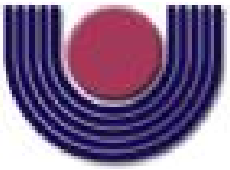


Unioeste - Universidade Estadual do Oeste do Paraná
CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS
Colegiado de Ciência da Computação
Curso de Bacharelado em Ciência da Computação

Uma Solução de Broadcast Hierárquico Tolerante a Falhas

Daniel Vasata

CASCAVEL
2015

Daniel Vasata

Uma Solução de Broadcast Hierárquico Tolerante a Falhas

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. Luiz Antonio Rodrigues

CASCAVEL
2015

Daniel Vasata

Uma Solução de Broadcast Hierárquico Tolerante a Falhas

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. Luiz Antonio Rodrigues
Colegiado de Ciência da Computação,
UNIOESTE

Prof. Alexandre Augusto Giron
Colegiado de Engenharia da Computação, UTFPR

Prof. Guilherme Galante
Colegiado de Ciência da Computação,
UNIOESTE

Cascavel, 19 de fevereiro de 2016

DEDICATÓRIA

Dedico este trabalho aos meus pais e minha família por sempre me incentivarem nesta árdua caminha.

AGRADECIMENTOS

Primeiramente a Deus que permitiu que tudo isso acontecesse, e que desde o inicio de minha vida é fonte de inspiração.

À minha família, onde obtive o apoio e o incentivo para seguir em frente.

Aos meus amigos que me proporcionaram momentos alegres quando passava por momentos difíceis.

Aos meus professores que me acompanharam ao longo de minha formação e ao meu orientador Luiz Antonio Rodrigues, pelo auxilio na construção deste trabalho.

Lista de Figuras

Lista de Tabelas

Sumário

Resumo

Um sistema distribuído pode ser definido como um conjunto de computadores independentes e interligados por uma rede apresentando-se ao usuário como um único sistema. Em um sistema distribuído podem ser utilizadas difusões confiáveis para envio de mensagens, garantindo a entrega de mensagens a partir de um emissor (fonte) a todos os demais processos corretos do sistema distribuído, garantindo suas propriedades. Todo sistema distribuído está sujeito a falhar, assim, a detecção e tratamento de falhas é um requisito para um sistema tolerante a falhas. Para detecção de falhas, é utilizado o VCube, que consiste de um sistema de diagnóstico de falhas que organiza os processos em uma topologia de hipercubo virtual. Este trabalho propõe um algoritmo de difusão confiável para sistemas sujeitos a falhas de colapso com recuperação fazendo uso da organização lógica dos processos e da detecção de falhas do VCube. Resultados de simulações mostram a eficiência da solução proposta, especialmente em cenários com falhas/recuperações.

Palavras-chave: *Crash-recovery*, Difusão Confiável, Simulação, Sincronismo, Sistema Distribuído, VCube.

Capítulo 1

Introdução

Um sistema distribuído (SD) pode ser definido como uma "coleção de computadores independentes entre si que se apresenta ao usuário como um sistema único e coerente"[Tanenbaum 2007]. Em geral, um sistema distribuído é composto por um conjunto P, de n processos $(p_0, p_1, ..., p_{n-1})$ que se comunicam em uma rede por troca de mensagens.

A difusão confiável (*reliable broadcast*) em um sistema distribuído garante a entrega de mensagens a partir de um emissor (fonte) a todos os demais processos corretos do sistema garantindo as propriedades de validade, integridade e acordo [Guerraoui e Rodrigues 2006]. Em uma rede local, por exemplo, o *broadcast* é utilizado pelo Protocolo de Resolução de Endereços (ARP - *Address Resolution Protocol*), uma mensagem para todos os computadores ligados a rede para adquirir o endereço MAC (*Media Access Control*) de um computador especifico. No entanto, em algumas redes como a Internet, esse serviço não está disponível. Neste caso, uma técnica simples é enviar mensagens ponto-a-ponto, mas essa opção não é escalável quando o número de processos é muito grande. Nesse sentido é comum a organização dos processos em estruturas lógicas como árvores.

Uma característica intrínseca dos SDs é a ocorrência de falhas. Assim, a detecção e tratamento delas é um requisito para prover tolerância a falhas. O VCube [Ruoso 2013] é um sistema de diagnóstico de falhas que organiza os processos em uma topologia virtual baseado em um hipercubo com n nodos. No VCube, cada nodo testa seu vizinho no hipercubo e obtém informações sobre os processos do sistema. Em caso de falhas, o hipercubo é reestruturado mantendo as propriedades logarítmicas da topologia.

Existem três principais tipos de falhas em um sistema distribuído: as falhas de colapso, omissão e temporização. Neste trabalho em especial, será trabalhado apenas com as falhas

de colapso (*crash*) e sua possibilidade de recuperação e retorno ao sistema (*crash-recovery*). A principal característica desta falha é que o processo falho fica impossibilitado de executar qualquer ação no sistema. Após a falha, o processo pode retornar ao estado inicial ou a um estado previamente salvo. Neste trabalho foi implementada uma solução que a salva o estado a cada mensagem recebida, possibilitando o retorno a um estado armazenado.

Este trabalho tem como objetivo propor uma solução de *broadcast* confiável em um sistema distribuído baseado na estrutura lógica do VCube considerando a possibilidade de falhas do tipo *crash-recovery*. A solução proposta garante as propriedades de validade, integridade e acordo. O objetivo foi dividido em quatro partes:

- Levantamento de soluções de *broadcast* confiável;
- Propor uma solução hierárquica considerando o VCube;
- Implementação da solução utilizando o simulador NEKO;
- Comparação da solução implementada com outras soluções considerando eficiência(latência e número de mensagens).

Um desafio observado nos estudos iniciais será garantir que todos os nodos do sistema recebam a mensagem que esta sendo repassada caso o nodo esteja correto enquanto a mesma é válida, isto é, enquanto o *broadcast* ainda não foi finalizado pelo emissor.

O restante do texto está apresentado da seguinte forma: no Capítulo 2 são apresentadas definições sobre sistemas distribuídos e suas divisões considerando o sincronismo; no Capítulo 3 são apresentadas definições sobre *broadcast* e algoritmos para o seu funcionamento; o Capítulo 4 apresenta o problema a ser solucionado e a técnica adotada para a resolução; o Capítulo 5 são apresentados os resultados obtidos nos testes realizados; por fim, o Capítulo 6 apresenta as conclusões obtidas sobre o trabalho desenvolvido e os trabalhos futuros.

Capítulo 2

Sistemas Distribuídos

Um *modelo* pode ser definido como uma coleção de atributos e um conjunto de regras que define como estes atributos interagem [Mullender 1993]. Com um modelo é possível estudar o comportamento de um sistema por meio de análise teórica ou de simulação. Neste capítulo serão analisados sistemas distribuídos sob o ponto de vista de características referentes à topologia do sistema distribuído, ao sincronismo e às falhas que podem ocorrer.

2.1 Definições Básicas

Um sistema distribuído é um conjunto finito com $n > 1$ processos independentes, que se comunicam usando troca de mensagens para a realização de alguma tarefa [Raynal 2013]. Considera-se que cada processo é executado em um nodo. Esta execução é caracterizada por uma execução de instruções sequenciais que possuem eventos internos para controlar o envio/recebimento de mensagens.

Um SD possui as seguintes características: concorrência, inexistência de relógio global e falhas independentes [Coulouris et al. 2013]. A concorrência é ocasionada pela execução simultânea de todos os nodos do sistema. A inexistência de um relógio global é ocasionada devido a falta de precisão na sincronização dos relógios dos nodos. Esse problema de sincronização ocorre devido a latência das trocas de mensagens entre os nodos.

Os processos de um sistema distribuído se comunicam geralmente de duas formas: ponto-a-ponto e *broadcast* [Hadzilacos e Toueg 1994]. Existem dois termos principais associados à troca de mensagens: *send* que é utilizado para o envio de mensagens e *receive* para o recebimento.

