Abaixo o Líder: Um Algoritmo Hierárquico Sem-Líder para Difusão Atômica

Lucas V. Ruchel[1*], Edson T. de Camargo[1†], Rogério Turchetti[‡], Elias P. Duarte Jr.[§], Luciana Arantes[¶] e Luiz A. Rodrigues[1]

[1]Universidade Estadual do Oeste do Paraná (Unioeste) – Cascavel – PR – Brasil
Programa de Pós-Graduação em Ciência da Computação - PPGComp

Abstract. *This work presents a hierarchical leaderless atomic broadcast algorithm with destination agreement. The algorithm is completely decentralized and all processes can send messages concurrently. The processes use spanning trees built on the VCube overlay network which reconfigures itself autonomically after the occurrence of failures. Processes fail by crashing. Trees are used to propagate local sequence numbers (timestamps) of the messages. Each process, after receiving timestamps from all the others, decides on the order in which messages are delivered. Simulation results show the advantage of the proposed solution in terms of both the number of messages and the broadcast latency when compared to a traditional all-for-all approach.*

Resumo. *Este trabalho apresenta um algoritmo hierárquico de difusão atômica sem-líder com acordo no destino, i.e. a ordem de entrega das mensagens é definida pelos processos destinatários. O algoritmo é totalmente descentralizado e todos os processos transmitem suas mensagens concorrentemente. Os processos utilizam árvores de difusão construídas através da rede de sobreposição VCube que, após a ocorrência de falhas, se reconfigura autonomicamente. Processos podem falhar por parada. As árvores são utilizadas para propagar os números de sequência (timestamps) locais das mensagens. Cada processo, após receber os timestamps de todos os demais, decide pela ordem de entrega das mensagens. Resultados de simulação mostram a maior eficiência da solução proposta em termos tanto do número de mensagens quanto da latência de difusão quando comparada com uma abordagem tradicional todos-para-todos.*

1. Introdução

A difusão atômica (*atomic broadcast*), também conhecida como difusão com ordem total, é um dos problemas fundamentais em sistemas distribuídos tolerante a falhas [Poke et al. 2017, Kshemkalyani and Singhal 2008]. Fornecida como uma primitiva de comunicação, a difusão atômica é capaz de oferecer consistência forte ao mesmo tempo em que tolera falhas de processos [Poke et al. 2017], equiparando-se ao problema do consenso [Chandra and Toueg 1996]. É ainda um dos pilares para projetar, por exemplo,

[*]Instituto Federal do Paraná (IFPR) – Cascavel – PR – Brasil
[†]Universidade Tecnológica Federal do Paraná (UTFPR) – Toledo – PR – Brasil
[‡]Universidade Federal de Santa Maria (UFSM) – Santa Maria – RS – Brasil
[§]Universidade Federal do Paraná (UFPR) – Curitiba – PR – Brasil
[¶]Sorbonne Universités/UPMC/CNRS/Inria – Paris – França

abordagens de replicação máquina de estado [Schneider 1990, Saramago et al. 2017] e bases de dados replicadas [Pedone et al. 1998].

Informalmente, a difusão atômica estende a difusão confiável, que garante a entrega das mensagens enviadas por um processo correto a todos os processos corretos, incluindo a propriedade de ordem total [Défago et al. 2004]. A *ordem total* determina que todos os processos corretos devem entregar o mesmo conjunto de mensagens recebidas na mesma ordem.

A difusão atômica pode ser construída através de uma abordagem chamada de acordo no destino (*destination agreement*). Nessa abordagem, a ordem de entrega das mensagens resulta de um acordo entre os processos de destino [Défago et al. 2004] e pode ser dividida em três variantes: (1) acordo sobre um conjunto de mensagens; (2) acordo sobre a aceitação de uma ordem de mensagem proposta ou; (3) acordo sobre um número de sequência. Tanto a primeira quanto a segunda variantes são implementadas utilizando um algoritmo de consenso, como Paxos [Lamport 1998], e protocolos de confirmação atômica, como os propostos em [Luan and Gligor 1990, Thanisch 2000]. Geralmente, essas duas variantes elegem um líder que atua como coordenador nas execuções do consenso sobre a ordem de entrega das mensagens. Já a terceira variante opera de forma descentralizada ao permitir que os processos de destino cheguem a um acordo em um número de sequência único (mas não consecutivo) para cada mensagem a ser entregue. Apesar de exigir maior número de mensagens para sua implementação, essa abordagem se destaca por permitir que cada processo difunda mensagens concorrentemente e sem impactar a escalabilidade representada pelo gargalo de um coordenador central.

