Uma Solução de Difusão Confiável Hierárquica em Sistemas Distribuídos Assíncronos

Luiz A. Rodrigues[1], Denis Jeanneau[2], Elias P. Duarte Jr.[3] e Luciana Arantes[2]

[1] Colegiado de Ciência da Computação – Unioeste-Cascavel, Brasil

[2]Sorbonne Universités, UPMC, CNRS, Inria, França

[3]Departamento de Informática – UFPR, Brasil

luiz.rodrigues@unioeste.br, denis.jeanneau@lip6.fr

Abstract. *Reliable broadcast is a key communication service for the development of distributed systems, since it allows messages to be sent simultaneously from one source to all other processes in the system. This paper presents a hierarchical reliable broadcast algorithm built over a virtual topology called VCube. Messages are propagated by spanning trees that self-adapt dynamically as soon as a crash is detected by an underlying monitoring system. Processes can fail by crashing and a failure is permanent. Special messages are used to treat false suspicions occasionally generated due to the asynchronous system model. Experimental results show the efficiency of the proposed solution compared to a one-to-all strategy.*

Resumo. *Difusão confiável é um serviço fundamental de comunicação para o desenvolvimento de sistemas distribuídos, pois permite o envio simultâneo de mensagens de um processo fonte para todos os demais processos do sistema. Este trabalho apresenta uma solução hierárquica para a difusão confiável de mensagens construída sobre a topologia virtual denominada VCube. As mensagens são propagadas por árvores geradoras que se adaptam dinamicamente à medida que falhas são detectadas pelo mecanismo de monitoramento. Os processos podem falhar por parada (crash) e uma falha é permanente. Mensagens especiais são utilizadas para tratar falsas suspeitas ocasionalmente geradas em virtude do modelo de sistema assíncrono. Resultados experimentais mostram a eficiência da solução proposta comparada com uma estratégia um-para-todos.*

1. Introdução

Difusão de mensagens (*broadcast*) é um componente básico para implementar diversos algoritmos e serviços distribuídos como notificação, entrega de conteúdo, replicação e comunicação em grupo [Leitão et al. 2007, Yang et al. 2009, Bonomi et al. 2013]. Um processo em um sistema distribuído utiliza difusão para enviar uma mensagem a todos os outros processos do sistema. No entanto, se este processo falha durante o procedimento de difusão, alguns processos podem receber a mensagem enquanto outros não. A difusão de melhor-esforço garante que, se o emissor é correto, todos os processos corretos recebem a mensagem difundida por ele. Por outro lado, se o emissor pode falhar, estratégias de difusão confiável (*reliable broadcast*) precisam ser implementadas [Hadzilacos e Toueg 1993].

Algoritmos de difusão tolerante a falhas são normalmente implementados utilizando enlaces ponto-a-ponto confiáveis e primitivas SEND e RECEIVE. Os processos invocam BROADCAST(m) e DELIVER(m) para difundir e entregar uma mensagem m para/de outros processos da aplicação, respectivamente. Para incluir tolerância a falhas, um detector de falhas [Freiling et al. 2011] pode ser utilizado para notificar o algoritmo de *broadcast*, que deve reagir apropriadamente quando uma falha é detectada.

Este trabalho apresenta um algoritmo de difusão confiável no qual cada processo do sistema é alcançado por meio de uma árvore dinâmica construída e mantida sobre uma topologia baseada em hipercubo virtual chamada VCube [Duarte et al. 2014]. O sistema é representado logicamente por um grafo completo com enlaces confiáveis. No VCube, os processos são organizados em *clusters* progressivamente maiores, formando um hipercubo completo quando não há processos falhos. Em caso de falhas, os processos corretos são reconectados entre si mantendo-se as propriedades logarítmicas do hipercubo. Uma versão preliminar do algoritmo foi apresentada em [Jeanneau et al. 2016]. Este artigo apresenta o algoritmo em detalhes e inclui resultados de simulação em cenários diversos.

Além da especificação, o algoritmo proposto foi comparado com uma estratégica um-para-todos ponto-a-ponto. Os resultados confirmam a eficiência da solução de broadcast proposta considerando: (1) a latência para entregar a mensagem a todos os processos corretos; e (2) o número total de mensagens, incluindo retransmissões em caso de falhas.

O restante do texto está organizado nas seguintes seções. A Seção 2 discute os trabalhos correlatos. A Seção 3 apresenta as definições básicas, o modelo do sistema e o VCube. O algoritmo de *broadcast* confiável proposto é apresentado na Seção 4. Os resultados de simulação são apresentados na Seção 5. A Seção 6 apresenta a conclusão e os trabalhos futuros.

2. Trabalhos Relacionados

Grande parte dos algoritmos de difusão são baseados em árvores geradoras. Schneider et al. (1984) introduziram um algoritmo de difusão tolerante a falhas no qual a raiz é o processo que inicia a transmissão, ou seja, o remetente. Se um processo p que pertence à árvore falhar, outro processo assume a responsabilidade de retransmitir as mensagens que p deveria ter transmitido se estivesse correto. Os processos podem falhar por *crash* e a falha de um processo é detectada por um módulo de detecção de falhas após um intervalo de tempo finito, mas não conhecido. Um processo pode enviar uma próxima mensagem somente após a difusão anterior ter sido concluída. No entanto, os autores não descrevem como a detecção de falhas é implementada, tampouco fornecem um algoritmo para construir e reorganizar a árvore após a falha.

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

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

Em Wu (1996), os autores apresentam um algoritmo de difusão tolerante a falhas para hipercubos baseado em árvores binomiais. O algoritmo pode recursivamente regenerar uma subárvore falha, induzida por um nodo falho, através de uma das folhas da árvore. No entanto, ao contrário da abordagem proposta neste trabalho, a solução exige uma mensagem especial para indicar que a árvore deverá ser reorganizada e, neste caso, as mensagens de difusão não são tratadas pelos nodos até que a árvore seja reconstruída.