Em redes ponto-a-ponto, cada enlace conecta um par de processos e cada processo pode enviar mensagens para um único destinatário por vez, neste tipo de rede cada nodo precisa se conectar diretamente com os demais. Em uma rede *broadcast*, um canal compartilhado conecta todos os processos, assim cada processo pode enviar uma mensagem para todos os processos simultaneamente. O *broadcast* é geralmente oferecido como um serviço em redes locais, mas não na Internet, por exemplo.

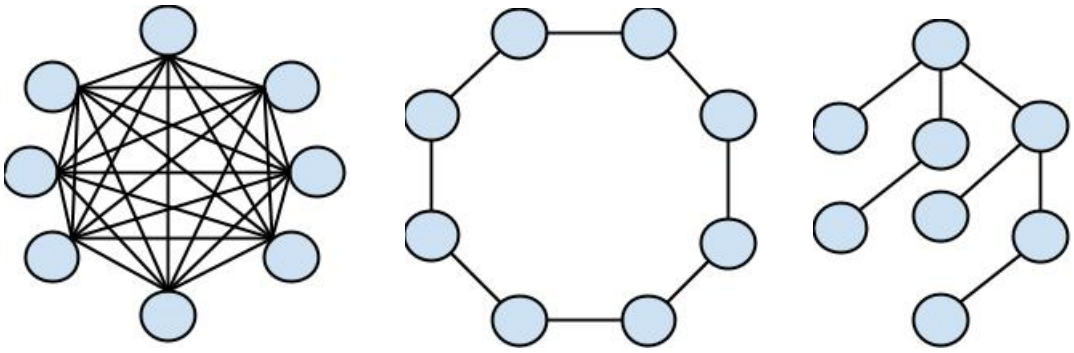
2.2 Topologia

A topologia descreve como os processos de um sistema distribuído podem enviar mensagens entre si [Lamport e Lynch 1990]. Uma topologia é descrita como um grafo $G = (V, E)$ no qual os vértices $V(G) = \Pi$ são os nodos e as arestas $E(G)$ são os enlaces entre os nodos. Um enlace representa a capacidade de dois nodos se comunicarem sem intermediários através de um enlace físico ou lógico. Sendo assim, uma aresta $(i, j) \in E(G)$ indica que o processo i pode se comunicar diretamente com o processo j. Este grafo pode ser direcionado ou não-direcionado. Em um grafo não direcionado, uma aresta (i, j) representa uma comunicação bidirecional $i \rightarrow j$, $j \rightarrow i$.

Em alguns casos cada nodo conhece tem o conhecimento da topologia utilizada, mas existem casos que os nodos não possuem esse conhecimento por completo, no mínimo, cada nodo conhece as arestas que o liga aos processos vizinhos.

A seguir podemos observar três topologias. A Figura 2.1(a) apresenta uma conexão conhecida como todos-para-todos, na qual cada nodo se conecta diretamente com os demais. Esse grande número de conexões possibilita uma fácil construção do sistema de envio de mensagens. Porém, possui um grave problema de escalabilidade, pois para cada novo nodo serão criadas $n - 1$ novas conexões. A Figura 2.1(b) apresenta uma topologia em anel, onde os nodos são conectados em série, formado um circuito fechado (anel). As mensagens são transmitidas por uma determinada direção passando de nodo em nodo até que cheguem aos seus destinos. Nessa abordagem cada nodo irá retransmitir a mensagem para o próximo nodo da sequencia, até que a mensagem chegue ao destino. Essas retransmissões ocasionam um aumento no atraso para cada novo nodo inserido no sistema. A Figura 2.1(c) apresenta uma topologia em árvore, que possibilita uma redução no número de propagações das mensagens quando comparada com a

topologia em anel e também uma melhor escalabilidade quando comparada com a topologia todos-para-todos. Porém, para o seu bom funcionamento, é necessário um rebalanceamento da mesma para cada novo inserido ou removido do sistema.



(a) Todos-para-todos (b) Anel (c) Árvore

Figura 2.1: Exemplos de Topologias de Sistemas Distribuídos

Considere um sistema com n nodos e a latência de envio de uma mensagem sendo 1 unidade de tempo t. Na topologia todos-para-todos, temos que a latência máxima para o envio de qualquer mensagem é 1 t. Enquanto na topologia em anel, a latência de envio pode chegar até $(n-1)t$ em seu pior caso. E na topologia em árvore o pior caso é ht, sendo h a altura da árvore.

2.3 Sincronismo

Um sistema distribuído deve considerar, em geral, dois atributos temporais: a velocidade de execução dos processos e o atraso de mensagens nos canais de comunicação. Dados estes limites, os sistemas podem ser classificados como síncronos ou assíncronos [Cristian 1991].

2.3.1 Sistemas Síncronos

Um sistema síncrono possui os limites de tempo de processamento, envio de mensagens, taxa de variação de relógio e diferença entre relógios locais conhecidos [Veríssimo e Rodrigues 2001]. Segundo Hadzilacos e Toueg (1994) existem três propriedades que caracterizam esse sistema:

1. Existe um limite superior de tempo utilizado por um processador para executar uma tarefa. Dada uma constante $\tau \geqslant 0$, todo processo p executa uma tarefa em no máximo τ unidades

de tempo. Sendo assim, pode-se determinar um Δ no qual um processador pode executar mais rápido que os outros [Turek e Shasha 1992].

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

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

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

Fetzer e Cristian (1995) acrescentam que dois processos p e q possuem seus relógios sincronizados se existir um parâmetro de erro máximo ε para todo tempo t tal que:

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

Os limites temporais podem ser definidos em quantidade de passos executados pelos processos, ao invés de serem baseados em termos de tempo real [Freiling, Guerraoui e Kuznetsov 2011]. Sendo assim, Δ e Φ tem os seguintes significados:

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

Assim, o limite ω correspondente ao tempo de ida e volta de uma mensagem trocada entre os processos p e q, pode-se definir como:

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

Inicialmente, são necessários Φ passos entre o envio da mensagem pelo processo p e a recepção no processo q. Logo após, considera-se que um processo utiliza s passo para receber e enviar uma resposta, serão necessários $s * \Delta$ passos para receber, processar e responder cada mensagem. Ao final, após Δ passos de processamento em q, a mensagem de resposta alcança

p. Considerando que q pode ser mais lento que p e Δ é medido em q, a multiplicação de Δ por Φ define um limite superior para q executar Δ passos.

A maior limitação em assumir um sistema síncrono é a dificuldade de garantir os limites temporais. Essas garantias exigem um controle em diversos aspectos do sistemas, inclusive na sua carga de processamento, o que exige usualmente *hardware* e *software* apropriados. Em redes locais, as condições podem até ser satisfeitas, porém se torna muito difícil estabelecer as condições para redes em larga escala.

2.3.2 Sistemas Assíncronos

Em um sistema assíncrono os limites temporais não são conhecidos, o que impossibilita qualquer asserção nesse sentido e assim impossibilitando a garantia que uma execução termine no tempo esperado [Krakowiak e Shrivastava 1999].

Considera-se que os processos não possuem acesso a relógios globais, embora uma sequência de eventos no sistema possa ser definida com o auxílio de relógios lógicos [Lamport 1978]. A passagem de tempo em um sistema pode ser calculada da seguinte maneira [Guerraoui e Rodrigues 2006]:

- Cada processo p possui um contador próprio l_p inicializado em 0;
- Para cada instrução executada por p, o contator l_p é incrementado em uma unidade;
- Quando p envia uma mensagem, envia junto o valor de seu contador l_p. Esta marcação do evento e, denominada (*timestamp*), é denotada por $t(e)$;
- Quando um processo q recebe uma mensagem de p com *timestamp* l_p, altera o seu contador para o maior valor entre o l_p local e o l_p da mensagem, e depois incrementa em uma unidade.

