Árvores Geradoras Mínimas Distribuídas e Autonômicas

Luiz A. Rodrigues[1,2], 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]Laboratoire d'Informatique - Université Pierre et Marie Currie
CNRS/INRIA/REGAL – Place Jussieu, 4 – 75005 – Paris, France

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

Abstract. *This paper presents an autonomic algorithm for constructing minimum spanning trees in a distributed system. Processes are virtually organized on a hypercube-like topology, called VCube. The spanning trees are dynamically created from any source process and regenerated transparently when processes become faulty. Any tree reconstruction is autonomously done based on the virtual topology and on fault information, supporting up to n-1 faulty processes. Two applications using the autonomic spanning tree algorithm are proposed, one for the best-effort broadcast and another for reliable broadcast. Besides the formal specification, simulation results are presented, including comparison results with other alternative solutions.*

Resumo. *Este artigo apresenta um algoritmo autonômico para a construção de árvores geradoras mínimas (spanning trees) em um sistema distribuído. Os processos do sistema são virtualmente organizados em uma topologia baseada em hipercubo, denominada* VCube*. As árvores são criadas dinamicamente a partir de qualquer processo como raiz e regeneradas de forma transparente quando processos falhos são detectados. A reconstrução é feita autonomicamente com base na topologia virtual e nas informações de diagnóstico, suportando até* $n - 1$ *processos falhos. Com base no algoritmo de árvore, duas aplicações são propostas: uma para o broadcast de melhor-esforço e outra para broadcast confiável. Além da especificação formal, resultados de simulação são apresentados, inclusive comparando os algoritmos com outras soluções alternativas.*

1. Introdução

Árvores geradoras (*spanning trees*) são utilizadas na solução de diversos problemas em sistemas distribuídos, tais como exclusão mútua, agrupamento, fluxo em redes, sincronização e difusão de mensagens (*broadcast*) [Elkin 2006, England et al. 2007, Dahan et al. 2009]. A principal motivação é encontrar uma rede de baixo custo que conecta todos os processos do sistema. Para a difusão de mensagens, por exemplo, uma solução alternativa simples é utilizar inundação (*flooding*) para enviar uma mensagem a todos os processos do sistema. O inconveniente desta solução é o custo da comunicação devido ao grande número de vezes que a mensagem é retransmitida a um mesmo processo. Por outro lado, as árvores geradoras minimizam este custo, visto que a mensagem precisa ser transmitida somente pelas $n - 1$ arestas da árvore [Gärtner 2003].

A possibilidade de ocorrência de falhas é intrínseca aos sistemas distribuídos. Uma aplicação distribuída tolerante a falhas precisa continuar sua execução corretamente na presença de falhas, sem um comprometimento do desempenho. Idealmente a adaptação do serviço deve ocorrer de forma transparente, como acontece nos sistemas denominados autonômicos [Kephart e Chess 2003]. Para os algoritmos de árvores geradoras distribuídas, além do problema de construção, existe ainda o custo de manutenção das árvores quando ocorrem falhas dos seus nodos, que implica na sua reconstrução ou reconfiguração após uma falha.

O algoritmo proposto neste trabalho permite a propagação de mensagens de uma aplicação distribuída por meio de uma árvore geradora mínima construída de forma distribuída a partir de uma topologia baseada em hipercubo virtual, denominada VCube. No VCube, os processos são organizados em clusters hierárquicos progressivamente maiores. A partir da raiz principal da árvore, cada cluster possui um processo lider que é a raiz da sub-árvore naquele cluster, responsável pela propagação das mensagens nos clusters menores. Em função da topologia em hipercubo, a solução apresenta importantes propriedades logarítmicas, mesmo quando processos falham. Além disso, as árvores são reconstruídas de forma autonômica após a ocorrência de falhas.

Duas aplicações distribuídas para *broadcast* foram implementadas sobre o algoritmo autonômico proposto: uma para *broadcast* de melhor-esforço e outra para *broadcast* confiável. Resultados de simulação mostram que o uso da topologia virtual e do algoritmo de árvore aumentam a escalabilidade do sistema e reduzem a latência de propagação das mensagens nas duas aplicações.

O restante deste artigo está organizado nas seguintes seções. A Seção 2 descreve os trabalhos relacionados. A Seção 3 apresenta as definições e o modelo do sistema. A topologia virtual VCube é descrita na Seção 4. A Seção 5 apresenta o algoritmo de árvore geradora proposto. As aplicações baseadas em árvore são descritas na Seção 6 e resultados de simulação são apresentados na Seção 7. A Seção 8 apresenta a conclusão e os trabalhos futuros.

2. Trabalhos Relacionados

Os dois algoritmos clássicos para a obtenção de árvores geradoras mínimas a partir de um grafo são o algoritmo de Kruskal e Joseph (1956) e o proposto por Prim (1957). O algoritmo de Kruskal inicialmente cria uma floresta na qual cada vértice é uma árvore. A cada passo, as árvores são conectadas entre si através das arestas de menor peso. As arestas que não interligam duas árvores são descartas, evitando ciclos. Ao final, uma única componente conexa é gerada e esta constitui a árvore geradora mínima do grafo. O algoritmo de Prim utiliza uma abordagem diferente, que emprega cortes mínimos para escolher as arestas de menor peso para incluí-las na árvore.

Muitos algoritmos distribuídos para construção de árvores geradoras são baseados nos algoritmos centralizados de Kruskal-Joseph e Prim. O primeiro deles foi definido por Gallager et al. (1983). O processo é semelhante ao utilizado por Kruskal. Inicialmente cada nodo é uma árvore. A cada nível, um nodo é eleito líder e uma aresta de peso mínimo que o interliga a um nodo em outra árvore é adicionada. O processo é repetido até formar uma única componente conexa. O algoritmo proposto por Dalal (1987) utiliza o modelo de Prim para conectar segmentos da árvore escolhendo a aresta de menor peso que conecta dois segmentos.