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

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

Em Rodrigues et al. (2014) foi apresentada uma solução para *broadcast* confiável utilizando árvores dinâmicas no VCube. O algoritmo permite a propagação de mensagens utilizando múltiplas árvores construídas dinamicamente a partir de cada emissor e que incluem todos os nodos do sistema. Diferente da solução proposta neste trabalho, o modelo é síncrono e o detector de falhas é perfeito.

3. Definições e Modelo do Sistema

Um sistema distribuído é composto por um conjunto finito P com $n > 1$ processos $\{p_0, .., p_{n-1}\}$ que se comunicam por troca de mensagens. Considera-se que cada processo executa uma tarefa e está alocado em um nodo distinto. Assim, os termos *nodo* e *processo* são utilizados com o mesmo sentido.

Comunicação. Os processos se comunicam através do envio e recebimento de mensagens. A rede é representada por um grafo completo, isto é, cada par de processos está conectado diretamente por enlaces ponto-a-ponto bidirecionais. No entanto, processos são organizados em uma topologia de hipercubo virtual, chamada VCube. Se não existem processos falhos, VCube é um hipercubo completo. Após uma falha, a topologia do VCube é modificada dinamicamente (mais detalhes na Seção 3.1). As operações de envio e recebimento são atômicas. Enlaces são confiáveis, garantindo que as mensagens trocadas entre dois processos nunca são perdidas, corrompidas ou duplicadas pelos enlaces.

Modelo de Falhas. O sistema admite falhas do tipo *crash* permanente. Um processo que nunca falha é considerado *correto* ou *sem-falha*. Caso contrário ele é considerado *falho*.

Detecção de falhas. O sistema é assíncrono, isto é, não existem limites conhecidos para a velocidade de processamento e atraso de transmissão de mensagens. A propriedade de completude do detector implementado pelo VCube é garantida, mas a acurácia não. Ou seja, todos os processos corretos detectam em algum momento a falha de um processo, mas falsas suspeitas podem acontecer arbitrariamente [Freiling et al. 2011].

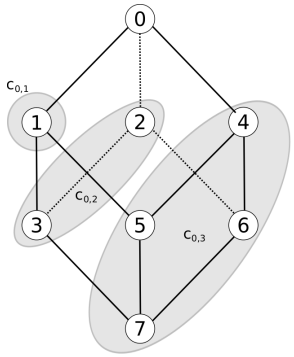
3.1. O VCube

O VCube é uma topologia baseada em hipercubo virtual criada e mantida com base nas informações de diagnóstico obtidas por meio de um sistema de monitoramento de processos descrito em Duarte Jr. et al. (2014). Cada processo que executa o VCube é capaz de testar outros processos no sistema para verificar se estão corretos ou falhos. Um processo é considerado *correto* ou *sem-falha* se a resposta ao teste for recebida corretamente dentro do intervalo de tempo esperado. Caso contrário, o processo é considerado *falho* ou suspeito. Os processos são organizados em *clusters* progressivamente maiores. Cada *cluster* $s = 1, .., \log_2 n$ possui 2^s elementos, sendo n o total de processos no sistema. Os testes são executados em rodadas. Para cada rodada um processo i testa o primeiro processo sem-falha j na lista de processos de cada *cluster* s e obtém dele as informações que ele possui sobre os demais processos do sistema.

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

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

A Figura 1 exemplifica a organização hierárquica dos processos em um hipercubo de três dimensões com $n = 2^3$ elementos. A tabela da direita apresenta os elementos de cada *cluster* $c_{i,s}$. Como exemplo, na primeira rodada o processo p_0 testa o primeiro processo no *cluster* $c_{0,1} = (1)$ e obtém informações sobre o estado dos demais processos armazenada em p_1. Em seguida, p_0 testa o processo p_2, que é primeiro processo no *cluster* $c_{0,2} = (2, 3)$. Por fim, p_0 executa testes no processo p_4 do *cluster* $c_{0,3} = (4, 5, 6, 7)$. Como cada processo executa estes procedimentos de forma concorrente, ao final da última rodada todo processo será testado ao menos uma vez por um outro processo. Isto garante que em $\log_2^2 n$ rodadas, todos os processos terão localmente a informação atualizada sobre o estado dos demais processos no sistema (latência de diagnóstico).



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

Figura 1. Organização Hierárquica do VCube de $d = 3$ dimensões com os *clusters* do processo 0 (esq.) e a tabela completa da $c_{i,s}$ (dir.)

4. O Algoritmo de Difusão Confiável Proposto

Um algoritmo de *broadcast* confiável garante que uma mensagem enviada por um processo emissor (fonte) é entregue (*delivered*) a todos os processos corretos, mesmo se o emissor falhar durante o procedimento de difusão. Para tanto, três propriedades devem ser satisfeitas [Kshemkalyani e Singhal 2008]:

- **Entrega confiável** (*validity*): se um processo correto i envia uma mensagem m, ele também entrega m em um tempo finito;
- **Integridade** (*integrity*): toda mensagem m é entregue por todos os processos corretos no máximo uma vez (não-duplicação) e somente se m foi previamente enviada por algum outro processo (não-criação);
- **Acordo** (*agreement*): se um processo correto entregou a mensagem m, então todo processo correto fará a entrega de m em um tempo finito.

4.1. Funções, Mensagens e Variáveis Locais

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

A função $cluster_i(j) = s$ calcula o identificador s do *cluster* do processo i que contém o processo j, $1 \leq s \leq d$. Por exemplo, considerando o 3-VCube da Figura 1, $cluster_0(1) = 1$, $cluster_0(2) = cluster_0(3) = 2$ e $cluster_0(4) = cluster_0(5) = cluster_0(6) = cluster_0(7) = 3$.