Desta forma, sejam dois eventos quaisquer e_1 e e_2, diz-se que $e_1 \rightarrow e_2$ (e_1 precede e_2) caso: (a) e_1 e e_2 ocorram ao mesmo processo e e_1 aconteceu antes de e_2; (b) e_1 é um envio de mensagem do processo p para o processo q e e_2 é a recepção desta mensagem; ou (c) existe um evento e' tal que $e_1 \rightarrow e'$ e $e' \rightarrow e_2$. Considera-se o *timestamp*, $e_1 \rightarrow r_2 \Rightarrow t(e_1) < t(e_2)$.

Embora relógios lógicos sejam úteis para algumas aplicações, a ausência de limites temporais impossibilita a implementação de soluções determinísticas para problemas de acordo em sistemas assíncronos que não permitem falhas. A dificuldade está na determinação do estado dos processos distribuídos, sendo que não é possível distinguir processo falho de processo lento [Fischer, Lynch e Paterson 1985].

2.4 Modelo de Falhas

Falhas independentes podem ser ocasionadas por diferentes motivos, afinal, todo sistema de computador pode falhar em algum momento e o sistema distribuído deve estar preparado para quando a falha ocorrer.

Em um sistema baseado em troca de mensagens, existem dois tipos de falhas: falhas de comunicação e falhas de processo [Lamport e Lynch 1990]. Falhas de comunicação geralmente resultam em perda de mensagens, duplicação ou corrupção de dados. E uma falha de processo ocorre quando o comportamento do algoritmo em execução foge do que foi especificado.

Durante a comunicação, mensagens podem ser perdidas por falha, omissão e temporização/desempenho [Mullender 1993, Jalote 1994, Birman 1996]. Estas são caracterizadas pelo não envio e/ou não recebimento de mensagens que deveriam ter sido enviadas e/ou recebidas por um determinado processo. Podem ser causadas por falta de espaço nos *buffers* do sistema operacional e da interface de rede ou por erros de transmissão detectados no receptor. Enquanto as falhas de temporização ocorrem quando uma propriedade temporal do sistema é violada, como a superação do limite de variação do relógio ou quando uma mensagem é entregue com um atraso maior que o tolerado pelos processos.

Existem também as falhas de colapso (*crash*). Durante esta falha, o processo não executa qualquer ação e não responde aos estímulos externos, ou seja, não executa processamento, nem envio ou recebimento de mensagens.

O modelo de falhas mais abrangente é aquele que considera falhas arbitrárias, também conhecidas como falhas bizantinas [Lamport, Shostak e Pease 1982]. Estas falhas incluem, além de falhas de comunicação e processo descritas anteriormente, aquelas geradas devido a comportamentos maliciosos.

Em uma definição hierárquica, as falhas de colapso são especificações das falhas de omis-

são, que por sua vez, são englobadas pelas falhas de temporização e as falhas arbitrárias [Jalote 1994]. Sendo assim, um sistema projetado para assumir falhas arbitrárias deve ser capaz de assumir qualquer tipo de falha. No entanto, a maior parte das aplicações implementam modelos que tratam apenas as falhas de colapso, visto que as falhas de omissão podem ser evitadas utilizando canais confiáveis e as falhas de temporização são abstraídas do modelo de sincronismo adotado.

Capítulo 3

Difusão de Mensagens em Sistemas Distribuídos

Difusão de mensagens (*broadcast*) é um componente básico para implementar muitos algoritmos e serviços distribuídos como entrega e replicação de conteúdo, notificação e comunicação em grupo [Leitão, Pereira e Rodrigues 2007, Yang, Li e Lou 2009, Bonomi, Pozzo e Baldoni 2013].

Neste capítulo são apresentados conceitos e algoritmos refentes à difusão de melhor-esforço (*best-effort broadcast*) e difusão confiável (*reliable broadcast*).

3.1 Difusão de Melhor-Esforço

A difusão de melhor-esforço (*best-effort broadcast*) garante que todos os processos corretos entregam o mesmo conjunto de mensagens se o emissor (fonte) estiver correto. Este modelo possui três propriedades [Guerraoui e Rodrigues 2006]:

- Entrega confiável: garante que se um processo i envia uma mensagem m para um processo j e nenhum deles falha, j recebe m em tempo finito;
- Não-duplicação de mensagens: garante que nenhuma mensagem é entregue mais de uma vez;
- Não-criação de mensagens: garante que nenhuma mensagem é entregue a menos que tenha sido previamente enviada.

Observa-se que em função da falha do processo fonte, alguns processos podem receber a mensagem enquanto outros não, o que não invalida as propriedades.

Durante uma difusão por melhor-esforço, o processo inicial espera uma mensagem de retorno para confirmação do recebimento da mensagem. Quando a mensagem de confirmação não retorna em um tempo pré estabelecido (em sistemas síncronos) é detectada uma falha.

Em uma execução sem falhas, o nodo inicial envia as mensagens de *broadcast* e aguarda as mensagens de confirmação. Após o recebimento de todas as confirmações o *broadcast* é dado como finalizado.

No contexto de falhas de *crash*, temos um possível caso de falha. Se processo p_i, que não é fonte, falha durante a execução do *broadcast* e não se recupera antes do término. Neste caso, o processo que está realizando a difusão deve garantir que os demais processos receberam a mensagem, pois p_i pode ser responsável pela propagação da mensagem. Note que a falha do fonte não invalida o protocolo.

3.2 Difusão Confiável

Um algoritmo de difusão confiável (*reliable broadcast*) garante que o mesmo conjunto de mensagens é entregue a todos os processos corretos, mesmo se o processo emissor (fonte) falhar durante o procedimento de envio. A difusão confiável herda as propriedades da difusão por melhor-esforço (entrega confiável, não-criação e não duplicação de mensagens) e acrescenta a propriedade de acordo (*agreement*). A propriedade de acordo consiste em garantir que todo processo não falho receba a mensagem em difusão, independente do estado do processo emissor.

Na difusão confiável observa-se quatro cenários de execução quando analisa-se as falhas de *crash*, sendo esses:

- Processo fonte e receptor correto. Neste caso a mensagem de *broadcast* será recebida pelo receptor.

- Processo fonte está correto, mas o processo receptor está falho. Neste caso, a mensagem deve ser retransmitida para os processo dependentes do processos falho para garantir que todos os nodos do sistema recebam a mensagem.

- Processo fonte falho e processo receptor está correto. Quando o processo receptor detecta que o processo fonte não está executando, o mesmo deve garantir a propagação para todos os processos corretos do sistema, assim garantindo a propriedade de acordo.

- Processos fonte e receptor falham. Neste caso, se um terceiro processo tenha recebido a mensagem de *broadcast* antes do processo fonte falhar, o mesmo irá retransmitir a mensagem para garantir que todos os nodos corretos recebam a mensagem.

3.3 Algoritmos de Difusão Confiável

O algoritmo um-para-todos é o mais simples para a difusão de mensagens. O mesmo consiste do nodo que deseja enviar uma mensagem de *broadcast* enviar a mensagem individualmente para cada nodo do sistema distribuído. No entanto, essa técnica exige o envio de $n - 1$ mensagens, onde n é o número de nodos do sistema, sobrecarregando o nodo inicial com envio de mensagens. A Figura 3.1 apresenta um exemplo utilizando esse algoritmo, onde o nodo 0 deseja realizar um *broadcast* em um sistema com $n = 8$ nodos.

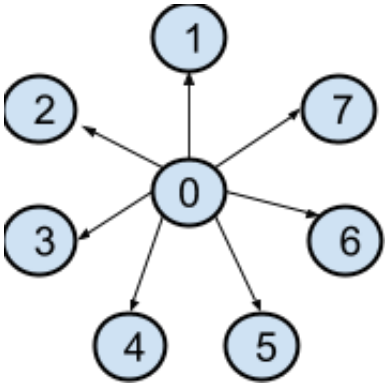


Figura 3.1: Exemplo de *broadcast* Utilizando a Técnica Um-Para-Todos

Em casos de falha no processo receptor, não será necessária uma retransmissão da mensagem, pois o processo inicial garante que os demais nodos receberam a mensagem. No entanto quando o processo inicial falha, todos os processos que receberam a mensagem, e detectaram que o processo inicial falhou irão retransmitir a mesma para os processos do sistema, gerando assim no pior caso um envio de $(n - f)^2$ mensagens, sendo n o número de processo do sistema e f o número de processos falhos do sistema.