Avresky (1999) apresenta três algoritmos para construção e manutenção de árvores em sistemas baseados em hipercubos sujeitos a falhas. Na fase inicial a árvore é construída usando busca em largura. Em caso de falha, o processo falho é desconectado da árvore e uma nova árvore é reconstruída pela conexão dos filhos do processo falho. Os algoritmos toleram falhas simples de processo e enlace, mas podem bloquear em certas combinações com falhas múltiplas.

O trabalho de Flocchini et al. (2012) propõe uma solução tolerante a falhas para árvores geradoras que reconstrói a árvore após a uma falha simples utilizando árvores alternativas pré-computadas. Isto é feito pelo cálculo e armazenamento distribuído das n árvores que podem ser geradas com um único processo falho. Em caso de recuperação do processo, a árvore anterior é restaurada para incluí-lo novamente na topologia.

3. Definições e Modelo do Sistema

Um sistema distribuído consiste de um conjunto finito Π de $n > 1$ processos independentes $\{p_0, ..., p_{n-1}\}$ que colaboram para a realização de alguma tarefa. Considera-se que cada processo é executado em um nodo distinto. Sendo assim, os termos *nodo* e *processo* são usados indistintamente. O sistema considerado é síncrono. Portanto, os atributos temporais relativos à velocidade de execução dos processos e ao atraso de mensagens nos canais de comunicação possuem limites conhecidos.

Seja $G = (V, E)$ o grafo conexo e não-direcionado que representa Π, no qual $n = |V|$ são os vértices que representam os processos e $m = |E|$ são as arestas, isto é, os enlaces de comunicação. Uma aresta (i, j) indica que o processo i pode se comunicar diretamente com o processo j e vice-versa. Uma árvore geradora (*spanning tree*) de G é um sub-grafo $T = (V, E')$ conexo e acíclico no qual $E' \subseteq E$, isto é, T contém todos os vértices de G e $|E'| = |V| - 1$. Se as arestas possuem pesos, uma *árvore geradora mínima* é aquela cujo a soma dos pesos das arestas é mínima. Se cada aresta possui um peso diferente, existe uma única árvore mínima. Se todas as arestas possuem o mesmo peso, todas as árvores do grafo são mínimas [Gallager et al. 1983]. Neste trabalho todos os enlaces possuem o mesmo peso.

Os processos comunicam-se enviando e recebendo mensagens. A topologia é totalmente conectada e cada par de processos está conectado por um enlace ponto-a-ponto bidirecional. Entretanto, os processos são organizados em um hipercubo virtual. Um hipercubo de d dimensões possui $n = 2^d$ processos. Cada processo i tem identificador entre 0 e $n - 1$, composto por um endereço binário de d *bits* $i_{d-1}, i_{d-2}, ..., i_0$. Dois processos estão conectados se os seus endereços diferem em apenas um *bit*. O envio e recebimento de mensagens são operações atômicas, mas se requer suporte extra para *multicast* e *broadcast*. Os enlaces são confiáveis e nunca falham. Portanto, nenhuma mensagem é perdida, corrompida ou duplicada durante a transmissão. Não existe particionamento da rede.

Os processos podem falhar por colapso (*crash*) e as falhas são permanentes. Um processo é dito *correto* ou *sem-falha* se não falha durante toda a execução do sistema. Caso contrário, o processo é considerado *falho*. Durante o colapso, o processo não executa qualquer ação e não responde aos estímulos externos, isto é, não executa processamento, nem envia ou recebe mensagens. Os processos falhos são detectados por um serviço de detecção de falhas perfeito, isto é, nenhum processo falho q é suspeito a menos que esteja realmente falho e, se q está falho, todo processo correto p será notificado pelo módulo detector sobre a falha de p em um tempo finito [Freiling et al. 2011].

4. VCube: Uma Topologia Virtual Escalável

O uso de uma topologia virtual para conectar processos de um sistema distribuído facilita a construção das aplicações, pois abstrai a estrutura física e facilita a reconfiguração do sistema quando necessário. A topologia empregada neste trabalho, denominada VCube, organiza os processos do sistema em um hipercubo virtual quando não existem falhas. Porém, se uma falha é detectada, os enlaces virtuais são reconfigurados para se adaptar ao novo estado do sistema. Mesmo na presença de falhas, VCube mantém as seguintes propriedades logarítmicas: em um hipercubo de d dimensões com $n = 2^d$ vértices, cada vértice está conectado a até $\log n$ vizinhos e a distância máxima entre dois vértices é $\log n$.

Para construir o VCube é utilizada a organização hierárquica proposta por Duarte Jr. e Nanya (1998). Os processos são organizados em clusters progressivamente maiores, conforme ilustrado na Figura 1. Cada cluster $s = 1, .., \log n$ possui 2^{s-1} processos. Nos clusters de nível $s = 1$ cada cluster possui um elemento. No segundo nível, dois clusters de tamanho um são agrupados, formando um cluster $s = 2$ de tamanho dois. No terceiro nível, dois clusters de tamanho dois se unem para formar um cluster $s = 3$ de tamanho quatro, e assim sucessivamente.

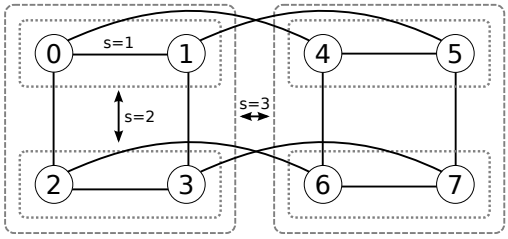


Figura 1. Organização em clusters no VCube de três dimensões.

Os processos integrantes de cada cluster s em relação a um processo i são listados em ordem pela função $c_{i,s} = \{i \oplus 2^{s-1}, c_{i\oplus 2^{s-1},1}, ..., c_{i\oplus 2^{s-1},s-1}\}$, na qual $\oplus$ representa a operação binária de *ou* exclusivo (*xor*).

Utilizando a topologia criada pelo VCube, foram definidas algumas funções. Sejam i e j dois processos do sistema. A função $cluster_i(j) = s$ determina a qual cluster s do processo i o processo j pertence. Como exemplo, no hipercubo de três dimensões da Figura 1, $cluster_0(1) = 1$, $cluster_0(2) = cluster_0(3) = 2$, $cluster_0(5) = 3$. Note que para todo par i, j, $cluster_i(j) = cluster_j(i)$.