Este trabalho apresenta um algoritmo de difusão atômica totalmente descentralizado, ou seja, todos os processos transmitem suas mensagens concorrentemente através de árvores dinamicamente construídas. O processamento é compartilhado igualmente entre os processos e não há a necessidade de eleição de um líder que poderia representar um gargalo e impactar na escalabilidade de todo o sistema. A ordem de entrega das mensagens é definida através de um número de sequência (também chamado de *timestamp*) definido a partir do relógio lógico local de cada processo. Além disso, perante falhas de processos a árvore é reconstruída sem a necessidade de mensagens adicionais. As árvores são construídas e mantidas sobre a topologia virtual VCube. O VCube [Duarte et al. 2014] é um hipercubo quando todos os processos estão sem-falhas, mas se reconfigura autonomicamente conforme processos falham, mantendo várias propriedades logarítmicas.

A literatura apresenta diversos algoritmos de difusão propostos sobre o VCube [Rodrigues et al. 2014a, Rodrigues et al. 2014b, Jeanneau et al. 2017, de Araujo et al. 2019], mas este é o primeiro a explorar a difusão atômica. O algoritmo proposto é simulado e comparado com uma abordagem de acordo no destino onde um processo envia uma mensagem a todos os demais diretamente. Resultados demonstram que o algoritmo proposto supera a versão todos para todos tanto em latência quanto em número de mensagens conforme o número de processos aumenta.

O restante deste artigo segue organizado da seguinte forma. A Seção 2 apresenta os trabalhos relacionados. O modelo do sistema é apresentado na Seção 3. A Seção 4 apresenta o algoritmo proposto. Os resultados de simulação são apresentados na Seção 5. Por fim, a conclusão é apresentada na Seção 6.

2. Trabalhos relacionados

O primeiro algoritmo de difusão sobre o VCube foi proposto em [Rodrigues et al. 2014a]. No trabalho são utilizados algoritmos de difusão por melhor-esforço e confiável, executando sobre um modelo de sistema síncrono. O trabalho explora a topologia de comunicação do VCube, onde os algoritmos utilizam árvores construídas dinamicamente por cada processo. Desta forma, ao ser detectada uma falha, cada processo reconstrói a sua árvore com base nos processos corretos. Para detecção de falhas, os autores utilizaram as informações providas pelo algoritmo de diagnóstico do próprio VCube.

Em um trabalho posterior [Jeanneau et al. 2017], os autores usaram uma abordagem semelhante para criar um algoritmo hierárquico de difusão confiável para sistemas assíncronos. Para garantir a entrega aos processos suspeitos, as mensagens da aplicação continuam sendo enviadas por meio de mensagens especiais, que não são propagadas na árvore do processo considerado falho.

Recentemente, o trabalho de [Terra et al. 2020] utiliza a difusão de melhor esforço, definida em [Rodrigues et al. 2014a] para implementar uma instância do algoritmo de consenso Paxos. Ou seja, a disseminação de mensagem ocorre de acordo com a árvore de difusão. Devido à organização do VCube em grupos progressivamente maiores, as mensagens são enviadas para o maior grupo provido na topologia virtual, que consiste de $n/2$ processos. Com isso, é reduzido o custo para implementação do Paxos, com menor número de mensagens trocadas entre os processos, mesmo em cenários com uma quantidade elevada de falhas.

Um outro trabalho semelhante a solução apresentada neste artigo, foi proposto em [Poke et al. 2017]. No trabalho os autores implementam um algoritmo de difusão atômica denominado de *AllConcur*. Ao invés de usar o VCube, o algoritmo utiliza um dígrafo G como rede sobreposta. Resumidamente, o funcionamento básico do algoritmo consiste em: inicialmente, cada processo correto realiza a difusão de uma mensagem m; registra as mensagens utilizando um mecanismo denominado de *early termination* e, quando termina de acompanhar todas as informações, realiza a entrega das mensagens de forma ordenada e determinística. Ao utilizar a abordagem sem líder o desempenho obtido foi 17x maior do que o Paxos, algoritmo comparado nos experimentos. Diferentemente do *AllConcur*, nossa solução utiliza uma árvore autoajustável, isto é, que se reconfigura automaticamente na presença de falhas. Além disso, em situações de falhas em processos, nosso serviço não necessita disseminar a ocorrência aos demais participantes – sem transmitir mensagens adicionais – como ocorre no *AllConcur*.