Rodrigues, Duarte e Arantes (2007) propuseram um algoritmo que utiliza uma árvore como rota de transmissão da mensagem. Nesse sistema, o nodo inicial (raiz) envia a mensagens para os nodos que está conectado (ramos). Os nodos internos na árvore, ao receberem a mensagem de *broadcast* irão repassar a mesma para as suas conexões. Esse método resolve o problema de

sobrecarga de envios de mensagem no nodo inicial, pois o mesmo distribui os envios para os demais nodos.

A Figura 3.2 apresenta um exemplo do funcionamento desta técnica. Onde o nodo 0 inicia o *broadcast* (Figura 3.2(a)). Quando os nodos 1, 2 e 4 recebem a mensagens, os mesmos irão retransmitir a mensagem para as suas conexões como apresentado na Figura 3.2(b). Por fim, a Figura 3.2(c) apresenta a ultima mensagem enviada durante a execução deste *broadcast* quando o nodo 7 recebe a mensagem.

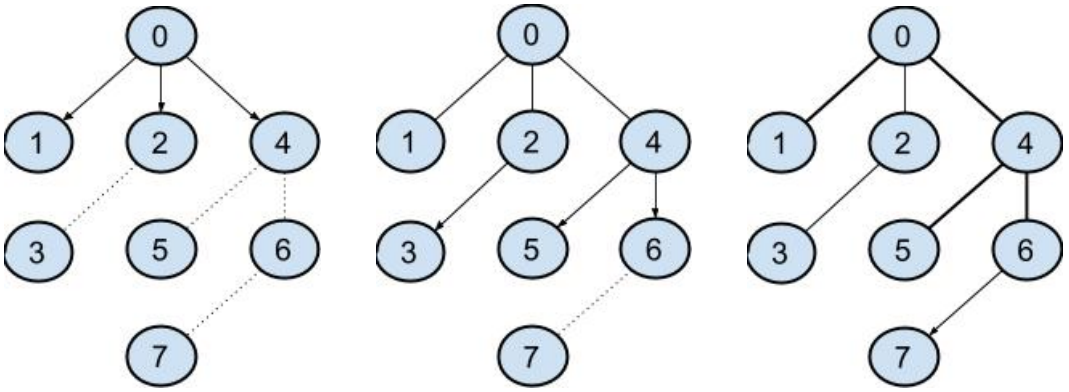


Figura 3.2: Exemplo de propagação de mensagens na árvore de difusão

Em caso de falhas no nodo receptor, a árvore é readaptada garantindo o envio para todos os processos corretos do sistema. Caso o nodo inicial falhe, o processo que detectar a falha e obteve a mensagem antes do inicial falhar, retransmite a mensagem para sua árvore garantindo a propriedade de acordo. Essa decisão também é valida para o caso onde o processo inicial e o receptor falham.

Capítulo 4

A Solução de Difusão Confiável Proposta

Todo sistema distribuído esta propenso a falhas de *crash*. Dado este problema, é desejável que o nodo falho se recupere da falha e seja reintegrado ao sistema. Neste trabalho é proposta uma solução para o problema considerando a recuperação do processo (*crash-recovery*).

Durante os estudos observou-se que um processo falho pode retornar durante a execução de um *broadcast*, sendo assim deve-se garantir que o mesmo receba a mensagem para satisfazer a propriedade de acordo. A Figura 4.1 apresenta um exemplo deste caso, onde o nodo 2 não está no sistema e retorna a ele antes da conclusão do *broadcast*. Os intervalos entre as linhas pontilhadas representam intervalos de tempo. A área em destaque representa o período no qual o nodo 2 está falho, e as linhas entre os nodos 0 e 3 mostram a troca de mensagem dentro do sistema. Observa-se que durante um curto período de tempo o nodo 2 esta ativo no sistema enquanto a mensagem repassada é válida. Este deve receber a mensagem sendo propagada, mas em função da latência de detecção o nodo 0 pode descobrir o retorno do nodo 2 somente após a conclusão do *broadcast*.

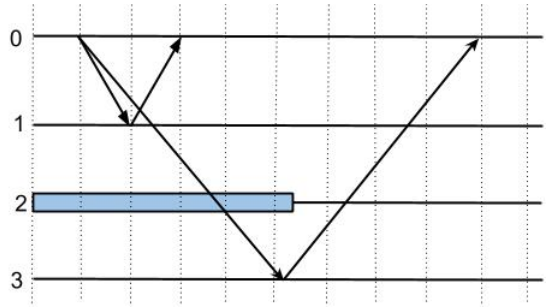


Figura 4.1: Processo 2 retorna durante o *broadcast*

Para a resolução do problema, foi utilizado uma estrutura em forma de cubo bem como a

árvore geradora apresentadas nas próximas seções.

4.1 VCube

Utilizar uma topologia virtual para conectar processos em um sistema distribuído facilita a construção das aplicações, pois abstrai a estrutura física e possibilita a reconfiguração do sistema sempre que necessário. A topologia VCube organiza os processos na forma de um hipercubo virtual quando não possui falhas. Porém, quando uma falha é detectada, os enlaces virtuais são reconfigurados para se adaptar ao novo estado do sistema. Mesmo quando parte do sistema está falho, o VCube mantém as seguintes propriedades logarítmicas: em um hipercubo de d dimensões com $n = 2^d$ vértices, cada vértice está conectado a até $log_2 n$ vizinhos e a distância máxima entre dois vértices é $log_2 n$, no máximo.

Na construção do VCube é utilizada a organização hierárquica proposta por Duarte Jr e Nanya (1998). Os processos são organizados em clusters progressivamente maiores, conforme apresentado na Figura 4.2. Cada cluster $s = 1, ..., log_2 n$ possui 2^{s-1} processos. Nos clusters de nível 1, cada cluster possui um elemento. No segundo nível, são agrupados clusters de tamanho um, formando um cluster de tamanho dois. No terceiro nível, dois clusters de tamanho dois são unidos para formar um terceiro cluster de tamanho quatro, e assim sucessivamente.

O VCube implementa um algoritmo para detecção de processos falhos, onde cada nodo verifica os estados de seus vizinhos e dos nodos de cada cluster a cada intervalo de tempo previamente estabelecido. Quando o nodo verifica o estado do de seu vizinho, caso o vizinho possua informações mais atualizadas que as suas com relação ao estado dos demais processos, o mesmo se atualiza com as informações adquiridas.

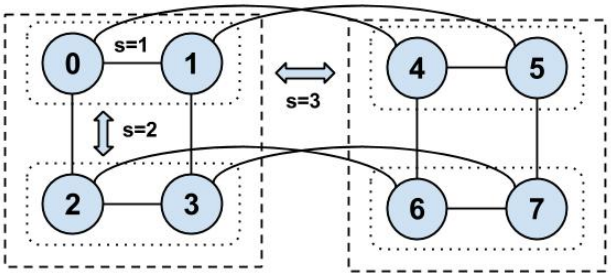


Figura 4.2: Organização do VCube com três dimensões

Os processos integrantes de cada cluster s relacionados a um processo i são listados através

da função abaixo, onde $\oplus$ representa a operação binária *ou* exclusivo (*xor*).

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

A topologia do VCube é formada pela conexão de cada processo i com seus vizinhos. No trabalho de Rodrigues, Duarte Jr. e Arantes (2014) baseado no VCube [Duarte e Nanya 1998] foi proposto um algoritmo que constrói uma árvore geradora sobre a topologia do VCube, essa técnica é descrita na próxima seção.