A segunda função é $FF\_neighbor_i(s) = j$. Esta função calcula o primeiro processo j correto do cluster $c_{i,s}$. Se todos os processos no cluster estão falhos, a função retorna $\perp$. No hipercubo da Figura 1, por exemplo, em um cenário sem falhas $FF\_neighbor_4(1) = 5$, $FF\_neighbor_4(2) = 6$, $FF\_neighbor_4(3) = 0$. Por outro lado, se p_6 está falho e p_4 já foi informado sobre esta falha pelo módulo detector (isto é, p_6 não pertence a $correct_4$), $FF\_neighbor_4(2) = 7$. Se p_6 e p_7 estão falhos, $FF\_neighbor_4(2) = \perp$. É importante ressaltar que esta função é dependente do conhecimento atual que um processo tem a respeito das falhas no sistema. Devido à latência para a detecção, em um mesmo espaço de tempo, é possível que dois processos possuam visões diferentes sobre quais processos estão corretos ou falhos.

Por fim, a função $neighborhood_i(h) = \{j \mid j = FF\_neighbor_i(s), j \neq \perp, 1 \leq s \leq h\}$ é definida. Esta função gera um conjunto que contém todos os processos

sem-falha virtualmente conectados ao processo i de acordo com $FF\_neighbor_i(s)$, para $s = 1, .., h$. O parâmetro h pode variar de 1 a $\log n$. Se $\log n$ é utilizado, o conjunto resultante contém todos os vizinhos corretos do processo i no VCube. Para qualquer outro valor de $h < \log n$, a função retorna apenas um subconjunto dos vizinhos contidos nos clusters $s = 1, .., h$. Assim como $FF\_neighbor_i$ esta função depende do conhecimento local que o processo i tem sobre o estado dos outros processos. Como exemplo, para o VCube da Figura 1 e em um cenário sem falhas, $neighborhood_0(1) = \{1\}$, $neighborhood_0(2) = \{1, 2\}$ e $neighborhood_0(3) = \{1, 2, 4\}$. Se o processo p_4 é detectado como falho pelo processo p_0 ($4 \notin correct_0$) então $neighborhood_0(3) = \{1, 2, 5\}$. Se p_1 está falho, $neighborhood_0(1) = \emptyset$.

Assim, a topologia do sistema no VCube é formada pela conexão de cada processo i com todos os seus vizinhos determinados por $neighborhood_i(\log n)$. De forma análoga, os vizinhos de um processo i restritos ao cluster s ao qual i pertence em relação a outro processo j são determinados por $neighborhood_i(cluster_i(j) - 1)$.

5. O Algoritmo Autonômico para Árvores Geradoras

Esta seção apresenta o algoritmo proposto para construir de forma autonômica uma árvore geradora mínima em um sistema distribuído com base na topologia VCube.

O Algoritmo 5.1 propõe um mecanismo de disseminação de informação para um sistema distribuído que propaga as mensagens do sistema a partir de um processo qualquer, denominado fonte. O algoritmo é considerado autonômico porque reconstrói a árvore dinamicamente à medida que processos falhos são detectados. Uma das vantagens da solução é que, ao invés de iniciar a reconstrução a partir do processo fonte, a regeneração da árvore é realizada apenas localmente, de acordo com o ramo afetado.

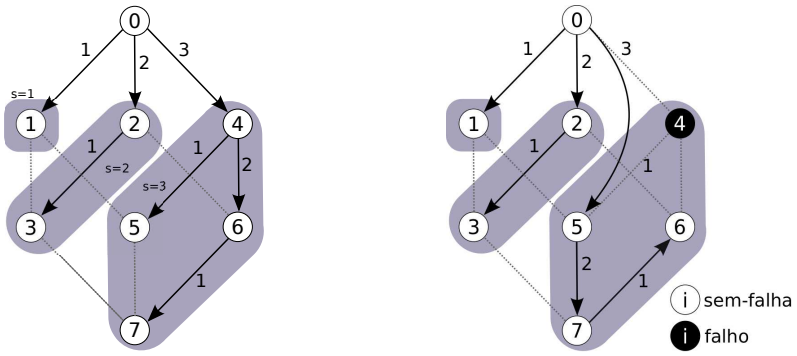
Algoritmo 5.1 Algoritmo Distribuído de Árvore Geradora Mínima no processo i

```
1: correct_i ← {0, .., n − 1} //lista dos processos corretos
2: procedure STARTTREE( )
3:     //envia a todos os vizinhos
4:     for all k ∈ neighborhood_i(log n) do
5:         SEND(⟨TREE⟩) para p_k
6: procedure RECEIVE(⟨TREE⟩) from p_j
7:     if j ∈ correct_i then
8:         //retransmite aos vizinhos dos clusters internos
9:         for k ∈ neighborhood_i(cluster_i(s) − 1) do
10:            SEND(⟨TREE⟩) para p_k
11: procedure CRASH(processo j) //j é detectado como falho
12:     correct_i ← correct_i ∖ {j}
13:     if k = FF_neighbor_i(cluster_i(j)), k ≠ ⊥ then
14:         SEND(⟨TREE⟩) to p_k
```

Considere inicialmente uma execução sem falhas. No primeiro passo, a propagação é iniciada no procedimento STARTTREE. Uma mensagem TREE é enviada para os $\log n$ vizinhos sem falha, um em cada cluster $s = 1, .., \log n$ (linhas 4-5). Ao receber uma mensagem de um processo j, o procedimento RECEIVE é executado e o processo i encaminha a mensagem para os *clusters* internos ao seu próprio cluster, isto é, aos clusters $s' = 1, .., cluster_i(j) - 1$ (linha 9).

