Um Serviço de Multicast Confiável Hierárquico com o VCube

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

[1] Colegiado de Ciência da Computação – Universidade Estadual do Oeste do Paraná
Caixa Postal 801 – 85819-110 – Cascavel – PR – Brasil

[2]Departamento de Informática – Universidade Federal do Paraná
Caixa Postal 19.081 – 81531-980 – Curitiba – PR – Brasil

[3]Sorbonne Universités, UPMC Université. Paris 06
CNRS, Inria, LIP6 – Place Jussieu, 4 – 75005 – Paris, France

luiz.rodrigues@unioeste.br, elias@inf.ufpr.br, luciana.arantes@lip6.fr

Abstract. *Multicast is a key group communication service for the development of distributed solutions, such as data replication and mutual exclusion. Messages can be simultaneously sent to a group of processes. This work presents a solution for hierarchical reliable multicast for the system formed over the virtual topology maintained by VCube. This topology is built and dynamically adapted based on the fault information obtained from an underlying monitoring system, forming a complete hypercube when there are no processes failures. Messages are propagated by using a spanning tree dynamically created on the virtual links maintained by VCube. Processes can fail by crashing and a crash is permanent. The system is reconfigured automatically after each failure, tolerating up to* $n - 1$ *fault processes. Experimental results confirm the efficiency of the proposed algorithm compared with other solutions in the literature.*

Resumo. *Multicast é um serviço fundamental de comunicação em grupo para o desenvolvimento de soluções distribuídas, tais como replicação de dados ou exclusão mútua, pois permite o envio simultâneo de mensagens a um grupo de processos. Este trabalho apresenta uma solução para a difusão seletiva confiável de mensagens (reliable multicast) hierárquico com base na topologia virtual mantida pelo VCube. A topologia é construída e adaptada dinamicamente com base nas informações de falhas obtidas de um sistema subjacente de monitoramento, formando um hipercubo completo quando não há falhas. As mensagens são propagadas por uma árvore geradora criada dinamicamente sobre os enlaces mantidos pelo VCube. Os processos podem falhar por crash e uma falha é permanente. O VCube se auto-reconfigura após cada falha, tolerando até* $n - 1$ *processos falhos. Resultados experimentais confirmam a eficiência do algoritmo proposto quando comparado com outras soluções da literatura.*

1. Introdução

Um serviço de *multicast* é essencial para a comunicação em sistemas distribuídos, pois permite o envio simultâneo de uma mensagem a todos os membros de um determinado grupo. Aplicações para *multicast* incluem, por exemplo, serviços de replicação de dados, propagação de eventos, sincronização de relógios, escrita colaborativa, jogos

distribuídos, vídeo conferência, memória compartilhada distribuída e exclusão mútua [Khazan et al. 1998, Défago et al. 2004, Sutra e Shapiro 2008].

Um processo em um sistema distribuído utiliza *multicast* para enviar uma mensagem a todos os processos pertencentes a um subgrupo do sistema [Schiper e Sandoz 1993, Défago et al. 2004]. No entanto, se o processo emissor falhar durante o procedimento de difusão, alguns processos do grupo podem receber a mensagem enquanto outros não. O *multicast* dito de melhor-esforço garante que, se o emissor é correto, todos os processos corretos do grupo receberão a mensagem difundida por ele. Por outro lado, se a falha do emissor precisa ser considerada, estratégias de *multicast* confiável (*reliable multicast*) devem ser implementadas [Hadzilacos e Toueg 1993, Popescu et al. 2007].

Algoritmos de difusão tolerante a falhas podem ser implementados utilizando enlaces ponto-a-ponto confiáveis e primitivas SEND e RECEIVE. Os processos da aplicação invocam MULTICAST(m, g) para difundir uma mensagem m a todos os processos pertencentes ao grupo g de destinatários. Cada processo em g efetua DELIVER(m) para entregar a mensagem localmente à aplicação. Um detector de falhas [Freiling et al. 2011] pode ser utilizado para notificar o algoritmo de *multicast*, que deve reagir apropriadamente quando uma falha é detectada. Muitos protocolos e algoritmos de *multicast* utilizam árvores de *multicast* para propagar as mensagens. Estas árvores geralmente têm raiz no processo emissor da mensagem e são propagadas através da árvore até alcançar todos os integrantes do grupo. O grande desafio é construir e manter estas árvores, especialmente em cenários sujeitos a falhas [Bista 2005, Lau 2006, Rahimi e Sarsar 2008, Irwin e Basu 2013].

Este trabalho apresenta um algoritmo de *multicast* confiável no qual cada elemento do grupo dos destinatários da mensagem é alcançado através de uma árvore dinâmica sobre uma topologia de hipercubo virtual chamada VCube [Ruoso 2013, Duarte et al. 2014]. O sistema é representado logicamente por um grafo completo com enlaces confiáveis. No VCube, os processos do sistema 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 para manter as propriedades logarítmicas do hipercubo.

Além da especificação, o algoritmo proposto foi comparado com outras duas soluções de *multicast*, uma utilizando uma estratégica ponto-a-ponto e outra por árvores construídas por inundação (*flooding*). Na implementação realizada, os grupos são formados de acordo com o algoritmo de quóruns majoritários proposto em Rodrigues, Duarte Jr. e Arantes (2014b) . Os quóruns são reconfigurados dinamicamente para conter a maioria dos processos corretos do sistema. Os resultados confirmam a eficiência da solução de multicast proposta considerando: (1) *vazão (throughput)*, dado pelo total de *multicasts* completados durante um intervalo de tempo; (2) a latência para entregar a mensagem a todos os processos corretos; e (3) 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 *multicast* confiável proposto é apresentado na Seção 4. A Seção 5 contém os resultados da avaliação experimental e a Seção 6 apresenta a conclusão e os trabalhos futuros.