4.2 Árvore Geradora

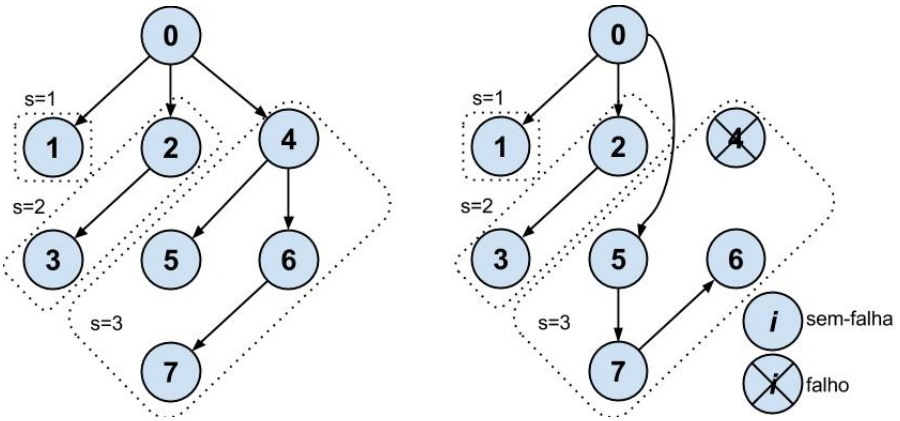
O algoritmo para construção da árvore é proposto a partir de um mecanismo de disseminação de informação para um sistema distribuído que propaga as mensagens do sistema a partir de um processo qualquer, conhecido como processo fonte. Este algoritmo é considerado autonômico devido a sua capacidade de reconstrução da árvore dinamicamente à medida que processos falhos são detectados.

Como exemplo considere uma árvore com 8 processos sem falha, tendo como nó raiz p_0. Inicialmente são atribuídos como filhos de p_0 aqueles que resultam como vizinho de p_0 no calculo 4.1. Logo após a inserção dos filhos de p_0, cada filho irá calcular os seus vizinhos que estão dentro de seu cluster utilizando novamente 4.1 e assim por adiante até que não haja mais nenhum nodo a ser inserido no sistema.

Considere um sistema inicial sem falhas. O primeiro passo é enviar uma mensagens para cada um dos $log_2 n$ vizinhos sem falha no VCube. O processo vizinho ao receber a mensagem inicial, propaga a mensagem para os *clusters* internos ao seu próprio *cluster*.

Considere um VCube com 8 nodos sem falhas, como apresenta a Figura 4.3(a). O processo p_0 é a raiz da árvore, que envia as mensagens para os nodos vizinhos p_1, p_2 e p_4. O processo p_1 recebe a mensagem mas não propaga, dado que p_1 não possui nodos em seu cluster. No entanto, como p_2 possui p_3 em seu *cluster*, o mesmo repassa a mensagem ao processo vizinho. Quando p_4 recebe a mensagem de p_0, o mesmo detecta que deve propagar a mensagem aos processos $p_5 \in C_{4,1}$ e $p_6 \in C_{4,2}$. E por fim p_6 propaga a mensagem para p_7.

A Figura 4.3(b) demonstra um caso de falha, sendo o processo p_0 raiz e p_4 o processo falho. Considerando que p_0 esteja ciente da falha, ao invés de enviar a mensagem a p_4, p_0 transmite a



(a) sem-falhas (b) processo $p4$ falho

Figura 4.3: Árvore geradora no VCube com 8 nodos

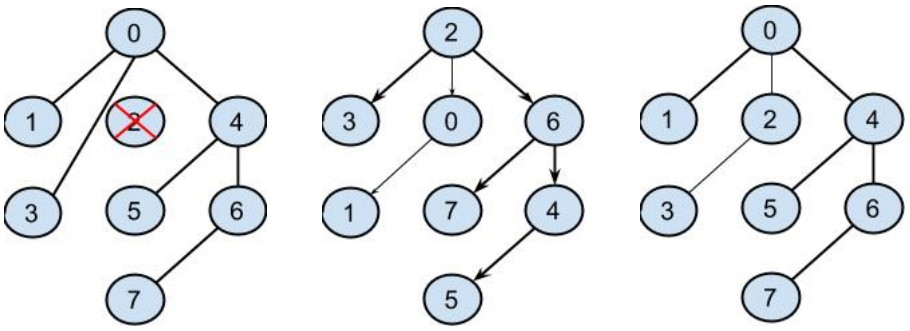
mensagem para p_5 que é o primeiro processo sem falha de $C_{0,3} = (4, 5, 6, 7)$. p_5 por sua vez, transmite a mensagem para $p_7 \in C_{5,2}$. Ao final, p_7 retransmite a mensagem para $p_6 \in C_{7,1}$, completando assim a árvore. Caso p_0 não estivesse ciente da falha no momento que inicia o *broadcast*, p_0 iria enviar a mensagem para p_4 e quando a falha de p_4 fosse informada a p_0 pelo VCube, a mensagem seria retransmitida para p_5, como explicado anteriormente.

4.3 O Algoritmo de Difusão Confiável Proposto

O algoritmo aqui proposto, segue a topologia de árvore gerada pelo algoritmo VCube. Sendo assim, a propagação da mensagem no sistema será realizada utilizando a árvore geradora descrita na seção anterior.

Visando uma possibilidade de retorno de p_i foi proposto um algoritmo para a reinserção do mesmo no sistema. Ao recuperar-se da falha p_i é reinicializado para evitar uma possível inconsistência de dados. Logo após a reinicialização, o processo envia uma mensagem para cada nodo do sistema informando seu retorno. Os demais nodos do sistema ao receberem a mensagem de p_i reconstroem sua árvore de *broadcast*.

A Figura 4.4 apresenta um exemplo da solução proposta. Inicialmente p_2 esta falho, como apresentado na Figura 4.4(a) a árvore com raiz em p_0. Após recuperar-se do *crash*, p_2 será reinicializado e enviará para os demais nodos do sistema uma mensagem informando seu retorno através de sua própria árvore, como na Figura 4.4(b). A Figura 4.4(c) apresenta a nova árvore gerada após o retorno de p_2 no sistema.



(a) p_2 esta falho (b) p_2 informa seu retorno (c) p_2 é reinserido no sistema

Figura 4.4: Exemplos da Solução Proposta Para o Retorno do Processo ao SD

Como dito inicialmente, um processo pode retornar do estado de falha durante a execução de um *broadcast*. No entanto, o mesmo processo receberá a mensagem que esta em *broadcast* apenas se informar seu retorno ao nodo que inicializou a troca de mensagem antes do termino do *broadcast*, garantindo assim a propriedade de acordo.

A Figura 4.5 apresenta um exemplo de linha de execução de um *broadcast*,onde o intervalo entre as marcações feitas em p_0 representam o tempo de vida da difusão. As setas de número 1 representam a mensagem sendo enviada do processo fonte ao destino. Observa-se que p_2 não recebe a mensagem, dado que o mesmo esta em estado de falha. No entanto a quando p_2 se recupera da falha, o mesmo informa seu retorno a p_0 (representado pela mensagem 2) e p_0 reenvia a mensagem para p_2 como mostra a seta de número 3. Por fim, as setas de número 4 representam a mensagem de confirmação do recebimento a p_0. Supondo que a mensagem de retorno de p_2 chegue após o termino do *broadcast*, p_0 irá apenas considerar p_2 como processo válido, mas não haverá reenvio de mensagem.