Como exemplo, considere o VCube sem processos falhos da Figura 2(a). O processo p_0 é a raiz e envia TREE para os vizinhos $FF\_neighbor_0(1) = 1$, $FF\_neighbor_0(2) = 2$ e $FF\_neighbor_0(3) = 4$. O processo p_1 recebe a mensagem, mas não retransmite, visto $cluster_1(0) = 1$ e $neighborhood_1(0) = \perp$. O processo p_2

recebe a mensagem e retransmite para seu vizinho p_3 no cluster $s = 1$. Quando p_3 recebe a mensagem ele calcula $cluster_3(2) = 1$ e para a retransmissão. No caso de p_4 a mensagem e recebida e retransmitida para os vizinhos $5 \in c_{4,1}$ e $6 \in c_{4,2}$. Finalmente, sendo $FF\_neighbor_6(1) = 7$, o processo p_6 envia a mensagem para o processo p_7.



(a) sem-falhas (b) processo p_4 falho

Figura 2. Árvore geradora no VCube de 3 dimensões.

Os casos com falhas podem ser divididos em dois cenários. Primeiramente, considere que um processo $j \in c_{i,s}$ está falho e o processo i já foi informado sobre esta falha pelo detector, isto é, $j \notin correct_i$ (linha 11). Neste caso, o processo i envia a mensagem para o vizinho sem falha $k = FF\_neighbor_i(s)$ e a mensagem é propagada corretamente por k a todos os clusters internos do cluster s. Em um segundo cenário, este mesmo processo j está falho, mas o processo i ainda não foi informado pelo detector, isto é, $j \in correct_i$. Neste caso, se $FF\_neighbor_i(s) = j$, a mensagem é enviada ao processo falho j e é descartada. A propagação termina prematuramente e a sub-árvore interna ao cluster s é desconectada. Entretanto, assim que o módulo detector informa i sobre a falha, um novo $FF\_neighbor_i(s) = k$ é eleito e a mensagem é retransmitida para k, reconstruindo-se assim a sub-árvore do cluster s (linha 13).

A Figura 2(b) ilustra um cenário com falhas. Seja o processo p_0 novamente a raiz e p_4 falho. Estando p_0 ciente da falha do processo p_4, ao invés de enviar TREE para p_4, p_0 transmite a mensagem a p_5 que é o primeiro processo sem falha de $c_{0,3}$. O processo p_5 por sua vez, repassa a mensagem para $p_7 \in c_{5,2}$. Por fim, p_7 retransmite a mensagem para $p_6 \in c_{7,1}$, completando a árvore. Se quando p_0 inicia o *broadcast* e $p_4 \in correct_i$, p_0 envia a mensagem para p_4 e, quando o detector informá-lo sobre a falha de p_4, a mensagem será retransmitida para p_5. Deste ponto em diante a propagação é análoga ao caso anterior.

O Teorema 1 formaliza a propagação em árvore proposta pelo Algoritmo 5.1.

Teorema 1. *Seja m uma mensagem propagada por um processo fonte src correto. Todo processo correto no sistema Π recebe m.*

Prova. A prova deste lema é por indução. Considere como base da indução um sistema com $n = 2$ processos: p_0 é o processo src que inicia o envio da mensagem m e $p_1 \in c_{0,1}$. Se p_1 é correto, $FF\_neighbor_0(1) = 1$ e p_0 envia m para p_1 (linha 4). Portando, p_1 recebe m e o lema é válido.

Como hipótese da indução, considere que o lema é válido para um sistema com $n = 2^k$ processos.

No passo da indução é demonstrado que o lema é válido para um sistema com $n = 2^{k+1}$ processos. Pela organização hierárquica do VCube, este sistema é constituído de dois subsistemas com $n = 2^k$ processos, como ilustrado pela Figura 3. A figura mostra que src e j são as raízes destes subsistemas. O processo src executa o algoritmo e envia m para cada processo retornado por $FF\_neighbor_{src}(s)$, $s = 1, .., k$ (linha 4). Sendo $j = FF\_neighbor_{src}(k)$ um processo correto, j corretamente recebe m. Se j é detectado como falho, uma cópia da mensagem é retransmitida para o próximo processo correto no mesmo cluster de j (linha 13). Assim, a mensagem m é transmitida nos dois subsistemas e, pela hipótese, todo processo correto recebe m em cada subsistema. Como todo processo em Π pertence a um destes sistemas, todo processo correto em Π recebe m.

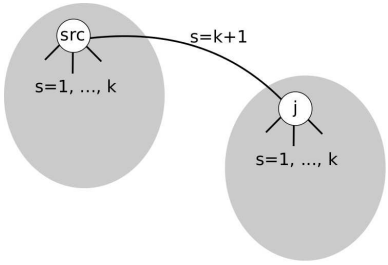


Figura 3. Propagação das mensagens de *broadcast*.

6. Broadcasts com VCube

Broadcast é um componente básico para implementar muitos algoritmos e serviços distribuídos como notificação, entrega de conteúdo, *publish/subscribe*, replicação e comunicação em grupo [Lin 2009]. Um processo utiliza *broadcast* para enviar uma mensagem a todos os outros processos do sistema. No entanto, se um processo falha durante o procedimento de difusão, alguns processos podem receber a mensagem enquanto outros não. O *broadcast* 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 *broadcast* confiável precisam ser implementadas [Hadzilacos e Toueg 1993].

Algoritmos de *broadcast* 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 receber mensagens para/de outros processos da aplicação, respectivamente. Para incluir tolerância a falhas, um detector de falhas perfeito pode ser utilizado para notificar o algoritmo de *broadcast*, que deve reagir apropriadamente quando uma falha é detectada.

Neste artigo, dois algoritmos de *broadcast* tolerante a falhas são descritos. O primeiro implementa *broadcast* de melhor-esforço e o segundo propõe uma solução confiável. As duas soluções utilizam o modelo de árvores geradoras proposto na Seção 5.

6.1. Broadcast de Melhor-esforço

Broadcast de melhor-esforço garante que todos os processos corretos entregarão o mesmo conjunto de mensagens se o emissor (fonte) é correto. Três propriedades caracterizam este modelo: entrega confiável (*validity*), não-duplicação e não-criação de mensagens

[Guerraoui e Rodrigues 2006]. A entrega confiável garante que, se um processo p_i envia uma mensagem m para um processo p_j e nenhum deles falha, p_j recebe m em um tempo finito. A não-duplicação garante que nenhuma mensagem é entregue mais de uma vez e a não-criação garante que nenhuma mensagem é entregue a menos que tenha sido previamente enviada.

