6.2 RESULTADOS EXPERIMENTAIS

A versão do Paxos inspirada no M-Ring Paxos utiliza um algoritmo tradicional de difusão de melhor esforço durante a primeira fase do algoritmo. Assim, durante a primeira fase, são enviadas $n - 1$ mensagens, e os processos sem-falha do sistema respondem à mensagem, gerando até $n - 1$ mensagens. Na segunda fase, os processos que responderam são dispostos em um anel, de forma que são enviadas mensagens *accept request* apenas para os processos sem-falha, os quais responderam à Fase 1 anteriormente. A mensagem *accept response* também é transmitida entre os processos sem-falha. O Paxos sobre o VCube por sua vez apresenta uma redução no número de mensagens para as duas fases do algoritmo, quando a mensagem precisa chegar apenas até uma maioria de *acceptors*. No algoritmo proposto, para cada um dos quatro tipos de mensagem (*prepare request*, *prepare response*, *accept request* e *accept response*) serão enviadas menos do que $n/2$ mensagens, enquanto na versão baseada em M-Ring Paxos, podem ser enviadas até $n - 1$ mensagens de cada tipo.

Para avaliar o impacto do algoritmo, uma versão do Paxos baseada em anel inspirada no M-Ring-Paxos. Ao final de cada simulação foram coletadas informações sobre o número de

mensagens trocadas. A quantidade total de mensagens trocadas até atingir o consenso em ambos os algoritmos pode ser vista nas Figuras 6.1, 6.2 e 6.3. Na Figura 6.1 nenhum processo está falho. Na Figura 6.2 há alguns processos falhos, que correspondem à $n/4 - 1$ processos. Já na Figura 6.3 há $n/2 - 1$ processos falhos, que é a quantidade máxima de falhas permitidas para se obter o consenso.

Nas figuras, há 5 barras vermelhas e 5 azuis, onde as vermelhas indicam a quantidade de mensagens enviadas no algoritmo Paxos sobre o VCube e as barras azuis indicam a quantidade de mensagens enviadas pela versão implementada, que é inspirada no M-Ring Paxos. Os resultados são exibidos para sistemas com 8, 16, 32, 64 e 128 processos. O eixo y representa a quantidade de mensagens enviada por cada algoritmo, e, na Figura 6.1, no eixo x está a quantidade de processos no sistema. Já nas Figuras 6.2 e 6.3, o eixo x apresenta quantos processos falhos há no sistema.

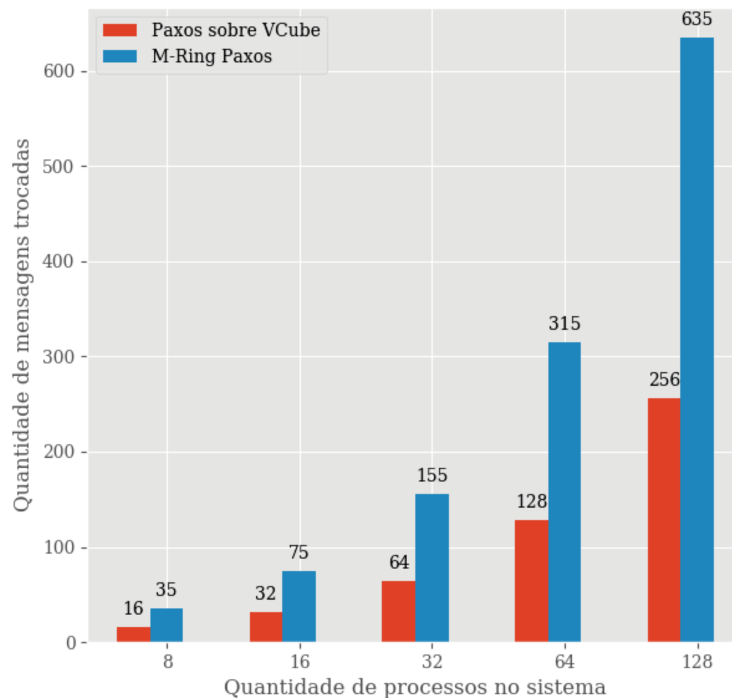

Figura 6.1: Quantidade total de mensagens trocadas no cenário sem-falhas.