Seja m a mensagem de aplicação a ser transmitida de um processo emissor (fonte), para todos os outros processos no sistema. Três tipos de mensagens são utilizadas:

- $\langle TREE, m \rangle$: a mensagem de *broadcast* que carrega a mensagem de aplicação e que deve ser retransmitida sobre a árvore gerada com base na topologia do VCube;
- $\langle DELV, m \rangle$: mensagem enviada ao processo suspeito para contornar falsas suspeitas. Em caso de falsa suspeita, essa mensagem é processada pelo receptor considerado falho, mas não é retransmitida na árvore;
- $\langle ACK, m \rangle$: mensagens utilizadas para confirmar o recebimento das mensagens $\langle TREE, m \rangle$.

Para simplificar, as mensagens serão identificadas por `TREE`, `DELV` e `ACK`.

Cada mensagem m contém ainda dois parâmetros: (1) o identificador da origem, isto é, o processo que iniciou a difusão, obtido com a função $source(m)$; e (2) o *timestamp*, um contador sequencial local que identifica de forma única cada mensagem gerada em um processo, obtido pela função $ts(m)$.

As variáveis locais mantidas pelos processos são:

- $correct_i$: conjunto dos processos considerados corretos pelo processo i;
- $last_i[n]$: a última mensagem recebida de cada processo fonte. $last_i[j]$ é a última mensagens difundida por j que foi entregue por i;
- ack_set_i: o conjunto com todos os ACKs pendentes no processo i. Para cada mensagem $\langle TREE, m \rangle$ recebida pelo processo i de um processo j e retransmitida para o processo k, um elemento $\langle j, k, m \rangle$ é adicionado a este conjunto. O símbolo $\perp$ representa um elemento nulo. O asterisco é usado como curinga para selecionar ACKs no conjunto ack_set. Um elemento $\langle j, *, m \rangle$, por exemplo, representa todos os ACKs pendentes para uma mensagem m recebida pelo processo j e retransmitida para qualquer outro processo.
- $pending_i$: lista de mensagens m recebidas pelo processo i de um processo fonte $source(m)$ que ainda não podem ser entregues à aplicação por estarem fora de ordem, isto é, $ts(m) > ts(last_i(source(m)) + 1$.

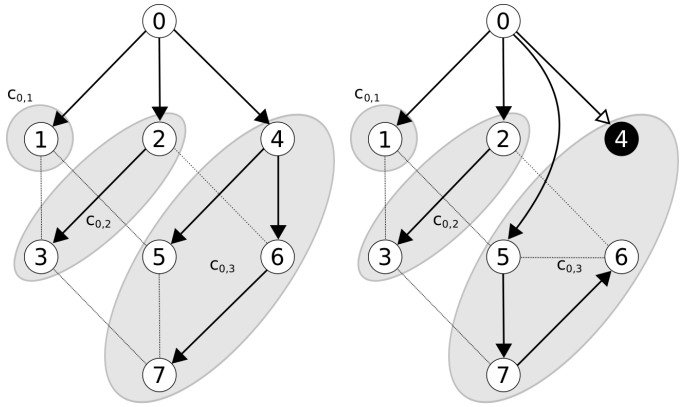
- $history_i$: histórico de mensagens que já foram retransmitidas por i. Este conjunto é utilizado para prevenir o envio duplicado de mensagens no mesmo *cluster*. $\langle j, m, h\rangle \in history_i$ indica que a mensagem m recebida do processo j já foi enviada por i para o *cluster* $c_{i,s}$ para todo $s \in [1, h]$.

4.2. Descrição do Algoritmo

O Algoritmo 1 apresenta uma solução para difusão confiável baseada em árvores geradores construídas dinamicamente com base no grafo de testes do VCube.

Considere a execução do sistema em um cenário sem processos falhos. Um processo i inicia o *broadcast* invocando a função BROADCAST(m). A linha 7 garante que um novo *broadcast* será iniciado apenas após o término do anterior, isto é, quando não há mais *acks* pendentes para a mensagem $last_i[i]$. Quando a mensagem m é entregue localmente em i (linha 9) e, pela invocação de BROADCAST_TREE (linha 10), i retransmite m para os vizinhos no VCube. Para isso, ele invoca a função BROADCAST_CLUSTER para cada *cluster* $s \in [1, log_2 n]$, que envia `TREE` para o primeiro processo k considerado correto no *cluster* (linha 27). Para cada mensagem `TREE` enviada, um *ack* é incluído na lista de *acks* pendentes (linha 28). A medida que os *acks* retornam, as pendências são removidas e os `ACK`s são propagados até a raiz.

A Figura 2(a) mostra uma execução sem falhas considerando o processo 0 (p_0) como fonte. Após fazer a entrega local, p_0 envia uma cópia da mensagem para p_1, p_2 e p_4, que são os vizinhos dele no VCube. Estes, por sua vez, retransmitem a mensagem na subárvore formada pelos *clusters* internos, isto é, aqueles contidos dentro do *cluster* ao qual estão em relação ao processo fonte i. No exemplo, p_2 retransmite para p_3; p_4 retransmite para p_5 e p_6; e p_6 retransmite para p_7.



(a) Sem falhas (b) p_4 é suspeito

Figura 2. Difusão confiável no processo 0 (p_0).