O Algoritmo 6.1 apresenta uma solução para *broadcast* de melhor-esforço que utiliza o mecanismo de árvore proposto neste trabalho. Dois tipos de mensagens são utilizados: $\langle TREE, m\rangle$ para identificar as mensagens de aplicação e $\langle ACK, m\rangle$ para confirmar o recebimento das mensagens de aplicação. Os processos são informados sobre o estado dos demais processos pelo algoritmo que mantém o VCube. O algoritmo executa corretamente mesmo que $n-1$ falhas ocorram. Um versão preliminar deste algoritmo foi publicada em Rodrigues (2013) aplicado a uma solução tolerante a falhas de exclusão mútua distribuída.

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;
- 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.

Um processo i que deseja difundir uma mensagem m por *broadcast* invoca o método BROADCAST. A linha 5 garante que um novo *broadcast* só é iniciado após o término do anterior, isto é, quando não há mais ACKs pendentes para a mensagem $last_i[i]$. Nas linhas 9-11 a nova mensagem é enviada a todos os vizinhos considerados corretos. Para cada mensagem enviada, um ACK é incluído na lista de ACKs pendentes.

Quando um processo recebe uma mensagem TREE de um processo j (linha 16), primeiramente ele verifica se tanto o processo fonte da mensagem quanto o processo j são considerados corretos. Se um deles esta falho, o recebimento é abortado, pois se j está falho, o processo que transmitiu m para j fará uma nova transmissão quando detectar a falha e i irá receber a mensagem através da nova árvore que será reconstruída. Além disso, se a fonte está falha, não é mais necessário continuar a retransmissão. Se a fonte e j estão corretos, o processo i verifica se a mensagem é nova comparando os *timestamps* da última mensagem armazenada em $last_i[j]$ e da mensagem recebida m (linha 21). Se m é nova, $last_i[j]$ é atualizado e a mensagem é entregue à aplicação. Em seguida, m é retransmitida para os vizinhos em cada cluster interno ao cluster de i. Se não existe vizinho correto ou se i é uma folha na árvore ($cluster_i(j) = 1$), nenhum ACK pendente é adicionado ao conjunto ack_set_i e CHECKACKS envia um ACK para j.

Se 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 $k = \perp$, o ACK alcançou o processo fonte e não precisa mais ser propagado.

Algoritmo 6.1 Broadcast de melhor-esforço hierárquico no processo i

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

4: procedure BROADCAST(message m)
5:     wait until ack_set_i ∩ {⟨⊥, ∗, last_i[i]⟩} = ∅
6:     last_i[i] = m
7:     DELIVER(m)
8:     //envia a todos os vizinhos
9:     for all j ∈ neighborhood_i(log_2 n) do
10:        ack_set_i ← ack_set_i ∪ {⟨⊥, j, m⟩}
11:        SEND(⟨TREE, m⟩) to p_j

12: procedure CHECKACKS(processo j, mensagem m)
13:    if ack_set_i ∩ {⟨j, ∗, m⟩} = ∅ then
14:        if {source(m), j} ⊆ correct_i then
15:            SEND(⟨ACK, m⟩) to p_j

16: procedure RECEIVE(⟨TREE, m⟩) from p_j
17:    if {source(m), j} ⊈ correct_i then
18:        return
19:    //verifica se m é nova
20:    if last_i[source(m)] = ⊥ or
21:       ts(m) = ts(last_i[source(m)]) + 1 then
22:        last_i[source(m)] ← m
23:        DELIVER(m)
24:    //retransmite aos vizinhos dos clustes internos
25:    for all k ∈ neighborhood_i(cluster_i(j) − 1) do
26:        if ⟨j, k, m⟩ ∉ ack_set_i then
27:            ack_set_i ← ack_set_i ∪ {⟨j, k, m⟩}
28:            SEND(⟨TREE, m⟩) to p_k
29:    CHECKACKS(j, m)

30: procedure RECEIVE(⟨ACK, m⟩) from p_j
31:    k ← x : ⟨x, j, m⟩ ∈ ack_set_i
32:    ack_set_i ← ack_set_i ∖ {⟨k, j, m⟩}
33:    if k ≠ ⊥ then
34:        CHECKACKS(k, m)