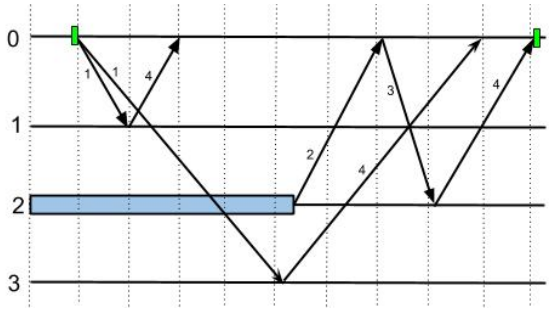


Figura 4.5: Linha de execução com o processo 2 retornando durante um *broadcast*

A solução proposta foi implementada utilizando os procedimentos a seguir. Além dos pro-

cedimentos, foram utilizadas variáveis para controle, sendo estas:

- *correct*$_i$: Vetor do tipo booleano que contem quais processos estão corretos ou falhos no sistema. Cada posição do vetor corresponde a um processo e recebe valor *true* se o processo estiver correto e *false* se o processo estiver falho.

- *ACKpendentes*$_i$: Lista que armazena quais difusões estão ativas e quais confirmações de recebimentos (ACKs) estão pendentes em cada processo. As informações sobre as difusões são armazenadas através de tuplas <origem, destino, mensagem>, onde origem é o processo que enviou a mensagem, destino é o processo processo para o qual a mensagem é enviada e a mensagem que esta sendo enviada.

- *last*: Vetor que contem a ultima mensagem recebida de cada processo.

- *recovery*: variável booleana que contem a informação se o processo está em fase de recuperação.

Além disso, as mensagens possuem dois campos internos, sendo estes:

- *id*: número de identificação da mensagem;

- *source*: número do processo que iniciou a difusão da mensagem.

O procedimento apresentado no Algoritmo 1 é responsável por iniciar a difusão de uma mensagem m no processo i. A função *neighborhood*$_i$ presente no *for* da linha 2 calcula os vizinhos corretos do processo i.

Algorithm 1: procedure Broadcast (Mensagem m) em i

```
1 begin
2     for all k ∈ neighborhood_i (log_2 n) do
3         send(m) para p_k
4         adicionar <p_i, p_k, m> em ACKPendentes_i
5     end
6 end
```

Já o procedimento contido no Algoritmo 2 é responsável pelo tratamento de uma mensagem m recebida de um processo j em p$_i$. Inicialmente é verificado se o processo j está correto. Caso esteja é verificada se m já foi previdamente recebida. Essa verificação deve ser feita pois caso um processo falhe, os demais processos irão realizar uma difusão da mensagem mais recente

recebida do processo falho. Caso m seja mais recente que a armazenada em *last*, a mensagem em *last* é atualizada e *last* é salvo em disco para que em caso de recuperação de i seja possível recuperar quais foram as ultima mensagens recebidas. Se o processo que enviou m estiver falho, o processo irá realizar uma difusão da mensagem e o método *Receive* é finalizado. Se o processo que enviou m estiver correto, é calculado se o número do cluster (cl) utilizando o método $cluster_i(j)$ do processo i, se o mesmo for maior que um, significa que o mesmo possui vizinho para os quais deve enviar a mensagem recebida, esse envio é representado no *for* da linha 12. Por fim, se i não possui vizinhos para enviar m, i envia uma mensagem de confirmação para o processo j confirmando o recebimento de m.

Algorithm 2: procedure Receive (Mensagem m) de p_j

```
1  begin
2      if j ∈ correct_i then
3          if last[m.id].id < m.id then
4              last[m.id]=m
5              save(last)
6              deliver(m)
7              if correct(m.source)=false then
8                  Broadcast(m)
9                  return
10             end
11         end
12         cl=cluster_i(j) - 1
13         if cl > 0 then
14             for all k ∈ neighborhood_i(cl) do
15                 send(m) para p_k
16                 adicionar <p_j, p_k , m> em ACKPendentes_i
17             end
18         end
19         if ACKPendentes_i ∩ <j, *, m> = ∅ then
20             send(<ACK , m>) para p_k
21         end
22     end
23 end
```

Foi criado outro procedimento *Receive*, mas este responsável exclusivamente pelo tratamento das mensagens de confirmação recebidas, este é apresentado no Algoritmo 3. Neste procedimento o processo que enviou a mensagem de confirmação é removido da lista de pendentes e se todas as confirmações pendentes de uma mensagem m forem recebidas é enviada

um ACK para o processo que enviou m para i. Destaca-se que quando um processo finalizar uma difusão de retorno sua variável responsável por armazenar esta informação recebe *false*.

Algorithm 3: procedure Receive (<ACK , m>) de p_j

```
1  begin
2      remove <k, p_j , m> de ACKPrendentes_i
3      if ACKPendentes_i ∩ <k, * , m> = ∅ then
4          recuperacao=false
5          send(<ACK, m>) para p_k
6      end
7  end
```

Assim como o *Receive* anterior, o Algoritmo 4 tem como objetivo de tratar as mensagens de retorno do sistema (*I_AM_BACK*). Neste procedimento o atualiza a informação em *correct* para *true*. Logo após a atualização a mensagem é repassada para os demais processos corretos do cluster. Caso não haja mais processos corretos é enviado um ACK para o j.

Algorithm 4: procedure Receive (<I_AM_BACK, m>) de p_j

```
1  begin
2      correct_i[m.source]=true
3      cl=cluster_i(j) - 1
4      if cl > 0 then
5          for all k ∈ neighborhood_i(cl) do
6              send(m) para p_k
7              adicionar <p_j, p_k , m> em ACKPendentes_i
8          end
9      end
10     if ACKPendentes_i ∩ <j, * , m> = ∅ then
11         send(<ACK , m>) para p_j
12     end
13 end
```

O procedimento *Crash* apresentado no Algoritmo 5 é executado quando um processo é detectado como falho. Dado o processo j falho, i deve atualizar sua lista de processos corretos e verificar qual o cluster de j, essa detecção é feita através do método *FF_neighbor$_i(j)$*. Após a detecção, o *for* da linha 4 tem como objetivo reenviar as mensagens em difusão para o cluster obtido. Quando uma mensagem é reenviada é adicionada uma nova pendencia em ACKPendentes$_i$ e a pendencia antiga é removida. Caso não haja mais pendencias sobre uma mensagem, é en-

viado um ACK para o processo que enviou m para i. Por fim, i realiza uma difusão da ultima mensagem recebida de j para garantir que todos os processos corretos do sistemas recebam a ultima mensagem enviada por j.

Algorithm 5: procedure Crash (Processo j) em i

```
1  begin
2      correct[j]=false
3      k=FF_neighbor_i(cluster_i(j))
4      for all <x, j, m> ∈ ACKPendentes_i do
5          if k ≠ null then
6              send(m) para p_k
7              adicionar <x, p_k, m> em ACKPedentes_i
8          end
9          remove <x, j, m> de ACKPendentes_i
10         if ACKPendentes_i ∩ <x, *, m> then
11             send(<ACK,m>) para p_x
12         end
13     end
14     Broadcast(last_i[j])
15 end
```

Quando um processo se recupera de uma falha, o procedimento do Algoritmo 6 é iniciado. O mesmo recupera as informações de quais mensagens o processo já havia recebido através da função *load* que pega os dados armazenados em disco pela função *save*. Dada a recuperação do histórico de mensagens recebidas, o processo entra em fase de recuperação e inicia uma difusão informando o seu retorno ao sistema.

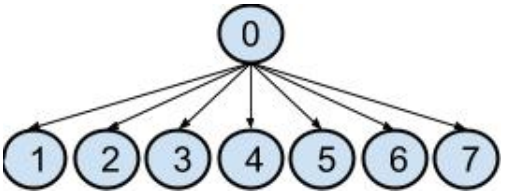
Algorithm 6: procedure Recover ( ) em i

```
1  begin
2      last_i=load()
3      recuperacao=true
4      Broadcast(I_AM_BACK)
5  end
```

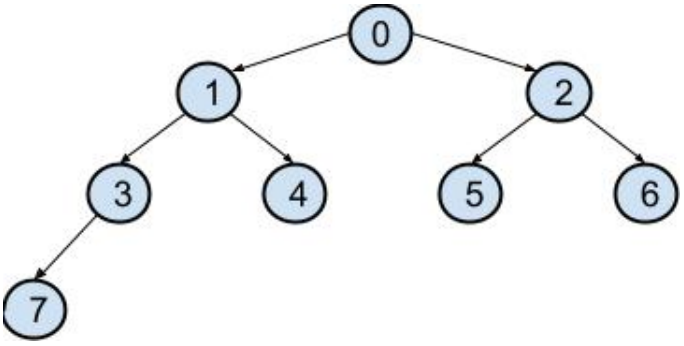
Capítulo 5

Análise dos Resultados

O Capítulo 4 apresentou uma solução de difusão confiável utilizando uma árvore geradora. A título de comparação foram selecionadas duas técnicas de *broadcast*, uma baseada em uma *Flat Tree* (Figura 5.1(a)), que na prática consiste de um sistema um-para-todos e outra em uma *Binary Tree* (Figura 5.1(b)) que é construída através de uma inserção em largura [Pješivac-Grbović et al. 2007].