2. Trabalhos Correlatos

A exemplo das soluções de *broadcast*, a maior parte das soluções de *multicast* é baseada em árvores geradoras (*spanning trees*). O problema de encontrar uma árvore mínima que conecta todos os elementos de um grupo de *multicast* é conhecido como problema da árvore de Steiner [Kshemkalyani e Singhal 2008]. O problema consiste em, dado um grafo ponderado $G = (V, E)$ composto por V vértices e E arestas e um subconjunto $V' \subseteq V$, calcular $E' \subseteq E$ tal que $T = (V', E')$ é um subgrafo de G que conecta todos os elementos de V' e T é uma árvore. Uma solução simples aproximada é construir um subgrafo de G com V' vértices e as arestas de menor peso que conectam os vértices de V' e em seguida calcular a árvore mínima deste subgrafo.

O trabalho de Bista (2005) propõe uma solução tolerante a falhas para *multicast* que mantém rotas de menor custo espalhadas entre os nodos do sistema. Cada nodo do grupo calcula o menor caminho até a raiz da árvore de *multicast* evitando passar pelo antecessor na árvore atual. Quando a comunicação com o antecessor falha, a árvore é reconstruída utilizando as informações previamente armazenadas.

Em Rodrigues, Duarte Jr. e Arantes (2014a) foi apresentado 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.

RPF (*Reverse Path Forwarding*) é uma técnica implementada em alguns protocolos para encaminhamento de pacotes *multicast* em redes IP. No primeiro passo, o nodo fonte i envia uma cópia da mensagem para cada enlace de saída. Cada nodo j que recebe a mensagem verifica, com base nas informações de sua tabela de roteamento, se ele é o "melhor" caminho para i (caminho reverso). Se j é o melhor caminho para i, ele propaga a mensagem para todos os enlaces, exceto para aquele do qual recebeu a mensagem anteriormente. Caso contrário, a mensagem é ignorada. Diferente da proposta apresentada neste trabalho, trata-se de uma estratégia de inundação controlada na qual todos os nodos recebem uma cópia da mensagem, formando uma possível árvore geradora [Bolton e Lowe 2004].

3. Definições e Modelo do Sistema

3.1. Definições

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

3.2. Modelo do Sistema

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 3.3). 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. Com o objetivo de permitir a aplicação do *multicast* por algoritmos distribuídos de exclusão mútua, que não admitem falsas suspeitas de falhas, considera-se que o VCube implementa um detector de falhas perfeito em um sistema síncrono. Nenhum processo é detectado como falho antes do início da falha e todo processo falho é detectado por todos os processos corretos em um tempo finito [Freiling et al. 2011].

3.3. 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 Ruoso (2013). 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*. 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):

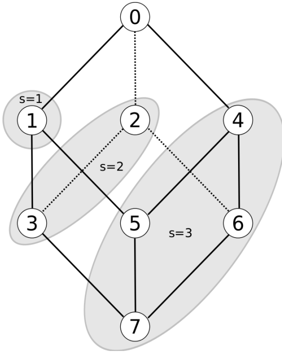
$$c_{i,s} = (i \oplus 2^{s-1}, c_{i \oplus 2^{s-1},1}, ..., c_{i \oplus 2^{s-1},s-1}) \quad (1)$$

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).

4. O Algoritmo de Multicast Proposto

Um algoritmo de *multicast* confiável garante que uma mensagem enviada por um processo emissor (fonte) para um conjunto de destinatários g é entregue (*deliver*) a todos os processos corretos de g, 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 $i \in g$ envia uma mensagem m, ele também entrega m em um tempo finito;



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.)

- **Integridade** (*integrity*): para toda mensagem m enviada por um processo $i \in g$, todo processo correto em g entrega m 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 em g entregou a mensagem m, então todo processo correto em g fará a entrega em um tempo finito.

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 *multicast* 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 $cluster_0(4) = cluster_0(5) = cluster_0(6) = cluster_0(7) = 3$.

A função $FF_neighbor_i(s) = j$ identifica o primeiro processo sem-falha j no *cluster* s do processo i ($j \in correct_i$). Por exemplo, considerando a tabela da Figura 1, $FF_neighbor_0(1) = 1$, $FF_neighbor_0(2) = 2$ e $FF_neighbor_0(3) = 4$. Em caso de falha do processo 2, $FF_neighbor_0(2) = 3$.

4.1. Descrição do Algoritmo

O Algoritmo 1 apresenta o pseudocódigo do algoritmo de *multicast* confiável proposto. Dois tipos de mensagens são utilizados: $\langle TREE, m, g\rangle$ para identificar a mensagem de aplicação m que está sendo propagada para o grupo g; e $\langle ACK, m\rangle$ para confirmar o recebimento de m pelo destinatário. Cada mensagem m contém ainda dois parâmetros: (1) o identificador da origem, isto é, o processo que iniciou a difusão, obtido com a função $source(m)$, e (2) o *timestamp*, um contador sequencial local que identifica de forma única cada mensagem gerada em um emissor, obtido pela função $ts(m)$. Os destinatários da mensagem podem ainda ser obtidos com a função $dest(m)$. Um processo obtém as informações sobre o estado dos demais processos pelo algoritmo VCube.