Devido a grande importância dos algoritmos para a difusão atômica, é possível notar na literatura diversas propostas com diferentes soluções e melhorias. As propostas frequentemente consideram um líder, responsável por gerenciar e ordenar todas as mensagens [Ekwall et al. 2004, Srivastava et al. 2014]. Há também soluções para difusão atômica que utilizam uma rede sobreposta [Kozhaya et al. 2019, Paznikov et al. 2020] resultando em uma comunicação que segue a ordem determinada pelo algoritmo, entre outras variações que são projetadas para tolerar qualquer tipo de falhas [Correia et al. 2006, Milosevic et al. 2011]. Há ainda soluções que não necessitam exclusivamente da existência de um líder [Poke et al. 2017, Paznikov et al. 2020], como também soluções específicas, destinadas para sistemas de tempo real [Delporte-Gallet and Fauconnier 1999], sistemas embarcados [Rufino et al. 1998], *blockchain* [Hao et al. 2020], entre outras.

Por fim, nas abordagens onde o acordo é realizado no destino, uma estratégia comumente explorada é usar registros de tempos denominados de *timestamps*. Em [Birman and Joseph 1987] os autores propõem o uso de *timestamps* como uma das primeiras versões do *Isis Tollkit*. O algoritmo proposto executa de maneira sincronizada e respeitando a ordem de um *timestamp* global. Para realizar um *broadcast* m, um emissor envia m para todos os destinatários. Ao receber m, um destinatário atribui um *timestamp* local e encaminha este *timestamp* para todos os demais destinatários. Uma vez que os processos têm recebido o *timestamp* local de todos os outros destinatários, um único *timestamp* global é atribuído para a mensagem m, com o máximo valor de todos os *timestamps* recebidos. Dessa forma, os processos seguem a ordem global para entregar todas as mensagens encaminhadas no sistema. Neste trabalho uma abordagem similar é proposta e detalhada nas próximas seções.

3. Modelo do Sistema

O sistema consiste de um conjunto finito P com $n > 1$ processos $\{p_0, .., p_{n-1}\}$. A comunicação entre os processos é realizada através de troca de mensagens. A topologia física corresponde a um grafo completo, isto é, cada processo pode enviar e receber mensagens de qualquer outro processo sem a necessidade de intermediários. Os processos são organizados utilizando uma topologia de hipercubo virtual detalhada a seguir. Os enlaces são confiáveis, assim, mensagens trocadas entre dois processos nunca são perdidas, corrompidas ou duplicadas. O sistema é síncrono, ou seja, existe um limite máximo para a velocidade dos processos e transmissão de mensagens. Processos podem falhar por parada (*crash*) e não se recuperam (*recover*). Se um processo nunca falha durante a sua execução, então é considerado correto ou sem-falha; ao contrário o processo é considerado falho.

3.1. Topologia Virtual

A rede de sobreposição adotada neste trabalho é implementada utilizando o VCube [Duarte et al. 2014]. Quando todos os processos estão corretos, o VCube é um hipercubo. Após a ocorrência de falhas o VCube se reconfigura autonomicamente, mantendo diversas propriedades logarítmicas.

Para criar a topologia virtual (Figura 1), os vizinhos do processo i no *cluster* s são retornados pela função $c_{i,s} = \{i \oplus 2^{s-1}, c_{i\oplus 2^{s-1},1}, \dots, c_{i\oplus 2^{s-1},s-1}\}$. O símbolo $\oplus$ corresponde à operação bit-a-bit ou exclusivo (*xor*). A Tabela 1 apresenta a função $c_{i,s}$ de um VCube de três dimensões ($n = 8$). Por exemplo, de acordo com a Figura 1 e a Tabela 1 o processo 0 possui os vizinhos 1, 2 e 4, nos *clusters* 1, 2 e 3, respectivamente.