Considere agora a execução do algoritmo em um cenário com processos falhos ou suspeitos. Considere ainda que uma difusão está em andamento, mas que o processo j ainda não foi suspeito. Se o emissor i está diretamente conectado com j, irá enviar a mensagem `TREE` para j e aguardar o `ACK`. Se j está realmente falho, a sua falha será detectada em algum momento pelo VCube executando em i, que fará a notificação. Quando j é detectado como falho pelo processo i o evento de CRASH(j) é automaticamente executado. Imediatamente, todos as mensagens enviadas a j e que ainda não foram confirmadas

Algoritmo 1 Algoritmo de Difusão Confiável no Processo i

```
1:  last_i[n] ← {⊥, ..., ⊥}
2:  ack_set_i ← ∅
3:  correct_i ← {0, ..., n − 1}
4:  pending_i ← ∅
5:  history_i ← ∅

6:  procedure BROADCAST(msg m)
7:      wait until ack_set_i ∩ {⟨⊥, ∗, last_i[i]⟩} = ∅
8:      last_i[i] ← m
9:      DELIVER(m)
10:     BROADCAST_TREE(⊥, m, log_2 n)

11: procedure BROADCAST_TREE(proc j, msg m, int h)
12:     start ← 0
13:     if ∃x : ⟨j, m, x⟩ ∈ history_i then
14:         start ← x
15:         history_i ← history_i \ {⟨j, m, x⟩}
16:     history_i ← history_i ∪ {⟨j, m, max(start, h)⟩}
17:     if start < h then
18:         for all s ∈ [start + 1, h] do
19:             BROADCAST_CLUSTER(j, m, s)

20: procedure BROADCAST_CLUSTER(proc j, msg m, int s)
21:     sent ← false
22:     for all k ∈ c_{i,s} do
23:         if sent = false then
24:             if ⟨j, k, m⟩ ∈ ack_set_i
                  and k ∈ correct_i then
25:                 sent ← true
26:             else if k ∈ correct_i then
27:                 SEND(⟨TREE, m⟩) to p_k
28:                 ack_set_i ← ack_set_i ∪ {⟨j, k, m⟩}
29:                 sent ← true
30:             else if ⟨j, k, m⟩ ∉ ack_set_i then
31:                 SEND(⟨DELV, m⟩) to p_k

32: procedure CHECK_ACKS(proc j, msg m)
33:     if j ≠ ⊥ and ack_set_i ∩ {⟨j, ∗, m⟩} = ∅ then
34:         SEND(⟨ACK, m⟩) to p_j

35: procedure HANDLE_MESSAGE(proc j, msg m)
36:     pending_i ← pending_i ∪ {m}
37:     while ∃l ∈ pending_i : source(l) = source(m)
          ∧(ts(l) = ts(last_i[source(l)]) + 1
          or last_i[source(l)] = ⊥ ∧ ts(l) = 0) do
38:         last_i[source(l)] ← l
39:         pending_i ← pending_i \ {l}
40:         DELIVER(l)
41:     if source(m) ∉ correct_i then
42:         BROADCAST_TREE(j, last_i[source(m)], log_2 n)

43: upon RECEIVE ⟨TREE, m⟩ from p_j
44:     HANDLE_MESSAGE(m)
45:     BROADCAST_TREE(j, m, cluster_i(j) − 1)
46:     CHECK_ACKS(j, m)

47: upon RECEIVE ⟨DELV, m⟩ from p_j
48:     HANDLE_MESSAGE(m)

49: upon RECEIVE ⟨ACK, m⟩ from p_j
50:     for all k = x : ⟨x, j, m⟩ ∈ ack_set_i do
51:         ack_set_i ← ack_set_i \ {⟨k, j, m⟩}
52:         CHECK_ACKS(k, m)

53: upon notifying CRASH(j)
54:     correct_i ← correct_i \ {j}
55:     for all p = x, m = y :
          ⟨x, j, y⟩ ∈ ack_set_i ∩ {⟨∗, j, ∗⟩} do
56:         BROADCAST_CLUSTER(p, m, cluster_i(j))
57:         ack_set_i ← ack_set_i \ {⟨p, j, m⟩}
58:         CHECK_ACKS(p, m)
59:     if last_i[j] ≠ ⊥ then
60:         BROADCAST_TREE(j, last_i[j], log_2 n)

61: upon notifying UP(j)
62:     correct_i ← correct_i ∪ {j}
```

(aquelas registradas em ack_set_i) são retransmitidas ao próximo processo considerado correto no mesmo *cluster* de i, se existir um. Além disso, o $broadcast$ da última mensagem recebida j é refeito utilizando a raiz de i (linha 60). Isso garante que que todos os processos corretos receberão uma cópia da última mensagem de j, mesmo que o *broadcast* de j tenha sido interrompido antes do término. Note que isso é feito por todos os processos corretos a medida que a notificação da falha de j é detectada, pois não há como determinar qual processo possui a mensagem mais atual de j.

Em um segundo cenário de falhas, considere que o processo j já havia sido suspeito antes do processamento da mensagem `TREE`. No caso do emissor, uma mensagem `TREE` será enviada para o primeiro processo correto k de cada *cluster* s conforme a lista ordenada gerada pela $c_{i,s}$. Para os processos considerados falhos posicionados antes de k na lista, uma mensagem `DELV` será enviada. Em caso de falsa suspeita, `DELV` é processada, mas não é retransmitida na subárvore de j. No exemplo da Figura 2(b) p_4 está suspeito por p_0. Sendo $c_{0,3} = (4, 5, 6, 7)$, o processo fonte p_0 envia `TREE` para p_5, que é o primeiro sem falha na lista, e `DELV` para p_4. A difusão continua na subárvore de p_5 que entrega a mensagem a todos os corretos no *cluster*.

4.3. Prova de Correção

O correto funcionamento do Algoritmo 1 como uma solução de difusão confiável é garantido pelas propriedades de entrega confiável, integridade (não-duplicação e não-criação) e acordo.

Lema 1 (Entrega Confiável)**.** *O Algoritmo 1 garante que se um processo fonte correto i envia uma mensagem m por difusão, ele também entrega m em um tempo finito.*

Prova. Se um processo i realiza o *broadcast* de uma mensagem m, a única maneira de i não entregar m é se i aguardar indefinidamente na linha 7. Esta espera é interrompida quando o conjunto ack_set_i não contém mais mensagens de confirmação pendentes em relação a mensagem $last_i[i]$ difundida previamente por i.