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;

Algoritmo 1 *Multicast* confiável no processo i

1: $last_i[n] \leftarrow \{\bot, .., \bot\}$ ▷ Inicialização
2: $ack_set_i = \emptyset$
3: $correct_i = \{0, .., n-1\}$

4: **procedure** MULTICAST(mensagem m, grupo g)
5: **if** $source(m) = i$ **then**
6: **wait until** $ack_set_i \cap \{\langle i, *, last_i[i] \rangle\} = \emptyset$
7: $last_i[i] = m$
8: DELIVER(m)
/* enviar a todos os vizinhos corretos que pertencem ao cluster s que contém elementos corretos de g*/
9: **for all** $s = 1, .., \log_2 n$ **do**
10: **if** $\{c_{i,s} \cap g \cap correct_i\} \neq \emptyset$ **then**
11: $j \leftarrow FF_neighbor_i(s)$
12: $ack_set_i \leftarrow ack_set_i \cup \{\langle i, j, m \rangle\}$
13: SEND($\langle TREE, m, g \rangle$) **to** p_j

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

18: **procedure** RECEIVE($\langle TREE, m, g \rangle$) **from** p_j
19: **if** $i \in g$ **then**
20: **if** $last_i[source(m)] = \bot$ **or**
21: $ts(m) > ts(last_i[source(m)]$ **then**
22: $last_i[source(m)] \leftarrow m$
23: DELIVER(m)
24: **if** $source(m) \notin correct_i$ **then**
25: MULTICAST(m, g)
26: **return**
/* retransmitir aos vizinhos corretos na árvore de $source(m)$ que pertencem ao cluster s que contém elementos corretos de g) */
27: **for all** $s = 1, .., cluster_i(j) - 1$ **do**
28: **if** $\{c_{i,s} \cap g \cap correct_i\} \neq \emptyset$ **then**
29: $k \leftarrow FF_neighbor_i(s)$
30: **if** $\langle j, k, m \rangle \notin ack_set_i$ **then**
31: $ack_set_i \leftarrow ack_set_i \cup \{\langle j, k, m \rangle\}$
32: SEND($\langle TREE, m, g \rangle$) **to** p_k
33: CHECKACKS(j, m)

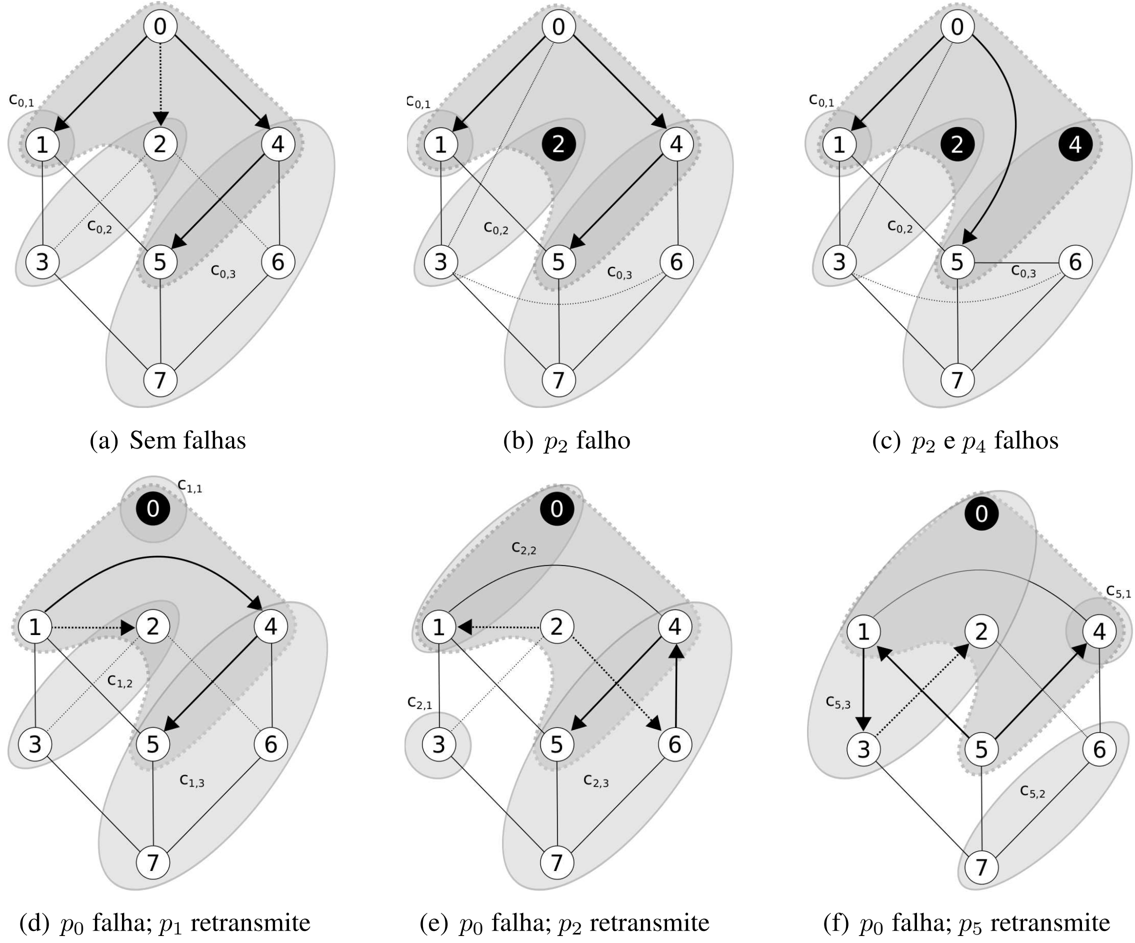
34: **procedure** RECEIVE($\langle ACK, m \rangle$) **from** p_j
35: $k \leftarrow x : \langle x, j, m \rangle \in ack_set_i$
36: $ack_set_i \leftarrow ack_set_i \setminus \{\langle k, j, m \rangle\}$
37: **if** $k \neq i$ **then**
38: CHECKACKS(k, m)