35: procedure CRASH(processo j) //j é detectado como falho
36:    correct_i ← correct_i ∖ {j}
37:    k ← FF_neighbor_i(cluster_i(j))
38:    for all p = x, q = y, m = z : ⟨x, y, z⟩ ∈ ack_set_i do
39:        if {source(m), p} ⊈ correct_i then
40:            //remove ACKs pendentes para ⟨j, ∗, ∗⟩ e
41:            //⟨∗, ∗, m⟩ : source(m) = j
42:            ack_set_i ← ack_set_i ∖ {⟨p, q, m⟩}
43:        else if q = j then //retransmite para novo vizinho k
44:            if k ≠ ⊥ and ⟨p, k, m⟩ ∉ ack_set_i then
45:                ack_set_i ← ack_set_i ∪ {⟨p, k, m⟩}
46:                SEND(⟨TREE, m⟩) to p_k
47:            ack_set_i ← ack_set_i ∖ {⟨p, j, m⟩}
48:            CHECKACKS(p, m)
```

A detecção de um processo falho j é tratada no procedimento CRASH. 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 j para o novo vizinho k no mesmo cluster de j, se existir um. Esta retransmissão desencadeia uma propagação na nova estrutura da árvore.

No Teorema 2 é provada a correção da solução de *broadcast* de melhor-esforço implementada pelo Algoritmo 6.1.

Teorema 2. *O Algoritmo 6.1 é uma solução para broadcast de melhor-esforço. O algoritmo garante que todo processo correto receberá o mesmo conjunto de mensagens se o emissor é correto.*

Prova. As propriedades de não-duplicação e não-criação são derivadas das propriedades dos enlaces, os quais são considerados confiáveis. Além disso, toda mensagem possui um *timestamp* único e, mesmo que seja retransmitida após uma falha, o receptor fará a entrega uma única vez (linha 21).

A entrega confiável é garantida pelo processo emissor (fonte). Quando um emissor envia uma mensagem m, ele aguarda pelas mensagens de confirmação (ACKs) de todos os vizinhos corretos. O Teorema 1 provou que se o emissor é correto, todo processo correto recebe m, mesmo que um processo intermediário j falhe durante a retransmissão. Neste caso, a mensagem é retransmitida para o próximo vizinho sem falha k no mesmo cluster do processo j (linha 37).

Teorema 3. *O total de mensagens enviadas para cada mensagem de aplicação em uma execução sem falhas do Algoritmo 6.1 é* $2 * (n - 1)$. *Se um processo* j *que recebeu a mensagem de um processo* j *falha antes de confirmar o recebimento, o total de mensagens extra depende da quantidade de processos no cluster de* j.
Prova. Em uma execução sem falhas, para cada mensagem TREE enviada, uma mensagem ACK é retornada. Seja $n - 1$ o número de arestas na árvore com n processos, o total de mensagens é o dobro do total de arestas.

Se um processo j é detectado como falho por um processo i depois que a mensagem TREE foi enviada para j, uma nova mensagem será enviada para o próximo vizinho $k = FF\_neighbor_i(cluster_i(j))$. Seja $s = cluster_i(j)$. No melhor caso, $k = \perp$ e nenhuma mensagem extra é enviada ($j \in c_{i,1}$ ou não existem mais processos corretos no cluster s). No entanto, se $k \neq \perp$, a quantidade de mensagens extras depende do número de processos detectados corretos no cluster s. Seja $n' = |c_{i,s}|$ o total de processos no cluster $c_{i,s}$. No pior caso, todos os processos do cluster estão corretos, exceto j, e j enviou a mensagem a todos os vizinhos antes de falhar. Assim, o total de mensagens extras será $1 + 2 * (n' - 2)$, uma TREE extra para k e $(n' - 2)$ TREEs + $(n' - 2)$ ACKs na subárvore. De forma geral, se existem f processos falhos em $c_{i,s}$ incluindo j, a quantidade de mensagens extras retransmitidas é $1 + 2 * (n' - 1 - f)$.

6.2. Broadcast Confiável

Um algoritmo de *broadcast* confiável garante que o mesmo conjunto de mensagens é entregue a todos os processos corretos, mesmo se o emissor (fonte) falhar durante o procedimento de difusão. Para tanto, o *broadcast* confiável herda as propriedades de entrega confiável, não-criação e não-duplicação do melhor-esforço e acrescenta a propriedade de acordo (*agreement*). Assim, a solução para *broadcast* confiável proposta neste trabalho é uma modificação do *broadcast* de melhor-esforço descrito na Seção 6.1 que inclui o tratamento para a falha do processo fonte. O Algoritmo 6.2 apresenta as modificações realizadas. As variáveis $last_i$, ack_set e $correct_i$ são as mesmas, bem como os métodos RECEIVE(ACK, m) e CHECKACKS.

Um processo p_i que deseja efetuar o *broadcast* invoca o procedimento BROADCAST(m). Se a mensagem m tem origem no mesmo processo, $source(m) = i$ e o tratamento é o mesmo realizado pelo algoritmo de melhor-esforço. A mensagem é então propagada para todos os vizinhos de p_i. Quando um processo p_i recebe a mensagem $\langle TREE, m\rangle$ de um processo p_j ele primeiramente verifica se $j \notin correct_i$. Se p_j está falho, a mensagem é descartada. Note que esta verificação difere do algoritmo de melhor-esforço pois não descarta mensagens recebidas de um $source(m)$ falho. A segunda modificação está nas linhas 15-17. Se p_i recebe uma mensagem nova de um processo fonte considerado falho, ele inicia um novo *broadcast* com a mensagem recebida para garantir que os demais processos recebam m corretamente. Neste caso, quando a linha 2 é executada, $source(m)$ não será igual a i e a mensagem é retransmitida aos demais processos através da árvore geradora de p_j.

A recuperação em caso de falhas também é muito semelhante à solução de melhor-esforço, exceto pelo *broadcast* da última mensagem recebida do processo detectado como falho (linha 33). Embora a prova de correção da solução tenha sido omitida, esta retransmissão, em conjunto com a retransmissão da linha 16, garante que todos os demais processos corretos receberão a última mensagem transmitida pelo processo fonte p_j falho, mesmo que um único processo tenha recebido a mensagem antes da falha de p_j.

Algoritmo 6.2 Broadcast de melhor-esforço hierárquico no processo i