(a) *Flat Tree*



(b) *Binary Tree*

Figura 5.1: Exemplos das árvores utilizadas para comparação.

Para a realização da simulação foi utilizado o simulador NEKO [Urban, Défago e Schiper 2001]. NEKO é um *framework* Java utilizado para construção de algoritmos distribuídos. Estes algoritmos podem ser simulados ou executados em uma rede

real.

As configurações da máquina na qual as simulações foram executadas são: Processador Intel Core i7-2620M CPU @ 2.70GHz x4, com 8,0 GiB de memória RAM utilizado o sistema operacional Ubuntu 14.04.

Foram avaliados cinco cenários, estes descritos na sequência com seus resultados e comparações, em todos os cenários o processo 0 simula a difusão da mensagem de uma aplicação.

Em todos os cenários e algoritmos o VCube é utilizado para detecção de falhas.

O ambiente foi configurado com tempo de processamento de um envio igual a 0,1 e o tempo de envio igual a 0,9 unidade de tempo. O tempo para o VCube executar a rotina de *timeout* é 5 intervalos de tempo.

5.1 Cenário Sem Falhas

Primeiramente foram obtidos os dados no cenário sem falhas para a duração, e o número de mensagens transmitidas em cada *broadcast*. Neste cenário foi realizado uma difusão de mensagem no tempo 500.

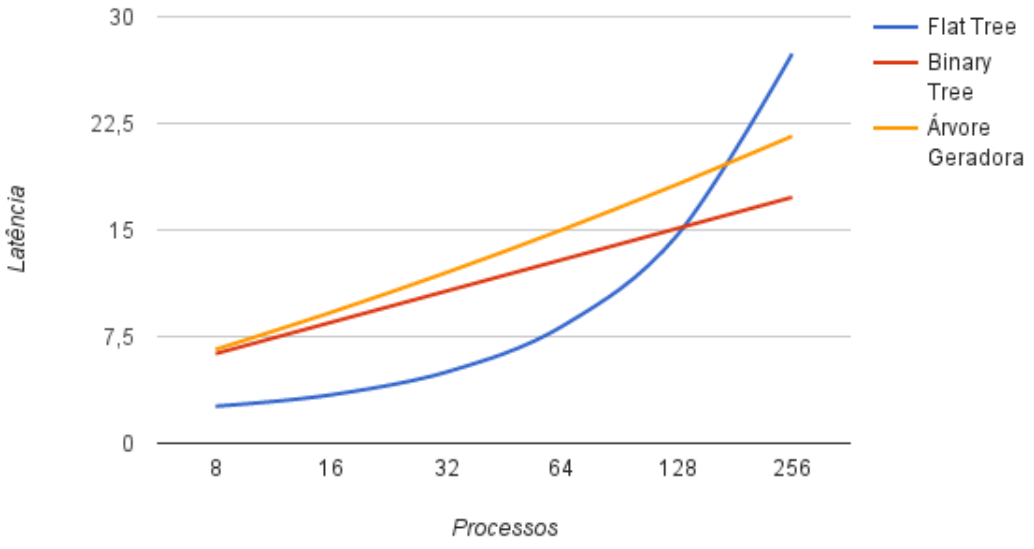


Figura 5.2: Crescimento na latência com o aumento de processos.

O gráfico contido na Figura 5.2 mostra o crescimento na latência da difusão conforme o

número de processos aumenta. Destaca-se que a *Flat Tree* inicialmente possui um resultado melhor mas ao ultrapassar 128 processos começa a ter um desempenho inferior aos demais. Isto acontece porque o tempo necessário para enviar $n-1$ cópias diretamente do processo fonte para os demais processos sobrepõe a difusão hierárquica da *Binary Tree* e da árvore geradora.

Na Tabela 5.1 observa-se o ganho em escalabilidade obtido na árvore geradora quando comparada com a *Flat Tree*. No entanto, a *Binary Tree* tem uma melhor escalabilidade que as demais árvores, isso ocorre devido a estrutura da *Binary Tree* que possui no máximo dois filhos por nó possibilitando uma melhor distribuição dos processos no sistema.

Tabela 5.1: Número máximo de envios e recebimentos no processo emissor.

Processos	**Flat Tree**	**Binary Tree**	**Árvore Geradora**
8	14	4	6
16	30	4	8
32	62	4	10
64	126	4	12
128	254	4	14
256	510	4	16

5.2 Cenários Com Uma Falha

Nesta seção foram testados diferentes cenários com falhas antes e depois do início da difusão, bem como com evento de recuperação de processo falho durante a difusão. Além disso um cenário com falha do processo fonte é considerado no último cenário.

5.2.1 Falha detectada antes da difusão da mensagem

No cenário com uma falha, o processo 4 falha no tempo 100 e uma difusão é realizada no tempo 500. Sendo assim, quando a difusão é feita todos os processos já estão cientes da falha.

Os resultados obtidos foram similares à execução sem falha, diferenciando apenas a latência que é menor. Isso ocorre devido à não necessidade de envio da mensagem para o processo 4. A latência pode ser observada na Tabela 5.2.

Tabela 5.2: Latência em unidades de tempo na difusão com uma falha.

Processos	Flat Tree	Binary Tree	Árvore Geradora
8	2,50	4,40	6,40
16	3,30	6,70	9,20
32	4,90	9,00	12,00
64	8,10	11,30	15,00
128	14,50	13,60	18,20
256	27,00	15,90	21,60

5.2.2 Falha detectada durante a difusão da mensagem

Neste cenário foi iniciado a difusão da mensagem no tempo 500 e o processo $n/2$ falha no tempo 501, provocando assim a necessidade de correção no envio da mensagem para garantir que todos os processos recebam a mesma.

Nesta simulação com resultados na Tabela 5.3 destaca-se que o tempo de envio utilizando a árvore geradora inicialmente é inferior ao tempo da árvore binaria e da *Flat Tree*. No entanto, a partir de 32 processos a latência da árvore geradora se torna inferior ao tempo da árvore binaria e a partir de 256 processos supera também o tempo da *Flat Tree*.

Tabela 5.3: Latência em unidades de tempo na difusão considerando uma falha durante o envio.

Processos	Flat Tree	Binary Tree	Árvore Geradora
8	10,10	14,10	14,20
16	10,30	15,40	16,30
32	11,10	19,20	18,30
64	16,30	24,90	20,30
128	29,00	33,10	30,10
256	40,20	36,00	32,20

5.2.3 Recuperação Durante a Difusão da Mensagem

Uma difusão é realizada pelo processo 0 no tempo 500 e o processo 4 falha no tempo 100 e retorna no tempo 501, provocando a necessidade de reenvio da mensagem.

Quando o processo 4 retorna da falha o mesmo informa os demais processos através de uma difusão de mensagem, possibilitando assim que os demais processos saibam que ele voltou e enviem as mensagens que estão em difusão para o mesmo. Sendo assim, quando 0 detecta o retorno do processo 4 no caso da *Flat Tree* e da *Binary Tree* a difusão é parada, a árvore é

reconstruída e depois a difusão é reiniciada com a nova árvore, já na árvore geradora a difusão é reiniciada apenas para o *cluster* que contem o processo 4.