39: **procedure** CRASH(processo j) //j é detectado falho
40: $correct_i \leftarrow correct_i \setminus \{j\}$
41: **for all** $p = x, q = y, m = z : \langle x, y, z \rangle \in ack_set_i$ **do**
42: **if** $p \notin correct_i$ **then**
▷ remover ACKs pendentes para $\langle j, *, * \rangle$
43: $ack_set_i \leftarrow ack_set_i \setminus \langle p, q, m \rangle$
44: **else if** $q = j$ **then**
▷ enviar m para vizinho k, se k existir
45: $s = cluster_i(j)$
46: **if** $\{c_{i,s} \cap dest(m) \cap correct_i\} \neq \emptyset$ **then**
47: $k \leftarrow FF_neighbor_i(s)$
48: **if** $\langle p, k, m \rangle \notin ack_set_i$ **then**
49: $ack_set_i \leftarrow ack_set_i \cup \{\langle p, k, m \rangle\}$
50: SEND($\langle TREE, m, dest(m) \rangle$) **to** p_k
51: $ack_set_i \leftarrow ack_set_i \setminus \langle p, j, m \rangle$
52: CHECKACKS(p, m)
/* garantir a entrega da mensagem mais atual a todos os processos corretos em $dest(m)$ da última mensagem recebida de j */
53: **if** $last_i[j] \neq \bot$ **and** $i \in dest(last_i[j])$ **then**
54: MULTICAST($last_i[j], dest(last_i[j])$)

- ack_set_i: Para cada mensagem $\langle TREE, m, g \rangle$ 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. Quando i recebe a confirmação de entrega de m do processo j, remove a tupla do conjunto.

O símbolo $\bot$ 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.

Um processo i inicia o *multicast* invocando o método MULTICAST(m, g). A linha 6 garante que um novo *multicast* só é iniciado após o término do anterior, isto é, quando não há mais *acks* pendentes para a mensagem $last_i[i]$. Nas linhas 8 - 13 a nova mensagem m é entregue localmente (propriedade de entrega confiável) e em seguida enviada a todos os vizinhos considerados corretos no VCube que pertencem ao conjunto dos destinatários g ou que estão em *clusters* que possuem algum elemento correto contido em g. Para cada mensagem enviada, um *ack* é incluído na lista de *acks* pendentes. A



(a) Sem falhas (b) p_2 falho (c) p_2 e p_4 falhos

(d) p_0 falha; p_1 retransmite (e) p_0 falha; p_2 retransmite (f) p_0 falha; p_5 retransmite

Figura 2. ***Multicast* no processo 0 (p_0) com destinatários** $g = \{0, 1, 2, 4, 5\}$

Figura 2 ilustra exemplos para um VCube de 3 dimensões. A Figura 2(a) mostra uma execução sem falhas considerando o processo 0 (p_0) como fonte e $g = \{0, 1, 2, 4, 5\}$. 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 em cada *cluster* do VCube e que também estão em g. Embora p_5 também pertença ao mesmo *cluster* de p_4, p_0 não envia uma cópia da mensagem para p_5, pois não há aresta no VCube conectando os dois processos.

Quando um processo i recebe uma mensagem $\langle TREE, m, g\rangle$ de um processo j (linha 18) ele verifica se ele pertence ao conjunto dos destinatários g. Se $i \in g$, i determina se a mensagem é nova comparando os *timestamps* da última mensagem armazenada em $last_i[j]$ e da mensagem recebida m (linha 21), garantindo a propriedade de integridade. Se m é um nova mensagem, $last_i[j]$ é atualizado e a mensagem é entregue à aplicação. Em seguida, o processo i verifica se o processo origem da mensagem está falho. Se a origem ainda é considerada correta, m é retransmitida para os vizinhos em cada *cluster* interno ao *cluster* de i que também fazem parte do grupo com os destinatários da mensagem. A Figura 2(a) ilustra a propagação feita por p_4 para p_5. Se i é uma folha da árvore (*clusters* $s = 1$) ou se não existe vizinho correto pertencente aos destinatários, nenhum *ack* pendente é adicionado ao conjunto ack_set_i e CHECKACKS envia uma mensagem $\texttt{ACK}$ para j. Estes são os casos de p_1 e de p_2, respectivamente, representados na Figura 2(b).

Por outro lado, se um processo $i \in g$ recebe uma mensagem nova $\langle TREE, m, g\rangle$, mas verifica que $source(m)$ foi detectado como falho, o processo de *multicast* é reiniciado considerando a árvore com raiz em i. Este cenário está ilustrado pelas Figuras 2(d), 2(e) e 2(f), nas quais p_1, p_2 e p_5 executam o *multicast* da última mensagem enviada por p_0. Na Figura 2(e), por exemplo, é possível notar que p_6, embora não pertença aos destinatários, é utilizado como ponte para alcançar p_4. Isto acontece, porque na árvore de p_2 não há ligação direta com o processo p_4 e p_6 é o primeiro processo sem falha no *cluster* $c_{2,3}$ de p_2. Procedimento semelhante está representado na Figura 2(f), quando p_5 efetua o *multicast* após a falha de p_0 e a mensagem passa por p_3 antes de chegar em p_2. Nestes dois casos, p_6 e p_2 recebem a mensagem, mas não a entregam à aplicação.