3.1.1. Funções Auxiliares

Com base na topologia do VCube, a função $FF\_neighbor_i(s) = j$ identifica o primeiro processo correto j ($j \in correct_i$) no cluster s do processo i. Por exemplo, para um VCube de três dimensões sem-falhas, $FF\_neighbor_0(1) = 1$, $FF\_neighbor_0(2) = 2$ e $FF\_neighbor_0(3) = 4$. Em caso de falha do processo p_2, $FF\_neighbor_0(2) = 3$. A falha de p_1 implica em $FF\_neighbor_0(1) = \emptyset$.

A função $neighborhood_i(d)$, definida como

$$neighborhood_i(d) = \{k | k = FF\_neighbor_i(s), 1 \leq s \leq d\} \quad (1)$$

s	C0,s	C1,s	C2,s	C3,s	C4,s	C5,s	C6,s	C7,s
1	1	0	3	2	5	4	7	6
2	2 3	3 2	0 1	1 0	6 7	7 6	4 5	5 4
3	4 5 6 7	5 4 7 6	6 7 4 5	7 6 5 4	0 1 2 3	1 0 3 2	2 3 0 1	3 2 1 0

Tabela 1. *Clusters* do VCube com 3 dimensões.

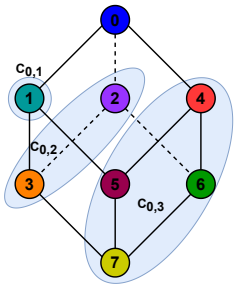


Figura 1. Topologia do VCube com três dimensões e *clusters* do processo 0.

é utilizada para calcular os vizinhos de um processo i considerando o nível d da árvore de difusão. Se i é a raiz, a função retorna todos os vizinhos sem-falha em cada cluster $s = 1..d$, $d = \log_2(n)$ (a dimensão do hipercubo). Se i é raiz de uma sub-árvore do processo fonte que originou m, a função calcula os vizinhos considerando os clusters internos que ele tem conexão no nível d. Por exemplo, para um hipercubo de três dimensões sem falhas com difusão de uma mensagem pelo processo p_0, $neighborhood_0(d = \log_2(n)) = \{1, 2, 4\}$. No processo p_4, que está no nível $d = 2$, $neighborhood_4(2) = \{5, 6\}$.

A função $cluster_i(j) = s$ é utilizada para determinar o cluster s que um processo i pertence em relação a um processo j. Esta informação é importante para determinar: a) o nível de i na árvore de j; b) o cluster de um processo j detectado como falho para o qual deve ser feita uma retransmissão. Como exemplo, $cluster_4(0) = 3$. Portanto, sub-árvore a partir de p_4 tem 2 níveis.

4. O Algoritmo Hierárquico de Difusão Atômica

A difusão atômica é uma difusão confiável que garante a ordem total de entrega de mensagens, ou seja, todos os processos corretos entregam o mesmo conjunto de mensagens na mesma ordem. Para tanto, as seguintes propriedade devem ser garantidas [Défago et al. 2004]:

- Validade: se um processo correto envia uma mensagem m, então ele entrega m;
- Integridade: qualquer mensagem m é entregue apenas uma vez e apenas se foi previamente enviada (não-criação);
- Acordo: se um processo correto entrega m, então todos os processos corretos também o fazem;
- Ordem total: se dois processos corretos p e q entregam duas mensagens m_1 e m_2 então p entrega m_1 antes de m_2 se e somente se q entrega m_1 antes de m_2.

O algoritmo de difusão atômica proposto neste trabalho transmite suas mensagens através da topologia virtual, descrita na Seção 3. Cada processo, ao difundir uma mensagem m, utiliza uma árvore geradora com raiz em si mesmo para encaminhar um contador lógico (*timestamp*) com a finalidade de ordenar a entrega das mensagens. As árvores são dinamicamente construídas e autonomicamente mantidas utilizando a estrutura hierárquica dos grupos e o conhecimento de processos falhos. Uma vez que um processo detecta a falha de outro processo, sua árvore é reconfigurada sem a necessidade de mensagens adicionais.