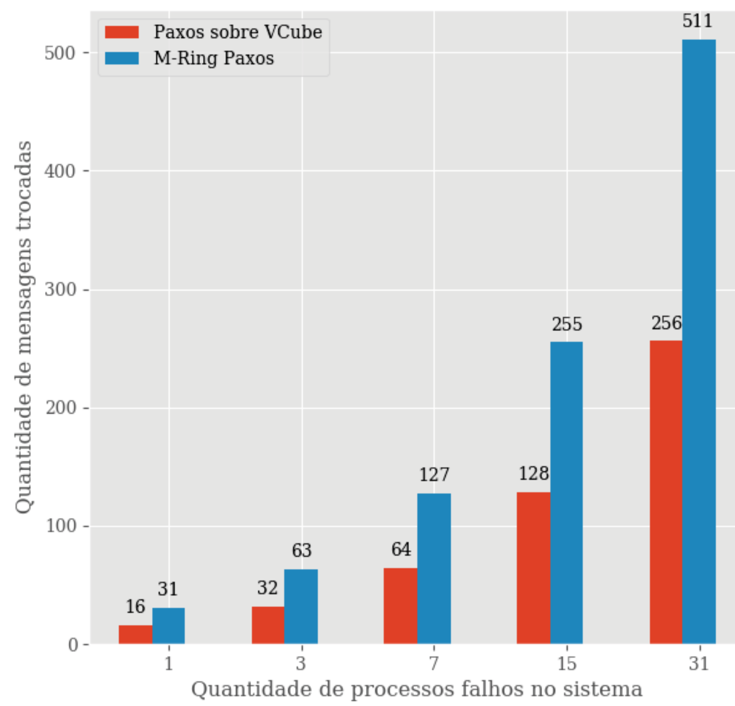

Figura 6.2: Quantidade total de mensagens trocadas no cenário com $n/4 - 1$ falhas.

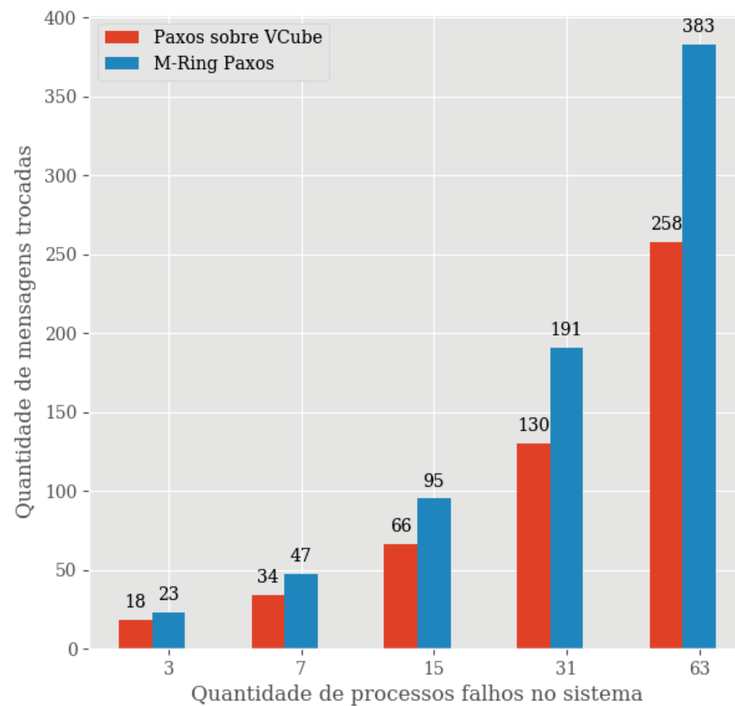

Figura 6.3: Quantidade total de mensagens trocadas no cenário com máximo de falhas permitidas.

Pode-se notar que nos três cenários, o algoritmo Paxos sobre o VCube tem vantagem em relação ao número de mensagens trocadas. Isso se deve principalmente por o algoritmo disseminar as mensagens apenas até atingir uma maioria de *acceptors* sem-falha. Além disso, o algoritmo economiza no número de mensagens ao concatenar mensagens respostas e enviar apenas uma mensagem (contendo todas as respostas) para cada ramo da árvore. Por utilizar o VCube, o algoritmo também evita o envio de mensagens para processos que estão falhos. As mensagens trocadas em ambos os algoritmos comparados para a detecção de falhas não são consideradas nos resultados.

7 CONCLUSÃO

Este trabalho apresentou o algoritmo Paxos sobre o VCube, um algoritmo hierárquico que implementa o Paxos na topologia virtual VCube. O Paxos é um dos mais importantes algoritmos de consenso, que garante que todos os processos em um sistema decidem pelo mesmo valor dentre valores que foram propostos. Reduzir o custo do Paxos pode trazer benefícios para o grande número de aplicações distribuídas que são baseadas neste algoritmo. O VCube é uma topologia virtual para sistemas distribuídos que organiza os processos hierarquicamente, que é escalável por definição, e apresenta diversas propriedades logarítmicas. O Paxos sobre o VCube herda essas propriedades.