Para todo processo j que i envia $last_i[i]$, i adiciona um *ack* pendente em ack_set_i (linha 28). Se j está correto, ele responde em um tempo finito com uma mensagem `ACK` (linha 34) e i remove $\langle \perp, j, last_i[j] \rangle$ de ack_set_i na linha 51. Se j está falho, em um tempo finito i será notificado da falha e removerá o *ack* pendente na linha 57.

Como resultado, todos os *acks* pendentes para $last_i[i]$ serão removidos do conjunto ack_set_i em um tempo finito e i realizará a entrega de m na linha 9.

Lema 2. *Para quaisquer processos i e j, o valor de $ts(last_i[j])$ é sempre incrementado ao longo do tempo.*

Prova. Por questões de simplificação, considere $ts(\perp) = -1$. O vetor $last_i$ é modificado somente nas linhas 8 e 38.

A modificação na linha 8 é feita somente quando i inicia o *broadcast* de uma nova mensagem m. Como o *timestamp* de uma nova mensagem enviada por um mesmo processo é sempre incrementado, $ts(m) > ts(last_i[i])$. Quando i invoca BROADCAST com m, $ts(last_i[i])$ será incrementado na linha 8.

Na segunda possibilidade, $last_i$ é modificado na linha 38. $last_i[source(l)]$ é atualizado com a mensagem l se $last_i[source(l)] = \perp$ e $ts(l) = 0$ (e portanto, $ts(last_i[source(l)]) = -1 < ts(l)$), ou se $ts(l) = ts(last_i[source(l)]) + 1$. Neste caso, $last_i[source(l)$ é atualizado somente se o novo valor de $ts(last_i[source(l)])$ é maior que o atual.

Lema 3 (Integridade)**.** *O Algoritmo 1 garante que para toda mensagem m é entregue por todos os processos corretos no máximo uma vez (não-duplicação) e somente se m foi previamente enviada por algum outro processo (não-criação).*

Prova. Os processos somente entregam uma mensagem se eles estão iniciando o *broadcast* (linha 9) ou se a mensagem está no conjunto $pending_i$ (linha 40). Mensagens são adicionadas no conjunto $pending_i$ somente na linha 36, depois de serem recebidas de outro processo. Considerando que os enlaces são confiáveis e que não criam mensagens, uma mensagem é recebida somente se foi previamente difundida.

Para mostrar que não há entrega duplicada, considere os seguintes casos:

- $\mathbf{source(m) = i}$. O processo i invocou BROADCAST com parâmetro m. Como provado pelo Lema 1, i entregará m na linha 9. Uma vez que BROADCAST é invocado uma única vez para cada mensagem, a única forma de i entregar m uma segunda vez é na linha 40. Como $last_i[i]$ foi atualizada com m na linha 8, o Lema 2 garante que m nunca satisfará a condição da linha 37.

- $\mathbf{source(m) \neq i}$. O processo i não é o emissor (fonte) da mensagem m e, portanto, não invocou BROADCAST(m). Neste caso, a única forma para i entregar m é na linha 40. Antes de i fazer a entrega de m pela primeira fez, ele atualiza o conjunto $last_i[source(m)]$ com m na linha 38. Novamente, a partir do Lema 2, a mensagem m jamais satisfará a condição de teste da linha 37 e, portanto, i poderá entregá-la uma única vez.

Lema 4 (Acordo)**.** *O Algoritmo 1 garante que se um processo correto entregou a mensagem m, então todo processo correto fará a entrega de m em um tempo finito.*

Prova. Seja m a mensagem de *broadcast* enviada por um processo i. Duas situações devem ser consideradas:

- **i é correto**. A prova por indução mostrará que cada processo correto recebe m.
  Como base da indução, considere um sistema com $n = 2$ processos e $P = \{i, j\}$. Neste caso, $c_{i,1} = \{j\}$. Portanto, i enviará a mensagem m para j na linha 31 se i suspeita j ou na linha 27 caso contrário. Se j está correto, ele receberá a mensagem m em um tempo finito (uma vez que os canais são confiáveis) e entregará m na linha 40. i também fará a entrega de m, por conta da propriedade de entrega confiável.
  Considere como hipótese que todo processo correto recebe m para um sistema com $n = 2^k$ processos. O passo da indução provará que a hipótese é válida para $n = 2^{k+1}$. O sistema com 2^{k+1} pode ser visto como dois subsistemas $P_1 = \{i\} \cup \bigcup_{x=1}^{k} c_{i,x}$ e $P_2 = c_{i,k+1}$ tal que $|P_1| = |P_2| = 2^k$.
  Os procedimentos BROADCAST_TREE e BROADCAST_CLUSTER garantem que para cada cluster $s \in [1, k+1]$, i enviará m para ao menos um processo em $c_{i,s}$. Seja j o primeiro processo na $c_{i,k+1}$. Se j está correto, ele receberá m em um tempo finito. Se j está falho e i já foi notificado, i enviará a mensagem de qualquer forma para os casos de falsas suspeitas (linha 31), mas a enviará também para o próximo processo na $c_{i,k+1}$ (linha 27), que, estando correto, terá a tarefa de propagar a mensagem na subárvore daquele *cluster*. i repetirá esse procedimento até enviar `TREE` para um processo não suspeito na $c_{i,k+1}$, ou até enviar `DELV` para todos os processos na $c_{i,k+1}$.
  Se j está falho e i for notificado somente após ter enviado a mensagem, o procedimento BROADCAST_CLUSTER será invocado novamente na linha 56 assim que a falha for notificada, o que garante que i enviará a mensagem para um processo não-suspeito na $c_{i,k+1}$, se existir um. Como resultado, exceto se todos os processos estejam suspeitos, ao menos um processo correto receberá m. Este processo fará então a retransmissão de m na subárvore de P_2, conforme a linha 45.
  Uma vez que um processo correto retransmite a mensagem m nos dois subsistemas P_1 e P_2, e como cada subsistema possui 2^k processos, cada processo correto em P receberá a mensagem m em um tempo finito.