4.1. Descrição do Algoritmo

Em linhas gerais o algoritmo funciona da seguinte forma: processo de origem (fonte) que deseja propagar uma mensagem, a envia para seus vizinhos (seus filhos na árvore). Ao receber a mensagem, cada processo se torna a raiz de uma sub-árvore e a encaminha para seus próprios grupos considerando a árvore geradora da origem. Depois que uma mensagem é propagada com sucesso em suas sub-árvores, o processo envia uma mensagem de confirmação (ACK) para o processo pai, ou seja, o processo do qual a mensagem foi recebida. Desta forma, um processo envia um ACK somente se recebeu confirmações de todos os seus processos filhos. Assim, ao receber ACK de um processo filho, é possível afirmar que todos os processos corretos daquele *cluster* também receberam a mensagem. Além disso, cada processo, ao receber uma mensagem pela primeira vez, atualiza o *timestamp* local e o replica em uma árvore com raiz nele próprio. Uma mensagem só é entregue ao receber o *timestamp* de todos os processos corretos para a mensagem propagada.

O Algoritmo 1 apresenta o pseudocódigo da solução. As variáveis locais mantidas pelos processos são:

- LC_i: relógio lógico local que identifica unicamente as mensagens enviadas por i. É usado para controlar quais mensagens já foram entregues de um determinado processo fonte;
- TS_i: *timestamp* utilizado para a ordem total;
- $correct_i$: conjunto dos processos considerados corretos pelo processo i;
- $last_i[n]$: a última mensagem recebida e entregue de cada processo fonte em ordem;
- $pending\_ack_i$: o conjunto de ACKs pendentes no processo i. Para cada mensagem $TREE\langle m, T\rangle$ (re)-transmitida por i de um processo j para um processo k, um elemento $\langle j, k, \langle m, T\rangle\rangle$ é adicionado a este conjunto;
- $received_i$: conjunto de mensagens $\langle m, ts_k(m)\rangle$ recebidas pelo processo i dos processos k que ainda não podem ser entregues à aplicação;
- $stamped_i$: conjunto de mensagens m para as quais já foram recebidos todos os $timestamps$ $\langle m, ts_k(m)\rangle$, mas que ainda não podem ser entregues porque estão fora de ordem em relação às demais mensagens $m' \neq m$.

As mensagens utilizadas no algoritmo são:

- $TREE\langle m, T\rangle$, que contém a informação m sendo propagada e o conjunto T de *timestamps* $ts(m)$. T tem de 1 até $log_2 n$ *timestamps*. Para identificar a qual processo k um *timestamp* pertence, a notação utilizada é $ts_k(m)$;
- $ACK\langle m, T\rangle$, para confirmar a recepção de uma mensagem $TREE\langle m, T\rangle$.

Toda mensagem m possui dois parâmetros: (i) o identificador do processo de origem $m.src$; e (ii) um número de sequência $m.seq$, obtido do relógio lógico local LC_i do processo de origem i, utilizado para identificar, de maneira única e sequencial, as mensagens enviadas por cada processo.

A difusão é iniciada com a função A-BROADCAST(m) (linha 7), que adiciona à mensagem o *timestamp* $ts_i(m)$ do processo de origem. A função FORWARD é utilizada para encaminhar $TREE$ aos vizinhos do processo i. Os parâmetros da função FORWARD são o número de *clusters* s de i que a mensagem deverá ser encaminhada e a mensagem contendo o $ts_i(m)$. Ao iniciar a difusão, o processo i utiliza $s = \log_2 n$, que representa

Algorithm 1 Difusão Atômica no Processo i

1: **procedure** INITIALIZATION( )
2: $LC_i \leftarrow TS_i \leftarrow 0$
3: $correct_i \leftarrow \{0, .., n-1\}$
4: $last_i[n] \leftarrow \{\perp, .., \perp\}$
5: $pending\_ack_i \leftarrow \emptyset$
6: $received_i \leftarrow stamped_i \leftarrow \emptyset$

7: **procedure** A-BROADCAST(m)
8: $m.src \leftarrow i$; $m.seq \leftarrow LC_i$
9: $ts_i(m) \leftarrow TS_i$
10: $LC_i \leftarrow LC_i + 1$
11: $TS_i \leftarrow$MAX(TS_i, LC_i)
12: $received_i \leftarrow received_i \cup \{\langle m, ts_i(m)\rangle\}$
13: FORWARD($\log_2 n, TREE\langle m, \{ts_i(m)\}\rangle$)

14: **procedure** FORWARD($s, TREE\langle m, T\rangle$)
15: **for all** $p \in neighborhood_i(s)$ **do**
16: SEND($TREE\langle m, T\rangle$) **to** p
17: $pending\_ack_i \leftarrow pending\_ack_i \cup \{\langle from, p, \langle m, T\rangle\rangle\}$