Quando uma mensagem $\langle ACK, m\rangle$ é recebida, o conjunto ack_set_i é atualizado e, se não existem mais *acks* pendentes para a mensagem m, CHECKACKS envia um $\langle ACK, m\rangle$ para o processo k do qual i recebeu a mensagem `TREE` anteriormente. No entanto, se $k = i$, a mensagem `ACK` alcançou o processo fonte ou o processo que retransmitiu a mensagem após a falha do processo fonte. Nesse caso, a mensagem de `ACK` não precisa mais ser propagada.

A detecção de um processo falho j é tratada no procedimento CRASH(j). Três ações são realizadas: (1) atualização da lista de processos corretos; (2) remoção dos *acks* pendentes que contém o processo j como destino ou aqueles em que a mensagem m foi originada em j; (3) reenvio das mensagens anteriormente transmitidas ao processo j para o novo vizinho k no mesmo *cluster* de j, se existir um, isto é, se há elemento naquele *cluster* que pertence ao conjunto $dest(m)$. Esta retransmissão desencadeia uma propagação na nova estrutura da árvore. As Figuras 2(b) e 2(c) apresentam os cenários de falha para os processos p_2 e p_4. Na primeira, p_2 falha, mas não existe outro processo sem falha do grupo no *cluster* dele. No entanto, após a falha de p_4 (Figura 2(c)), p_0 retransmite a mensagem para o processo p_5, visto que $c_{0,3} = (4, 5, 6, 7)$, p_5 é o próximo processo sem falha no *cluster* $s = 3$ e p_5 pertence ao grupo. A propagação termina, pois o *cluster* de p_5 não contém outros processos no grupo.

4.2. Prova de Correção

O correto funcionamento do Algoritmo 1 como uma solução *multicast* confiável (Teorema 2) é garantido pelas propriedades de entrega confiável (Lema 2), integridade (não-duplicação e não-criação) (Lema 3) e acordo (Lema 4).

Em Rodrigues, Duarte Jr. e Arantes (2014a) foi demonstrado que a árvore geradora formada pela propagação das mensagens nos *clusters* do VCube garante a entrega das mensagens a todos os processos corretos do sistema. O Teorema 1 demonstra esse resultado, adaptando-o para grupos de processos do VCube.

Teorema 1 (Árvore Geradora). *Toda mensagem m enviada por um processo i é retransmitida a todos os processos corretos j que pertencem ao grupo multicast $g = dest(m)$ de i através de uma árvore geradora com raiz em i.*

Prova. Considere um sistema P com n processos. Se o conjunto dos destinatários g contém todos os processos do sistema ($|g| = n$), a árvore proposta em Rodrigues, Duarte Jr. e Arantes (2004a) garante a entrega a todos os processos corretos do sistema.

Por outro lado, se $g \subset P$, o algoritmo garante que uma cópia da mensagem é

enviada para cada *cluster* que contém um elemento correto $j \in g$. A propagação continua até que não haja mais processos corretos que pertençam a g nos *clusters* internos. ■

Lema 1 (Coerência). *Seja m uma mensagem enviada para o grupo multicast g de um processo i. Nenhum processo $j \notin g$ entrega m, mesmo que j faça parte da árvore geradora de i.*

Prova. O Teorema 1 garante que uma árvore geradora é formada a partir de um emissor i contendo todos os processos no grupo *multicast* g de i. No entanto, é possível que um processo $j \notin g$ façam parte da árvore geradora de i quando existe um destinatário $k \in g$ no mesmo *cluster* s de j e j é o primeiro processo sem-falha em s (Figura 2(d)). Nestes casos, quando j recebe m, ele verifica se pertence ao grupo antes de efetuar a entrega (linha 19). Se $j \in g$, ele verifica se m é nova e entrega a mensagem à aplicação (*deliver*). Caso contrário, faz apenas a retransmissão para os processos subsequentes na árvore. ■

Lema 2 (Entrega confiável). *Se um processo correto i efetua o multicast de uma mensagem m, então ele também entrega m em um intervalo de tempo finito.*

Prova. Considere a função MULTICAST(m, g) do Algoritmo 1. O emissor i aguarda até receber todos as mensagens de confirmação (ACK) relacionadas à última mensagem enviada por ele, armazenada em $last_i[i]$ (linha 6). Para garantir a entrega confiável, m é entregue para o próprio i e o valor de $last_i[i]$ é atualizado (linhas 7 e 8). ■

Lema 3 (Integridade). *Todo processo correto i no grupo multicast $g = dest(m)$ composto pelo conjunto dos destinatários da mensagem m entrega m no máximo uma vez (não-duplicação) e, se e somente se, m foi previamente enviada por multicast por algum processo (não-criação).*

Prova. A entrega da mensagem m pelo processo fonte i é garantida pelo Lema 2. No Teorema 1 é garantido que todo processo correto j no grupo $g = dest(m)$ recebe a mensagem m propagada por i através da árvore geradora. Ao receber m, j verifica se m é uma nova mensagem através do seu *timestamp* (linha 21) e, em caso positivo, entrega m. Assim, mesmo que uma mensagem seja retransmitida após a detecção de um falha e alcance um processo que já a recebeu, o receptor nunca fará a entrega duplicada. A propriedade de não-criação é garantida pela característica dos enlaces confiáveis. ■

Lema 4 (Acordo). *Se um processo fonte i envia uma mensagem m para ao menos um processo correto j no conjunto dos destinatários $g = dest(m)$ e j entrega m, então todo processo correto em g recebe e entrega m, mesmo que i venha a falhar antes de terminar o procedimento de multicast.*

Prova. Se um processo fonte é correto, o Teorema 1 em conjunto com as propriedades de integridade (Lema 3) e entrega confiável (Lema 2) garante que m será entregue por todos os processos corretos no grupo $g = dest(m)$.