```
1: procedure BROADCAST(message m)
2:     if source(m) = i then
3:         wait until ack_set_i ∩ {⟨⊥, *, last_i[i]⟩} = ∅
4:         last_i[i] = m
5:         DELIVER(m)
6:     ...
7: procedure RECEIVE(⟨TREE, m⟩) from p_j
8:     if j ∉ correct_i then
9:         return
10:    //verifica se m é nova
11:    if last_i[source(m)] = ⊥ or
12:       ts(m) = ts(last_i[source(m)]) + 1 then
13:        last_i[source(m)] ← m
14:        DELIVER(m)
15:        if source(m) ∉ correct_i then
16:            BROADCAST(m)
17:            return
18:    ...
19: procedure CRASH(processo j) //j é detectado como falho
20:    correct_i ← correct_i ∖ {j}
21:    k ← FF_neighbor_i(cluster_i(j))
22:    for all p = x, q = y, m = z : ⟨x, y, z⟩ ∈ ack_set_i do
23:        if p ∉ correct_i then
24:            //remove ACKs pendentes para ⟨j, *, *⟩
25:            ack_set_i ← ack_set_i ∖ {⟨p, q, m⟩}
26:        else if q = j then //retransmite para novo vizinho k
27:            if k ≠ ⊥ and ⟨p, k, m⟩ ∉ ack_set_i then
28:                ack_set_i ← ack_set_i ∪ {⟨p, k, m⟩}
29:                SEND(⟨TREE, m⟩) to p_k
30:            ack_set_i ← ack_set_i ∖ {⟨p, j, m⟩}
31:            CHECKACKS(p, m)
32:    if last_i[j] ≠ ⊥ then
33:        BROADCAST(last_i[j])
```

7. Avaliação Experimental

Os algoritmos de *broadcast* propostos foram implementados utilizando o Neko [Urbán et al. 2002], um *framework* Java para simulação de algoritmos distribuídos, e comparados com duas outras soluções. Para facilitar a discussão dos resultados, o *broadcast* de melhor-esforço foi chamado de ATREE-B e a abordagem confiável de ATREE-R. A primeira solução comparada utiliza uma abordagem todos-para-todos (nomeadas ALL-B e ALL-RB) e a segunda implementa uma estratégia baseada em árvore, porém, não autonômica (nomeadas NATREE-B e NATREE-R). ATREE, ALL e NATREE são utilizadas para referenciá-las de forma genérica. Para implementar a estratégia ALL, a função $neighborhood$ foi modificada para incluir todos os processos sem-falha, fazendo com que o emissor envie a mensagem a todos os demais processos diretamente utilizando enlaces ponto-a-ponto. A estratégia NATREE constrói uma árvore utilizando inundação. A árvore é criada utilizando a própria mensagem TREE de *broadcast*. Cada processo que recebe a mensagem pela primeira vez se junta à árvore e retransmite a mensagem aos demais processos do sistema, exceto àquele do qual ele recebeu a mensagem. Se um processo já está na árvore e recebe uma outra cópia da mensagem ele envia uma mensagem de NACK e não retransmite mais a mensagem. Uma vez criada a árvore, as demais mensagens são enviadas somente pelas arestas da árvore. Em caso de falha, uma nova árvore é criada a partir do processo fonte utilizando o mesmo procedimento de inundação.

O mecanismo de detecção de falhas utilizado é o Hi-ADSD (*Hierarchical Adaptive Distributed System-Level Diagnosis*), proposto por Duarte Jr. e Nanya (1998). Trata-se de um algoritmo de diagnóstico distribuído e adaptativo com baixa latência e reduzido número de mensagens quando comparado a outras soluções existentes. A latência de diagnóstico varia de $\log n$ a $\log^2 n$ rodadas.

7.1. Parâmetros de Simulação

Uma vez que não existem mecanismos auxiliares de *broadcast* e *multicast*, 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.

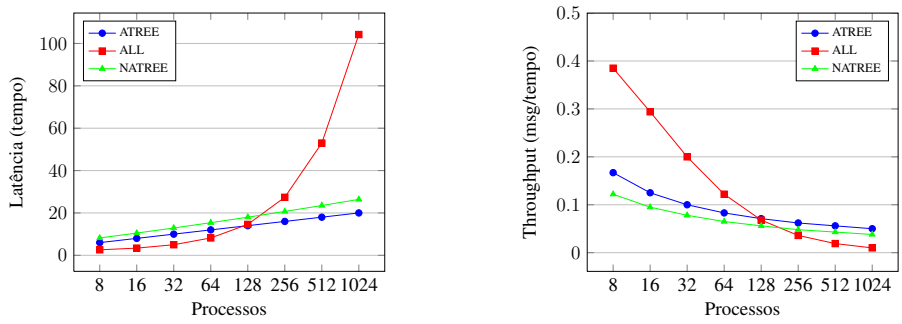
Para avaliar o desempenho de soluções de *broadcast*, duas métricas são frequentemente utilizadas [Boichat e Guerraoui 2005]: (1) *throughput*, dado pelo total de mensagens de *broadcast* completos durante um intervalo de tempo; (2) a latência para entregar a mensagem de *broadcast* a todos os processos corretos.

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.

7.2. Resultado dos Experimentos

Os experimentos foram realizados em duas etapas. Inicialmente, foram utilizados cenários sem falhas com sistemas de 8 a 1024 processos. Além disso, a latência e a quantidade de mensagens enviadas por cada aplicação foram analisadas pontualmente em um sistema com 512 processos. Em seguida, cenários com falhas foram gerados aleatoriamente para um sistema com 512 processos contendo de 1 a 8% de processos falhos.

Experimentos em cenários sem-falhas. Nos cenários sem-falhas, como não há retransmissões, os algoritmos de *broadcast* de melhor-esforço e confiável propostos possuem o mesmo comportamento. A Figura. 4 mostra a latência e o *throughput* considerando que uma única mensagem é enviada pelo processo p_0. O caminho mais longo em um VCube com n processos é $\log n$. Portanto, quando n é pequeno, o tempo para enviar a mensagem pelo caminho mais longo é maior que o tempo para enviar as $n - 1$ mensagens sequencialmente pela estratégia ALL. Considerando que o tempo de envio de cada mensagem é $t_s = 0.2$, o intervalo entre o envio da mensagem TREE e a recepção do ACK correspondente através do caminho mais longo da árvore é $2 \log_2 n(t_s + t_r + t_t)$. Já na estratégia ALL, para enviar $n - 1$ mensagens são utilizadas $(n - 2)t_s + t_t + tr$ unidades de tempo.



(a) Latência (b) Throughput

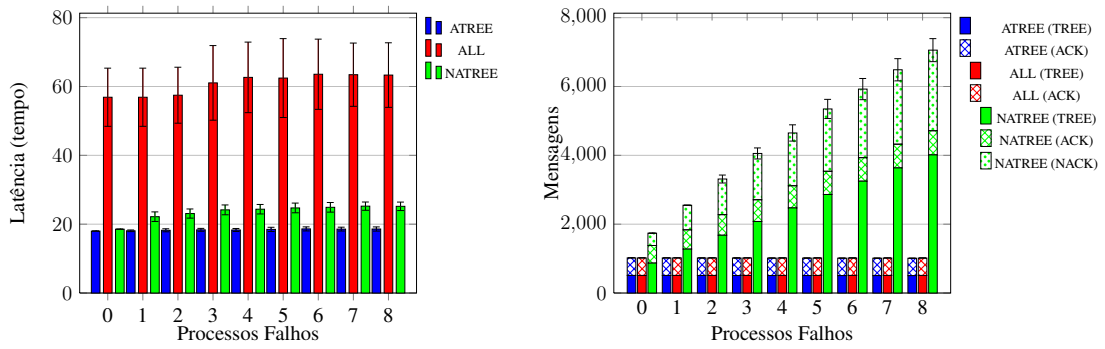
Figura 4. Broadcast de melhor-esforço em uma execução sem-falhas.

Assim, embora ALL seja mais eficiente para sistemas pequenos, seu desempenho cai rapidamente em sistemas maiores. A solução com árvore não-autonômica NATREE apresenta comportamento semelhante a ATREE, exceto pela latência extra para aguardar as mensagens de ACK/NACK durante a configuração da árvore. Em cenários sem-falha, o desempenho das duas soluções passa a ser o mesmo depois após a árvore estar completa.