18: **upon** `receive` $TREE\langle m, T\rangle$ **from** p_j
19: **if** $j \notin correct_i$ **then**
20: RETURN
21: $TS_i \leftarrow$ MAX($\forall ts_k(m) \in T, TS_i + 1$)
22: **if** $m.seq > last_i[m.src]$ **and** $m \notin \{received_i \cup stamped_i\}$ **then**
23: $ts_i(m) \leftarrow TS_i$
24: $T \leftarrow T \cup \{ts_i(m)\}$
25: **for all** $p \in neighborhood_i(\log_2 n) \setminus neighborhood_i(cluster_i(j)-1)$ **do**
26: SEND($TREE\langle m, \{ts_i(m)\}\rangle$) **to** p
27: $pending\_ack_i \leftarrow pending\_ack_i \cup \{\langle i, p, \langle m, ts_i(m)\rangle\rangle\}$
28: **for all** $ts_k(m) \in T$ **do**
29: $received_i \leftarrow received_i \cup \{\langle m, ts_k(m)\rangle\}$
30: FORWARD($cluster_i(j)-1, TREE\langle m, T\rangle$)
31: CHECKDELIVERABLE(m)
32: CHECKACKS(j, $\langle m, T\rangle$)

33: **upon** `receive` $ACK\langle m, T\rangle$ **from** p_j
34: $p \leftarrow x : \langle x, j, \langle m, T\rangle\rangle \in pending\_ack_i$
35: $pending\_ack_i \leftarrow pending\_ack_i \setminus \langle p, j, \langle m, T\rangle\rangle$
36: CHECKDELIVERABLE(m)
37: CHECKACKS(p, $\langle m, T\rangle$)

38: **procedure** CHECKDELIVERABLE(m)
39: **if** recebeu $\langle m, ts_k(m)\rangle$ de todo $k \in correct_i$ **and** $pending\_ack_i \cap \langle *, *, \langle m, *\rangle\rangle = \emptyset$ **then**
40: $sn(m) \leftarrow$ MAX($ts_k(m) \mid \langle m, ts_k(m)\rangle \in received_i$)
41: DELIVER(m, $sn(m)$)
42: **for all** $\langle m' = m, ts(m)\rangle \in sorted(received_i)$ **do**
43: **if** $m'.seq = last_i[m.src] + 1$ **then**
44: $last_i[m.src] \leftarrow m'$
45: $received_i \leftarrow received_i \setminus \{\langle m', *\rangle\}$

46: **procedure** DELIVER(m, $sn(m)$)
47: $stamped_i \leftarrow stamped_i \cup \{\langle m, sn(m)\rangle\}$
48: $deliverable \leftarrow \emptyset$
49: **for all** $\langle m', sn(m')\rangle \in stamped_i \mid \forall \langle m'', ts(m''\rangle \in received_i : sn(m') < ts(m'')$ **do**
50: $deliverable \leftarrow deliverable \cup \{\langle m', sn(m')\rangle\}$
51: Entrega todas as mensagens em $deliverable$ na ordem $(sn(m), m.src)$
52: $stamped_i \leftarrow stamped_i \setminus deliverable$

53: **procedure** CHECKACKS(p, $\langle m, T\rangle$)
54: **if** $pending\_ack_i \cap \langle p, *, \langle m, T\rangle\rangle = \emptyset$ **then**
55: **if** $m.src \neq i$ **and** $\{p\} \in correct_i$ **then**
56: SEND($ACK\langle m, T\rangle$) **to** p

57: **upon notifying** `crash`(j)
58: $correct_i \leftarrow correct_i \setminus \{j\}$
59: **for all** $p = x, m = y : \langle x, j, \langle y, T\rangle\rangle \in pending\_ack_i$ **do**
60: **if** $\{p\} \in correct_i$ **then**
61: **if** $k = FF\_neighbor_i(cluster_i(j)) \neq \emptyset$ **then**
62: SEND($TREE\langle m, T\rangle$) **to** p
63: $pending\_ack_i \leftarrow pending\_ack_i \cup \langle p, k, \langle m, T\rangle\rangle$
64: $pending\_ack_i \leftarrow pending\_ack_i \setminus \langle p, j, \langle m, T\rangle\rangle$
65: CHECKACKS(p, $\langle m, T\rangle$)
66: **for all** $\langle m, *\rangle \in received_i$ **do**
67: CHECKDELIVERABLE(m)

a dimensão do hipercubo e corresponde ao número de *clusters* que cada processo possui. Na função FORWARD, a mensagem m é encaminhada aos processos vizinhos de i com base no valor de s (linha 15).