- **i está falho**. Se i falha antes de enviar m para algum outro processo correto, então nenhum processo entrega m e a propriedade de acordo está garantida. Se i falha após ter enviado a mensagem a todos os vizinhos corretos em cada *cluster*, a mensagem é entregue a todos os processos corretos em cada subárvore. Por outro lado, se i falha após enviar a mensagem m para alguns processos e um processo correto j recebe m, então j será notificado pelo detector de falhas em um tempo finito. Se j detectar a falha de i antes de receber a mensagem m, quando ele

recebê-la fará um novo *broadcast*, conforme a linha 42. Se j detectar a falha de i após o recebimento de m, o mesmo iniciará um *broadcast* completo na linha 60. Se j é correto, todo processo correto receberá m em um tempo finito.

Teorema 1. *O Algoritmo 1 é uma solução para difusão confiável. Ele garante as propriedades de entrega confiável, integridade e acordo.*

Prova. As propriedades de entrega confiável, integridade e acordo são garantidas pelo Lema 1, Lema 3 e Lema 4, respectivamente.

5. Avaliação Experimental

Nesta seção são apresentados os resultados dos experimentos de simulação realizados com o algoritmo de *broadcast* proposto. Os testes estão divididos em duas partes. Primeiro são apresentados os resultados para cenários sem processos falhos e, em seguida, para os cenários com falhas.

Para comparar a solução proposta, denominada VCUBE-RB, foi implementado uma solução um-para-todos, chamada ALL-RB, na qual o processo emissor (fonte) envia uma cópia da mensagem diretamente a cada processo do grupo e aguarda pelas confirmações (`ACKs`).

Os algoritmos foram implementados utilizando o Neko [Urbán et al. 2002], um *framework* Java para simulação de algoritmos distribuídos.

5.1. Parâmetros de Simulação

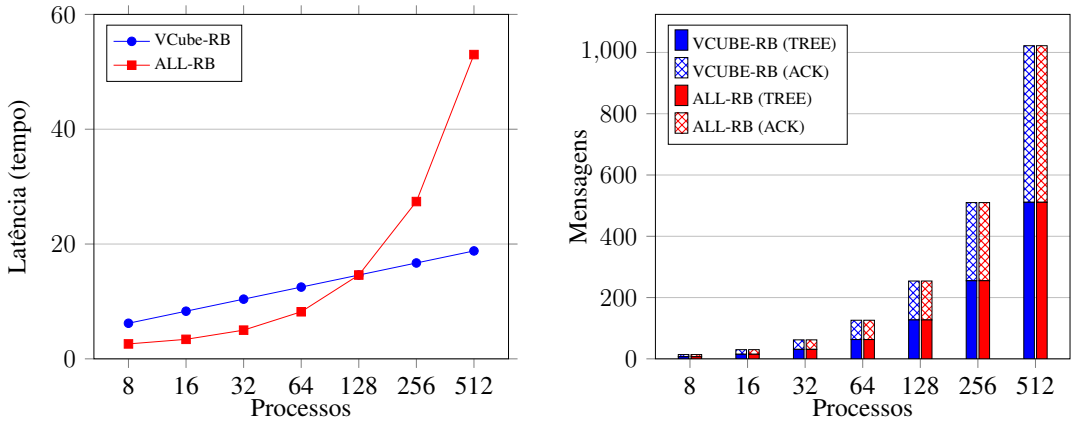
Quando um processo precisa enviar uma mensagem para mais de um destinatário ele deve utilizar primitivas SEND sequencialmente. Assim, para cada mensagem, t_s unidades de tempo são utilizadas para enviar a mensagem e t_r unidades para recebê-la, além do atraso de transmissão t_t. Estes intervalos são computados para cada cópia da mensagem enviada.

Para avaliar o desempenho de soluções de difusão, duas métricas foram utilizadas: (1) a latência para entregar a mensagem de *broadcast* a todos os processos corretos; e (2) total de mensagens enviadas pelo algoritmo, que incluem as mensagens `TREE`, `ACK` e `DELV`.

Os algoritmos propostos foram avaliados em diferentes cenários variando o número de processos e a quantidade de processos falhos. Os parâmetros de comunicação foram definidos em $t_s = t_r = 0.1$ e $t_t = 0.8$, sendo t_s o tempo de processamento no envio, t_t o tempo de transmissão e t_r o tempo de processamento no recebimento de cada mensagem. O intervalo de testes do detector foi definido em 30.0 unidades de tempo. Um processo é considerado falho se não responder ao teste após $4 * (t_s + t_r + t_t)$ unidades de tempo, isto é, 4.0.

5.2. Cenários sem Falhas

Por questões de simplificação, mas sem perda de generalidade, uma única mensagem de *broadcast* é enviada pelo processo 0 (p_0). A Figura 3 apresenta os resultados obtidos para sistemas com diferentes tamanhos. Na estratégia ALL-RB, a latência é menor para sistemas até 128 processos, mas aumenta rapidamente após este limiar em razão do atraso de processamento para o envio das cópias da mensagem `TREE` para cada um dos $n - 1$ processos corretos no sistema (lembre-se que cada mensagem enviada consome $t_s = 0.1$



(a) Latência (b) Total de Mensagens

Figura 3. Latência e total de mensagens em uma execução sem falhas.

unidades de tempo). Por outro lado, a raiz no VCUBE-RB envia $\log_2 n$ mensagens, uma para cada vizinho, sendo estas propagadas em paralelo nos demais níveis da árvore.