Se o processo fonte i falha antes de enviar a mensagem m para algum processo correto j em cada *cluster* $s = 1, .., \log_2 n$, nenhum processo correto receberá m e a propriedade de acordo está mantida. Por outro lado, se ao menos um processo correto j receber m antes de i falhar, este será notificado sobre a falha do processo fonte i. Neste caso, o acordo é garantido por duas ações: (1) difusão durante a recepção da última mensagem recebida de i (neste caso uma mensagem nova) se i já foi detectado como falho (linha 25); (2) difusão da última mensagem de um processo j detectado como falho e removido de

$correct_i$ (linha 54). Nos dois casos, o processo que recebe a mensagem m assume a responsabilidade de garantir a entrega de m para todos os demais processos corretos em $g = dest(m)$ com origem em $source(m)$. No caso da segunda ação, todos os processos pertencentes a g executam a difusão após a detecção da falha de j, visto que não se sabe qual deles possui a mensagem mais atual. ■

Teorema 2. *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 2, Lema 3 e Lema 4, respectivamente. ■

Teorema 3. *Seja $|g_i| <= n - f$ o tamanho do grupo multicast de um processo i em um sistema com n processos dos quais f estão falhos. Para cada mensagem de multicast enviada pela aplicação, o total de mensagens enviadas pelo Algoritmo 1 é, no mínimo, $2|g_i|$ e, no máximo, $|g_i|^2 + 1$.*

Prova. Em uma execução sem falhas, para cada mensagem `TREE` enviada, uma mensagem `ACK` é retornada. Sendo $|g_i|$ o tamanho do grupo *multicast* do processo i, o total de mensagens `TREE`+`ACK` é o dobro de $|g_i|$.

Em caso de falhas, o pior caso é dado pela falha do processo fonte i, que exige a retransmissão da última mensagem recebida por cada processo no grupo de i (linhas 25 e 54). Cada processo $j \in g_i, j \neq i$ iniciará um novo *multicast* se tornando a raiz da árvore de i. Assim, além das $2|g_i|$ mensagens possivelmente geradas no *multicast* iniciado por i (no pior caso i envia a mensagem para todos os seus vizinhos antes de falhar), outras $2(|g_i| - 1)$ mensagens serão geradas por cada um dos $|g_i| - 1$ processos j, totalizando $2|g_i| + (|g_i| - 1)^2 = |g_i|^2 + 1$ mensagens. ■

5. Avaliação Experimental

Nesta seção são apresentados os resultados dos experimentos de simulação realizados com o algoritmo de *multicast* utilizando grupos formados de acordo com a solução de quóruns proposta por Rodrigues, Duarte Jr. e Arantes (2014b) , descrita a seguir. 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 de *multicast* proposta, denominada ATREE (*autonomic tree*), outras duas abordagens foram utilizadas. Na primeira, denominada ALL, o processo emissor (fonte) envia uma cópia da mensagem diretamente a cada processo do grupo e aguarda pelas confirmações (ACKS). A segunda abordagem utiliza árvores criadas inicialmente por inundação *flooding* sobre a topologia do hipercubo. Quando um processo recebe uma nova mensagem, envia um ACK e se acopla à árvore. Se o processo já está na árvore, envia um NACK. A árvore é composta por todos os processos corretos do sistema, mas somente aqueles que pertencem ao grupo entregam a mensagem à aplicação. Após cada falha a árvore precisa ser reconstruída a partir da raiz. Por esse motivo essa estratégia foi denominada NA-TREE (*non-autonomic tree*).

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

5.1. O Sistema de Quóruns do VCube

Seja $P = \{p_0, .., p_{n-1}\}$ um sistema distribuído com $n \geq 1$ processos e $Q = \{q_1, .., q_m\}$ um conjunto de subgrupos de processos de P. Q é um sistema de quóruns em P se todo par (q_i, q_j) possui uma intersecção não vazia.

O algoritmo de quóruns baseado no VCube [Rodrigues et al. 2014b] utiliza a organização virtual e a função $c_{i,s}$ apresentados na Seção 3.3. O estado dos processos considerados corretos por um processo i é informado pelo próprio VCube acoplado à i e armazenado no conjunto $correct_i$. Em termos gerais, cada processo i constrói o próprio quórum adicionando a si mesmo e a metade absoluta dos elementos j que ele considera sem-falha ($j \in correct_i$) em cada *cluster* $c_{i,s}$ do VCube. Assim, em média cada quórum possui $\lceil (n/2) + 1 \rceil$ elementos.