O *throughput* foi calculado como $1/latency$, visto que apenas um processo envia uma única mensagem. Assim como a latência, na solução ALL o desempenho é melhor

para sistemas pequenos, até 128 processos, mas diminui rapidamente quando n aumenta. Estes resultados confirmam a escalabilidade da estratégia hierárquica proposta.

Experimentos em cenários com falhas. Nos cenários com falhas foram utilizados sistemas com 512 processos com variações na quantidade de processos falhos. Falhas foram geradas em um intervalo de 1 a $\log n$. Para cada variação de falhas foram gerados 100 diferentes cenários com falhas distribuídas aleatoriamente usando uma distribuição normal. Diferente dos testes sem falha nos quais uma única mensagem é enviada, nos experimentos com falha os processos enviam 10 mensagens de *broadcast*. A média da latência está representada no gráfico da Figura 5(a). A solução ALL tem latência maior que ATREE porque precisa enviar todas as mensagens sequencialmente, como já justificado anteriormente. Na solução NATREE, o resultado é similar a ALL visto que, para cada falha, a árvore precisa ser reconstruída por inundação. ALL é mais eficiente para cenários sem-falha ou com poucas falhas, mas degrada rapidamente quando a quantidade de falhas aumenta. Em relação à quantidade de mensagens, para ATREE e ALL o resultado é muito semelhante, conforme registrado na Figura 5(b). No caso de NATREE, a reconstrução da árvore a partir da raiz aumenta consideravelmente o total de mensagens enviadas. Estes resultados demonstram a eficiência da solução proposta na criação e manutenção autonômica da árvore, neste caso utilizada pelo *broadcast*.



(a) Latência (b) Total de mensagens

Figura 5. Broadcast confiável com $n = 512$ e diferente número de falhas.

8. Conclusão

Este trabalho apresentou uma solução para criação autonômica de árvores geradoras mínimas em sistemas distribuídos sujeitos a falhas. Os processos do sistema são organizados em uma topologia lógica, denominada VCube. As árvores são geradas dinamicamente a partir de um processo fonte e reconfiguradas localmente quando processos falham. Dois algoritmos de *broadcast* foram implementados utilizando a estratégia de árvore proposta, uma solução de *broadcast* de melhor-esforço e uma outra abordagem confiável. Resultados de simulação demonstraram a eficiência das soluções propostas, destacando a escalabilidade delas.

Como trabalhos futuros, pretende-se estender o VCube para falhas *crash* com recuperação. Outro caminho a ser seguido é considerar a modificação do modelo para tratar falhas de particionamento de rede. Além disso, espera-se implementar o algoritmo de árvore geradora como um serviço que possa ser invocado por aplicações arbitrárias sobre uma rede.

Referências

Avresky, D. (1999). Embedding and reconfiguration of spanning trees in faulty hypercubes. *Parallel and Distributed Systems, IEEE Transactions on*, 10(3):211–222.

Boichat, R. e Guerraoui, R. (2005). Reliable and total order broadcast in the crash-recovery model. *J. Parallel Distrib. Comput.*, 65(4):397–413.

Dahan, S., Philippe, L. e Nicod, J. (2009). The distributed spanning tree structure. *Parallel and Distributed Systems, IEEE Transactions on*, 20(12):1738–1751.

Dalal, Y. (1987). A distributed algorithm for constructing minimal spanning trees. *Software Engineering, IEEE Transactions on*, SE-13(3):398–405.

Duarte Jr., E. P. e Nanya, T. (1998). A hierarchical adaptive distributed system-level diagnosis algorithm. *IEEE Trans. Comput.*, 47(1):34–45.

Elkin, M. (2006). A faster distributed protocol for constructing a minimum spanning tree. *J. Comput. Syst. Sci.*, 72(8):1282–1308.

England, D., Veeravalli, B. e Weissman, J. (2007). A robust spanning tree topology for data collection and dissemination in distributed environments. *IEEE Trans. Parallel Distr. Syst.*, 18(5):608–620.

Flocchini, P., Mesa Enriquez, T., Pagli, L., Prencipe, G. e Santoro, N. (2012). Distributed minimum spanning tree maintenance for transient node failures. *IEEE Trans. Comput.*, 61(3):408–414.

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

Gallager, R. G., Humblet, P. A. e Spira, P. M. (1983). A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Program. Lang. Syst.*, 5:66–77.

Gärtner, F. C. (2003). A survey of self-stabilizing spanning-tree construction algorithms. Technical report, Swiss Federal Institute of Technology (EPFL).

Guerraoui, R. e Rodrigues, L., editores (2006). *Introduction to Reliable Distributed Programming*. Springer-Verlag, Berlin, Germany.

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

Kephart, J. e Chess, D. (2003). The vision of autonomic computing. *Computer*, 36(1):41–50.

Kruskal Jr., J. B. (1956). On the shortest spanning subtree of a graph and the traveling salesman problem. Anais do *American Mathematical Society*, páginas 48–50.

Lin, J.-W. (2009). Multi-tree broadcast in peer-to-peer networks. Anais do *Int'l Conf. Internet Multimedia Syst. Archit. and Applications*.

Prim, R. C. (1957). Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36(6):1389–1401.

Rodrigues, L. A. (2013). Fault-tolerant broadcast algorithms for the virtual hypercube topology. Anais do *Student Forum - DSN-W'43*, páginas 1–4.

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.