O algoritmo proposto organiza os processos *acceptors* no VCube, e o coordenador utiliza uma difusão de melhor esforço sobre o VCube para enviar as mensagens das fases 1 e 2 do Paxos para os demais processos. A difusão sobre o VCube utiliza a organização em *clusters* oferecida pela topologia para disseminar as mensagens de forma autonômica. O serviço de detecção de falhas oferecido pelo VCube permite que a decisão seja tomada mandando mensagem apenas para $(N/2)$ processos, caso estejam sem-falha, evitando possíveis mensagens desnecessárias.

O algoritmo foi implementado através de simulação e foram apresentados resultados em comparação com o Ring Paxos. Os resultados consideram quantidades progressivamente maiores de número de processos no sistema e com diferentes quantidades de processos falhos. Os resultados apresentaram redução no número de mensagens trocadas entre processos durante o consenso quando em comparação com uma versão do Paxos baseada em anel inspirada no Ring Paxos.

A principal vantagem do Ring Paxos é dispor os processos em um anel e, quando há várias instâncias executando, o *pipeline* gera um *throughput* elevado. Como os resultados obtidos para esse trabalho são para apenas uma instância dos algoritmos, um trabalho futuro é estender o algoritmo para permitir a execução de múltiplas instâncias consecutivas do Paxos. Além disso, a implementação em um sistema real é o próximo passo para o algoritmo, que já se mostrou eficiente nos resultados obtidos com simulação.

REFERÊNCIAS

Avizienis, A., Laprie, J.-C., Randell, B. e Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33.

Bianchini, R. e Buskens, R. (1991). An adaptive distributed system-level diagnosis algorithm and its implementation. Em *Fault-Tolerant Computing: The Twenty-First International Symposium*, páginas 222–229.

Cachin, C., Guerraoui, R. e Rodrigues, L. (2011). *Introduction to Reliable and Secure Distributed Programming*. Springer Publishing Company, Incorporated, 2nd edition.

Chandra, T. D., Griesemer, R. e Redstone, J. (2007). Paxos made live: An engineering perspective. Em *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, página 398–407, New York, NY, USA. Association for Computing Machinery.

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

de Camargo, E. T. e Duarte Jr, E. P. (2017). *Tolerância a falhas em sistemas MPI com grupos dinâmicos de processos recomendados e registro de mensagens distribuído baseado em paxos*. Tese de doutorado, UFPR. 110 pgs.

Duarte Jr, E. P., Bona, L. C. E. e Ruoso, V. K. (2014). Vcube: A provably scalable distributed diagnosis algorithm. *Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*, páginas 1–8.

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

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

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

Isard, M. (2007). Autopilot: Automatic data center management. *Operating Systems Review*, 41:60–67.

Jalili Marandi, P., Primi, M., Schiper, N. e Pedone, F. (2017). Ring Paxos: High-Throughput Atomic Broadcast†. *The Computer Journal*, 60(6):866–882.

Kshemkalyani, A. e Singhal, M. (2008). Distributed computing: Principles, algorithms, and systems. *Cambridge University Press*.

Lamport, L. (2001). Paxos made simple. *Sigact News - SIGACT*, 32.

Lamport, L. (2006). Fast paxos. *Distributed Computing*, 19(2):79–103.

MacDougall, M. H. (1987). *Simulating Computer Systems: Techniques and Tools*. MIT Press, Cambridge, MA, USA.

Moraru, I., Andersen, D. G. e Kaminsky, M. (2013). There is more consensus in egalitarian parliaments. Em *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, páginas 358–372.

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

Rodrigues, L. A., Duarte Jr, E. P. e Arantes, L. (2014). Arvores geradoras mínimas distribuídas e autonômicas. *32º Simpósio de Redes de Computadores e Sistemas Distribuídos (SBRC'2014)*, páginas 1–14.

Rodrigues, L. A., Duarte Jr, E. P. e Arantes, L. (2015). Um serviço de multicast confiável hierárquico com o vcube. Em *16o Workshop de Tolerância a Falhas (WTF)*, páginas 1–14.

Schneider, F. B. (1990). Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319.

Van Renesse, R. e Altinbuken, D. (2015). Paxos made moderately complex. *ACM Comput. Surv.*, 47(3):42:1–42:36.