Como exemplo, considere o cenário sem falhas representado pelo hipercubo de três dimensões da Figura 1. O quórum q_0 calculado pelo processo p_0 é composto por ele mesmo e pela metade absoluta dos *clusters* $c_{0,1} = (1)$, $c_{0,2} = (2, 3)$ e $c_{0,3} = (4, 5, 6, 7)$. Logo, $q_0 = \{0, 1, 2, 4, 5\}$.

5.2. 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, três métricas foram utilizadas: (1) vazão (*throughput*), dado pelo total de chamadas de *multicast* completadas durante um intervalo de tempo; (2) a latência para entregar a mensagem de *multicast* a todos os processos corretos do grupo; e (3) total de mensagens enviadas pelo algoritmo, que incluem as mensagens TREE e ACK.

Os algoritmos propostos foram avaliados em diferentes cenários variando o número de processos e a quantidade de processos falhos. Os parâmetros de comunicação foram definidos em $t_s = t_r = 0,1$ e $t_t = 0,8$. O intervalo de testes do detector foi definido em $5,0$ unidades de tempo. Um processo é considerado falho se não responder ao teste após $4 * (t_s + t_r + t_t)$ unidades de tempo.

5.3. Cenários sem Falhas

Considere um cenário sem falhas para a propagação das mensagens de *multicast* nos quóruns do VCube. Cada mensagem deve ser entregue à $\lceil (n/2) + 1 \rceil$ processos, já incluindo o emissor. Nos testes sem falhas uma única mensagem é enviada pelo processo 0 (p_0). Para a estratégia NATREE é considerado a construção apenas da árvore com raiz em p_0.

A Figura 3 apresenta os resultados obtidos para sistemas com diferentes tamanhos. A latência nas soluções ATREE e NATREE é muito próxima (Figura 3(a)), assim como a vazão (Figura 3(b)). Na estratégia ALL, a latência é menor para sistemas até 256 processos, mas aumenta rapidamente após este limiar em razão do atraso de processamento para o envio as $n - 1$ mensagens na estratégia ALL. O número de mensagens é equivalente para ATREE e ALL (Figura 3(c)), mas é muito maior em NATREE, visto que a primeira mensagem é propagada por inundação (Figura 3(d)). Para as abordagem ATREE e ALL,

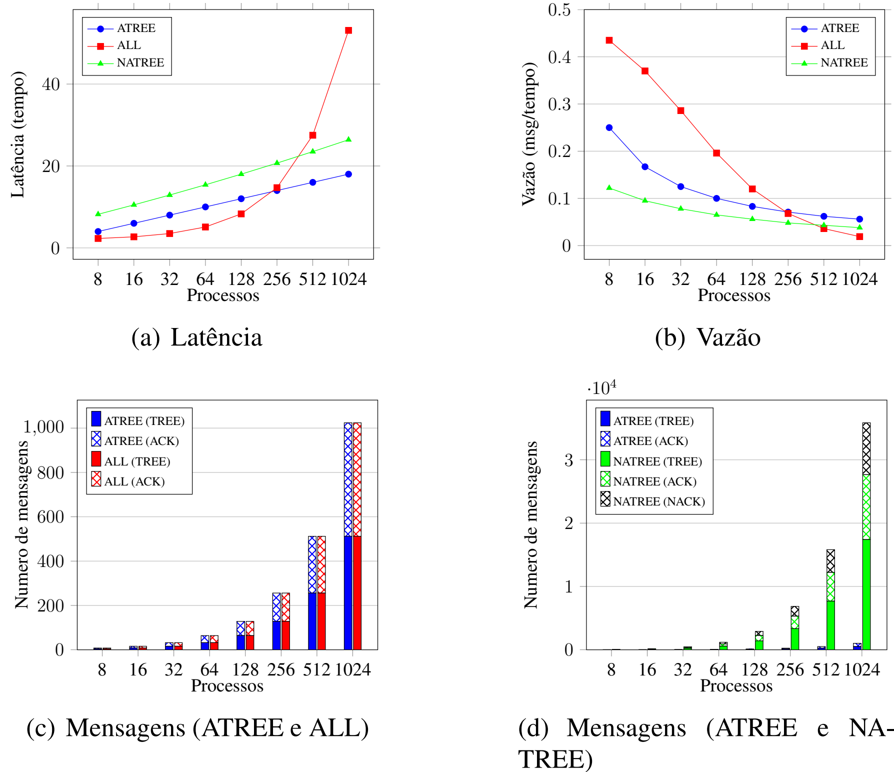


Figura 3. Latência, Vazão e total de mensagens em uma execução sem-falhas.

o total de mensagens enviadas é igual ao dobro do tamanho do grupo (o quórum neste caso, com $n/2+1$ elementos). Para 1024 processos, por exemplo, NATREE envia 35846 mensagens (17411 mensagens de aplicação `TREE`, 10241 mensagens de confirmação `ACK` e 8194 confirmações negativas `NACK`), ao passo que as outras duas soluções enviam apenas 1024 mensagens (512 `TREE` e 512 `ACK`). Após a primeira mensagem, o número de mensagens de NATREE é proporcional ao número de processos do sistema, embora para cada nova falha a árvore tenha que ser reconstruída por inundação.

5.4. Cenários com Falhas

Considerando o pior caso, os cenários com falha foram comparados considerando apenas a falha do processo emissor (fonte), isto é, aquele que inicia o *multicast* da mensagem. Para cada solução comparada, o processo 0 inicia a difusão de uma mensagem e falha logo após ter enviado uma cópia da mensagem para cada vizinho. Isso garante que todos os processos do quórum de p_0 receberão uma cópia da mensagem e, portanto, terão que reiniciar o *multicast* desta última mensagem recebida. Em função da necessidade de recriar múltiplas árvores a cada falha, cada uma com raiz em um dos processos integrantes do quórum de p_0, a estratégia NATREE não foi comparada por apresentar latência e total de mensagens muito superior as duas outras soluções.