Tabela 5.4: Latência em unidades de tempo na difusão considerando um retorno durante o envio.

Processos	**Flat Tree**	**Binary Tree**	**Árvore Geradora**
8	2,70	7,30	9,20
16	3,50	9,50	9,30
32	5,10	11,70	12,00
64	8,30	13,90	15,00
128	14,70	16,10	18,20
256	27,50	18,30	21,60

Assim como no cenário com falha durante a difusão, o cenário com recuperação durante a difusão também possui uma aproximação do tempo de difusão sem falhas como mostra a Tabela 5.4, isso se da devido a capacidade de envio para um único *cluster* quando utilizada a difusão com a árvore geradora.

5.2.4 Emissor Falha Durante a Difusão

Uma difusão é iniciada pelo processo 0 no tempo 500, mas o mesmo falha no tempo 501, antes de finalizar a difusão. Para garantir que todos os processos tenham recebido a mensagem enviada pelo processo 0, cada um dos demais processos realiza a difusão da ultima mensagem recebida do processo 0. Isso ocasiona em um grande aumento no número de envios e recebimentos em cada nodo. Esse aumento pode ser observado na Tabela 5.5.

Tabela 5.5: Número de envios e recebimentos de mensagens na falha do emissor.

Processos	**Flat Tree**	**Binary Tree**	**Árvore Geradora**
8	189	194	194
16	885	898	1.122
32	3.813	3.842	4.002
64	15.813	15.874	16.124
128	64.389	64.514	65.230
256	259.845	260.098	260.542

Capítulo 6

Conclusão

Um sistema distribuído é definido como um conjunto finito com processos independentes, que se comunicam usando troca de mensagens para realizar alguma tarefa. Este sistema possui três características: concorrência, inexistência de relógio global e falhas e pode ser divido em síncrono e assíncrono.

Um sistema possui os limites de tempo de processamento, envio de mensagens, taxa de variação de relógio e diferença entre relógios locais conhecidos, enquanto em um sistema assíncrono os limites temporais não são conhecidos.

Em um sistema distribuído existe um componente básico chamado *broadcast* que é responsável pela propagação de mensagens dentro do sistema. Este *broadcast* pode ser classificado como melhor-esforço ou confiável. A difusão por melhor-esforço não considera a falha no processo que inicia o envio da mensagem, diferente do *broadcast* confiável que considera falhas no emissor.

Considerando um sistema distribuído síncrono, foi proposto um algoritmo de *broadcast* que considera o modelo de falhas de *crash-recovery*.

O algoritmo proposto quando comparado com as difusões em *Flat Tree* e *Binary Tree* mostrou-se mais eficiente em casos onde a falha ocorre durante a difusão da mensagem. No caso onde o processo retorna ao sistema, o tempo de envio tende a ser o mesmo que o tempo sem falhas. Isso ocorre pois a árvore geradora consegue calcular qual é o cluster onde o processo esta falho ou retornou e reenviar a mensagem apenas para o determinado cluster, enquanto a *Flat Tree* e a *Binary Tree* precisam reenviar a mensagem para todos os processos do sistema.

No entanto, nos casos de nos casos de difusão sem falhas ou com falhas conhecidas pelos processos, o algoritmo mostrou-se melhor que a *Flat Tree* após 128 processos e inferior a árvore

binária.

6.1 Trabalhos Futuros

Como trabalhos futuros pode-se implementar a solução proposta juntamente com uma aplicação real, por exemplo, exclusão mútua, replicação de dados, entre outras.

Referências Bibliográficas

[Birman 1996]BIRMAN, K. P. Building secure and reliable network applications. Greenwich, CT, USA, 1996.

[Bonomi, Pozzo e Baldoni 2013]BONOMI, S.; POZZO, A. D.; BALDONI, R. Intrusion-tolerant reliable brodcast. 2013.

[Coulouris et al. 2013]COULOURIS, G. et al. *Sistemas distribuídos: Conceitos e Projeto*. 5. ed. [S.l.: s.n.], 2013.

[Cristian 1991]CRISTIAN, F. Understanding fault-tolerant distributed systems. *Communications of the ACM*, ACM, v. 34, n. 2, p. 56–78, 1991.

[Duarte e Nanya 1998]DUARTE, E. P.; NANYA, T. A hierarchical adaptive distributed system-level diagnosis algorithm. *Computers, IEEE Transactions on*, IEEE, v. 47, n. 1, p. 34–45, 1998.

[Fetzer e Cristian 1995]FETZER, C.; CRISTIAN, F. An optimal internal clock synchronization algorithm. 1995.

[Fischer, Lynch e Paterson 1985]FISCHER, M.; LYNCH, N. A.; PATERSON, M. Impossibility of distributed consensus with one faulty process. 1985.

[Freiling, Guerraoui e Kuznetsov 2011]FREILING, F. C.; GUERRAOUI, R.; KUZNETSOV, P. The failure detector abstraction. 2011.

[Guerraoui e Rodrigues 2006]GUERRAOUI, R.; RODRIGUES, L. Introduction to reliable distributed programming springer-verlag. Berlin, Alemanha, 2006.

[Hadzilacos e Toueg 1994]HADZILACOS, V.; TOUEG, S. A modular approach to fault-tolerant broadcasts and related problems. 1994.

[Jalote 1994]JALOTE, P. Fault tolerance in distributed systems. 1994.

[Krakowiak e Shrivastava 1999]KRAKOWIAK, S.; SHRIVASTAVA, S. K. Advances in distributed systems. advanced distributed computing: From algorithms to systems. London, UK, 1999.

[Lamport 1978]LAMPORT, L. Time, clocks and the ordering of events in a distributed system. 1978.

[Lamport e Lynch 1990]LAMPORT, L.; LYNCH, N. *Distributed computing: models and methods*. MA, USA: [s.n.], 1990.

[Lamport, Shostak e Pease 1982]LAMPORT, L.; SHOSTAK, R.; PEASE, M. The byzantine generals problem. 1982.

[Leitão, Pereira e Rodrigues 2007]LEITãO, J.; PEREIRA, J.; RODRIGUES, L. Hyparview: A membership protocol for reliable gossip-based broadcast. 2007.

[Mullender 1993]MULLENDER, S. *Distributed Systems*. New York, NY, USA: [s.n.], 1993.

[Pješivac-Grbović et al. 2007]PJEŠIVAC-GRBOVIĆ, J. et al. Performance analysis of mpi collective operations. *Cluster Computing*, Springer, v. 10, n. 2, p. 127–143, 2007.

[Raynal 2013]RAYNAL, M. Distributed algorithms for message-passing systems. Springer, v. 500, 2013.

[Rodrigues, Jr. e Arantes 2014]RODRIGUES, L.; JR., E. D.; ARANTES, L. Árvores geradoras mínimas distribuídas e autonômicas. 2014.

[Ruoso 2013]RUOSO, V. K. Uma estratégia de teste logarítmica para o algoritmo hi-adsd. 2013.

[Tanenbaum 2007]TANENBAUM, A. S. *Distributed Systems: Principles and Paradigms*. [S.l.: s.n.], 2007.

[Turek e Shasha 1992]TUREK, J.; SHASHA, D. *The Many Face of Consensus in Distributed Systems*. [S.l.: s.n.], 1992.

[Urban, Défago e Schiper 2001]URBAN, P.; DÉFAGO, X.; SCHIPER, A. Neko: A single environment to simulate and prototype distributed algorithms. In: IEEE. *Information Networking, 2001. Proceedings. 15th International Conference on*. [S.l.], 2001. p. 503–511.

[Veríssimo e Rodrigues 2001]VERíSSIMO, P.; RODRIGUES, L. Distributed systems for system architects. New York, NY, USA, 2001.

[Yang, Li e Lou 2009]YANG, Z.; LI, M.; LOU, W. R-code: network coding based reliable broadcast in wireless mesh networks with unrealible link. 2009.