O total de mensagens é equivalente para VCUBE-RB e ALL-RB, conforme ilustrado na Figura 3(b), sendo igual ao dobro do número de processos no sistema menos um, isto é, $n - 1$ mensagens de aplicação (`TREE`) e $n - 1$ mensagens de confirmação (`ACK`). Como não há falhas e não foram geradas falsas suspeitas, nenhuma mensagem `DELV` é enviada pelo VCUBE-RB.

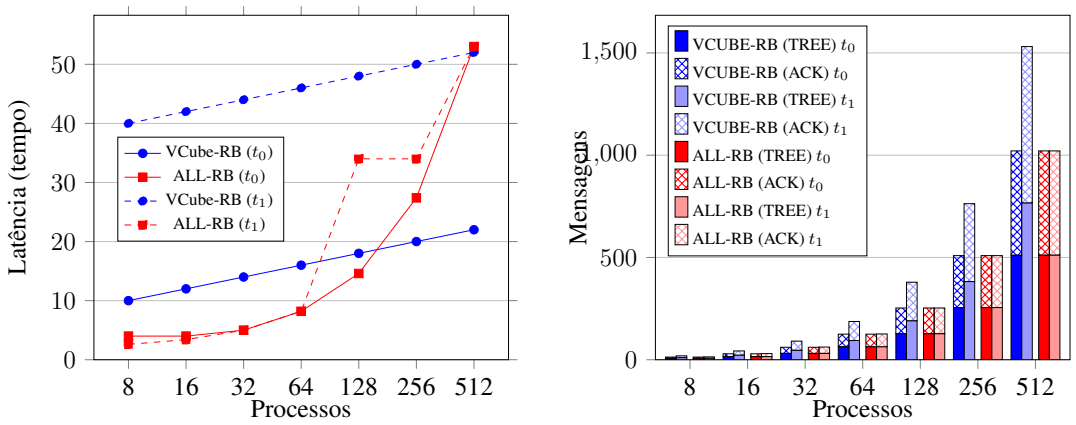
5.3. Cenários com Falhas

Os cenários com falhas foram divididos em dois casos. O primeiro considera a falha de um processo intermediário na árvore do VCUBE-RB e o segundo considera a falha do processo emissor. Por questões de simplificação, novamente um único processo emissor (p_0) realiza o *broadcast*.

Falha do Processo na Subárvore. O processo falho no nível intermediário da árvore é identificado por $p_{n/2}$. Este processo é o primeiro que recebe a mensagem no *cluster* de maior tamanho do sistema, o que pode acarretar na retransmissão de grande número de mensagens na subárvore do *cluster*. Em um VCube de 8 processos por exemplo, a falha do processo p_4, pode gerar retransmissões para p_5, p_6 e p_7, dependendo do momento em que a falha acontece. Para ALL-RB o mesmo processo foi utilizado na falha, embora o cenário seja equivalente para qualquer processo falho.

As falhas no processo $p_{n/2}$ foram simuladas em dois momentos distintos. No primeiro, a falha acontece no tempo $t_0 = 0.0$. Neste caso, nenhuma mensagem extra é gerada, visto que o processo falho não recebe a mensagem e, após a detecção da falha, uma nova mensagem será retransmitida para o próximo processo correto no *cluster*, que fará a retransmissão na subárvore de difusão uma única vez. No segundo experimento, a falha é configurada no tempo $t_1 = \log_2 n$. Este tempo é suficiente para que o processo intermediário receba e retransmita a mensagem `TREE` antes de falhar. Assim, quando a falha for detectada, uma nova mensagem será enviada para o próximo processo sem falha no *cluster* que fará uma nova difusão na subárvore daquele *cluster*. Cada processo correto no *cluster* do processo falho receberá a mensagem duas vezes. Note que a entrega é feita uma única vez em função dos *timestamps* das mensagens duplicadas.

A Figura 4 compara a latência e o número de mensagens geradas nos dois casos da falha do processo intermediário da árvore. A latência de difusão é aumentada em relação ao cenário sem falhas devido à latência de detecção do detector de falhas. Lembre-se que o *timeout* foi configurado em 4.0 unidades de tempo. No caso de ALL-RB, após a detecção da falha pelo processo fonte, a difusão é imediatamente concluída. No caso de VCUBE-RB, uma vez detectada a falha, a árvore é imediatamente reconfigurada e a difusão segue na subárvore do *cluster* do processo que falhou. Nestes cenários, além de possíveis mensagens adicionais, tem-se uma maior latência no VCUBE-RB nos sistemas menores, ainda próximos a 128. No entanto, a medida que o número de processos aumenta, a latência de ALL-RB é novamente comprometida pelo envio das múltiplas cópias da mensagem a partir do processo fonte.



(a) Latência (b) Total de Mensagens

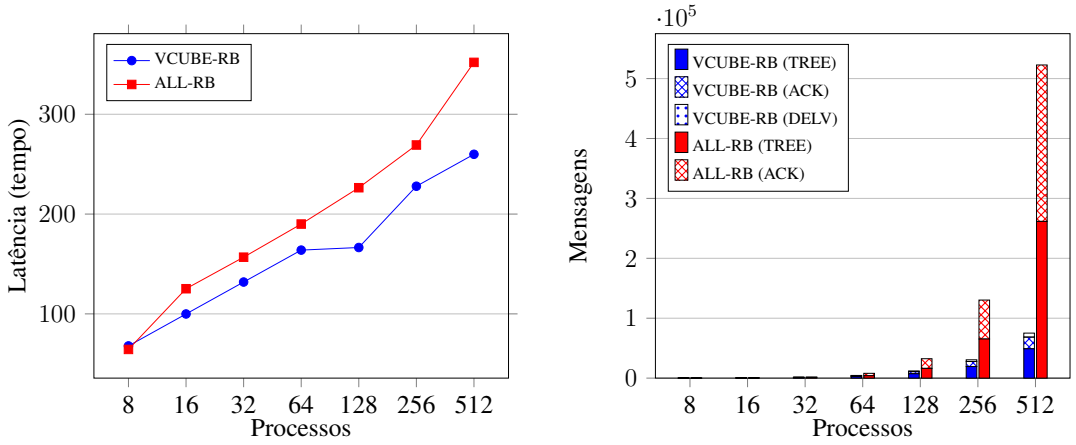
Figura 4. Latência e total de mensagens em uma execução com falha do nodo intermediário $p_{n/2}$ nos tempos $t_0 = 0.0$ e $t_1 = \log_2 n$.