A Figura 4 apresenta os resultados obtidos para as estratégias ATREE e ALL considerando diferentes tamanhos de sistema. O instante em que p_0 falha e a latência de detecção do VCube são os mesmos para os dois algoritmos. Assim, a latência difere pelo intervalo de tempo necessário para que cada processo no quórum propague a mensagem aos demais membros após serem notificados pelo VCube. Para a estratégia ALL cada

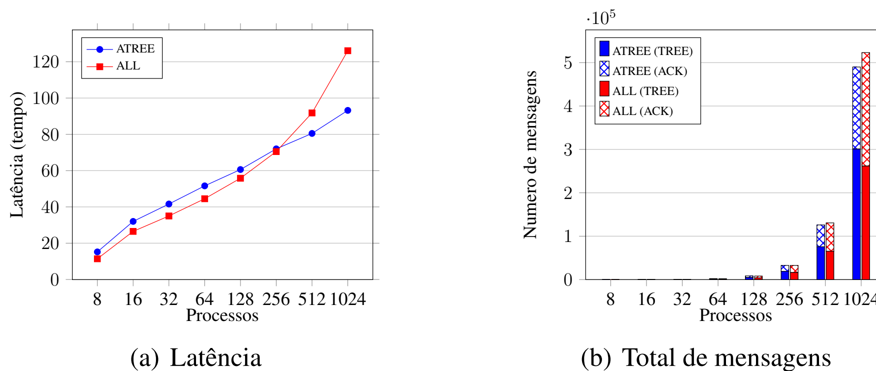


Figura 4. Latência, Vazão e total de mensagens para a falha do emissor (fonte).

processo do quórum de p_0 envia uma nova cópia da última mensagem de p_0 diretamente para todos os outros membros do quórum e aguarda pelas confirmações. Para a solução ATREE, cópias da última mensagem também são enviadas por cada membro do quórum de p_0, porém utilizando cada árvore com raiz em cada processo.

A Figura 4(a) compara a latência das abordagens ATREE e ALL. Novamente, mesmo apresentando menor latência para sistemas até 256 processos, a estratégia ALL é superada pela estratégia proposta ATREE após este limiar. Em relação ao total de mensagens, percebe-se na Figura 4(b) que as duas estratégias tem comportamento semelhante. 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 ATREE, 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.

6. Conclusão

Este trabalho apresentou uma solução distribuída para a difusão seletiva confiável (*multicast*) em sistemas distribuídos 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.

Resultados de simulação comparando a solução proposta com outras duas abordagens mostra a eficiência do algoritmo em cenários com e sem falhas, especialmente para sistemas com mais de 256 processos. O cálculo da árvore sob demanda em cada processo e sem a necessidade de troca de mensagens diminui a latência e o total de mensagens.

Como trabalhos futuros, uma aplicação para exclusão mútua baseada em quóruns será implementada utilizando o algoritmo proposto.

Referências

Bista, B. (2005). A fault-tolerant scheme for multicast communication protocols. In: *11th Asia-Pacific Conference on Communications*, APCC'05, pp. 289–293.

Bolton, C. e Lowe, G. (2004). Analyses of the reverse path forwarding routing algorithm. In: *Int'l Conf. on Dependable Systems and Networks*, DSN'04, pp. 485–494.

Défago, X., Schiper, A. e Urbán, P. (2004). Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421.

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.

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

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.

Irwin, R. e Basu, P. (2013). Reliable multicast clouds. In: *IEEE Military Communications Conference*, MILCOM'13, pp. 1087–1092.

Khazan, R., Fekete, A. e Lynch, N. (1998). Multicast group communication as a base for a load-balancing replicated data service. In: Kutten, S., editor, *Distributed Computing*, v. 1499 de *LNCS*, pp. 258–272. Springer Berlin Heidelberg.

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.

Lau, P. (2006). A fault-tolerant best-effort multicast algorithm. In: *IEEE Int'l Conf. on Communication Technology*, ICCT'06, pp. 1–4.

Popescu, A., Constantinescu, D., Erman, D. e Ilie, D. (2007). A survey of reliable multicast communication. In: *3rd Conference on Next Generation Internet Networks*, EuroNGI'07, pp. 111–118.

Rahimi, M. e Sarsar, N. (2008). Load balanced multicast with multi-tree groups. In: *14th Asia-Pacific Conference on Communications*, APCC'08, pp. 1–5.

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

Rodrigues, L. A., Duarte Jr., E. P. e Arantes, L. (2014b). Uma solução autonômica para a criação de quóruns majoritários baseada no VCube. In: *Anais do XV Workshop de Testes e Tolerância a Falhas*, WTF'14, pp. 74–87.

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

Schiper, A. e Sandoz, A. (1993). Uniform reliable multicast in a virtually synchronous environment. In: *13th Int'l Conf. on Distrib. Comput. Syst.*, ICDCS'93, pp. 561–568.

Sutra, P. e Shapiro, M. (2008). Fault-tolerant partial replication in large-scale database systemsIn: Luque, E., Margalef, T. e Benítez, D., editores, *Euro-Par*, v. 5168 de *LNCS*, pp. 404–413. Springer Berlin Heidelberg.

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.