Falha do Processo Emissor (Fonte). Considerando o pior caso, os cenários com falha foram comparados considerando apenas a falha do processo emissor (fonte), isto é, aquele que inicia o *broadcast* da mensagem. Para cada solução comparada, o processo zero (p_0) inicia a difusão de uma mensagem e falha logo após ter enviado uma cópia da mensagem para cada vizinho. Como cada envio consome $t_s = 0.1$, no ALL-RB a falha ocorre após o tempo $n * 0.1$ e para VCUBE-RB após $\log_2 n * 0.1$. Isso garante que todos os processos receberão uma cópia da mensagem e, portanto, terão que reiniciar o *broadcast* desta última mensagem recebida após a detecção da falha. Embora aconteçam em tempos diferentes para cada algoritmo, a falha do emissor é gerada na primeira rodada do VCube, não interferindo no tempo de detecção.

A Figura 5 apresenta os resultados obtidos considerando diferentes tamanhos de sistema. A latência difere pelo intervalo de tempo necessário para que cada processo propague a mensagem aos demais membros após serem notificados pelo VCube. Para a estratégia ALL-RB cada processo envia uma nova cópia da última mensagem de p_0 diretamente para todos os outros membros do sistema e aguarda pelas confirmações. Para a solução VCUBE-RB, cópias da última mensagem também são enviadas por cada processo do sistema, porém utilizando a árvore com raiz em cada processo.

A Figura 5(a) compara a latência das duas soluções implementadas. Embora o valor da latência seja próximo para ambos, VCUBE-RB apresenta menor latência a me-

dida que o número de processos cresce. Em relação ao total de mensagens, percebe-se na Figura 5(b) que o VCUBE-RB utilizou um número muito menor de mensagens. Este resultado é fruto do mecanismo de retransmissão na árvore, que evita que duas mensagens iguais sejam propagadas na mesma subárvore se a mesma já foi encaminhada e o ACK está pendente. Nota-se no gráfico um desequilíbrio entre o número mensagens TREE em relação as respectivas mensagens de confirmação ACK. Isto acontece em cenários com falhas porque quando um processo recebe uma mensagem TREE de um processo falho (fonte ou intermediário) ele não devolve o ACK. No caso da estratégia VCUBE-RB, por exemplo, a propagação do ACK até o processo emissor que está falho é interrompida assim que a falha é detectada por um processo no caminho reverso da árvore. Além disso, o VCUBE-RB não retransmite mensagens que já foram propagadas na árvore. Isso reduz consideravelmente o total de mensagens retransmitidas após a falha em comparação com a estratégia um-para-todos sem este controle.



(a) Latência (b) Total de mensagens

Figura 5. Latência e total de mensagens para a falha do emissor (fonte).

6. Conclusão

Este trabalho apresentou uma solução distribuída para a difusão confiável (*reliable broadcast*) em sistemas distribuídos assíncronos sujeitos a falhas de *crash*. Árvores com raiz em cada processo são construídas e mantidas dinamicamente sobre uma topologia de hipercubo virtual denominada VCube. O VCube organiza os processos em *clusters* progressivamente maiores e os conecta através de enlaces confiáveis de forma que, se não há falhas, um hipercubo completo é formado. Em caso de falhas, os processos são reorganizados de forma a manter as propriedades logarítmicas do hipercubo.

Além da prova formal do algoritmo, resultados de simulação comparando a solução proposta com uma abordagem um-para-todos mostram a eficiência do mesmo em cenários com e sem falhas, especialmente para sistemas com mais de 128 processos. O cálculo da árvore sob demanda em cada processo e sem a necessidade de troca de mensagens, somada ao controle de mensagens já transmitidas naquela árvore, diminui a latência e o total de mensagens.

Como trabalhos futuros, pretende-se implementar o algoritmo e testá-lo em uma rede real, como o PlanetLab.

Referências

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

Duarte, Jr., E. P., Bona, L. C. E. e Ruoso, V. K. (2014). VCube: A provably scalable distributed diagnosis algorithm. In: *5th Work. on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ScalA'14, pp. 17–22, Piscataway, USA. IEEE Press.

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

Freiling, F. C., Guerraoui, R. e Kuznetsov, P. (2011). The failure detector abstraction. *ACM Computing Surveys*, 43:9:1–9:40.

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

Hadzilacos, V. e Toueg, S. (1993). Fault-tolerant broadcasts and related problems. In: Distributed systems, pp. 97–145. ACM Press, New York, NY, USA, 2 ed.

Jeanneau, D., Rodrigues, L. A., Arantes, L. e Duarte, E. P. (2016). An autonomic hierarchical reliable broadcast protocol for asynchronous distributed systems with failure detector. In: *7th Latin-American Symp. Dep. Comput. (LADC)*, pp. 91–98.

Kshemkalyani, A. D. e Singhal, M. (2008). *Message ordering and group communication*, In: Distributed Computing: Principles, Algorithms, and Systems. Cambridge University Press, New York, NY, USA, 1 ed.

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

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

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

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

Rodrigues, L. A., Duarte Jr., E. P. e Arantes, L. (2014). Árvores geradoras mínimas distribuídas e autonômicas. In: *XXXII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, SBRC'14.

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

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

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

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