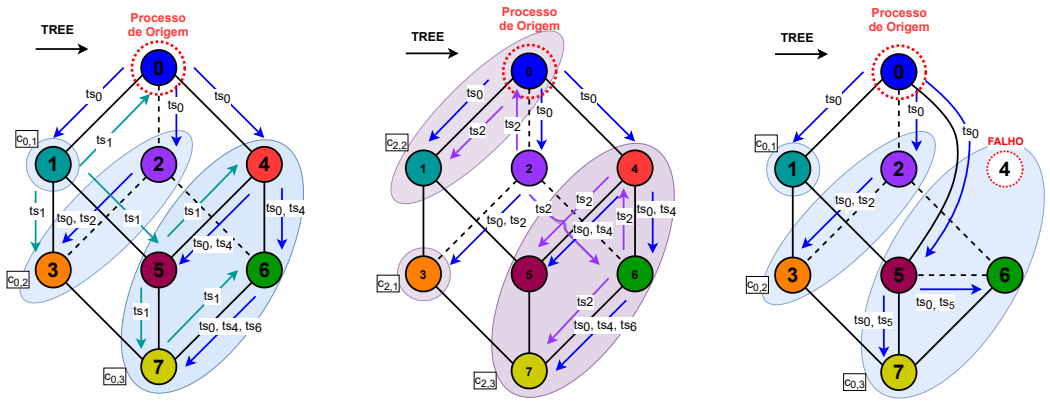


Figura 2. Encaminhamento de mensagens – exemplos ilustrativos.

Ao receber a mensagem $TREE$, i verifica se j está na lista de processos corretos (linha 19). Se estiver falho, seu *timestamp* deve ser desconsiderado. Se j estiver correto, o processo i atualiza seu *timestamp* local TS_i com base nos *timestamps* recebidos do processo j (linha 21). Ao verificar que a mensagem foi recebida pela primeira vez, ou seja, não foi entregue e não está no conjunto de mensagens recebidas ou marcadas para entrega, a mensagem é retransmitida na árvore do processo i com $T = ts_i(m)$, aos processos que não sobrepõem a árvore do processo j (linha 25). Para os processos que sobrepõem a árvore de j a mensagem m é encaminhada com *timestamp* T (linha 30), tal que se m foi recebida pela primeira vez, $ts_i(m)$ é agregado a T (linha 24).

Considerando um VCube de três dimensões, a Figura 2 apresenta o esquema de difusão de mensagens do processo p_0 em sua árvore (ver seta azul escura) e o encaminhamento da mensagem m aos vizinhos dos *clusters* 1, 2 e 3. A Figura 2a destaca o comportamento do processo p_1 ao receber a mensagem m pela primeira vez. O processo p_1, identifica que está no último nível da árvore de p_0 (folha), através da função $\{cluster_1(0) - 1 = 0\}$. Desta forma, p_1 apenas encaminha seu *timestamp* aos processos $\{0, 3, 5\}$ (representado por ts_1 nas setas). A Figura 2b destaca o processo p_2, que é um ramo na árvore de p_0 ($cluster_2(0) - 1 = 1$). Com isso, encaminha m com seu *timestamp* aos processos $\{0, 6\}$. Na árvore do processo p_0, p_2 agrega seu próprio *timestamp* a m e encaminha ao processo p_3. Assim m passa a conter os *timestamps* de p_0 e p_2.

Ao receber a mensagem $TREE$ e encaminhar as mensagens aos próximos processos, cada *timestamp* contido em T é adicionado à lista de *timestamps* recebidos de m (linha 29) e é chamada a função CHECKDELIVERABLE(m). Esta função verifica se os *timestamps* de todos os processos corretos do sistema foram recebidos para m (linha 39). Caso seja tenham sido recebidos, é obtido o maior *timestamp* para m (linha 40), representado por $sn(m)$ e chamada a função DELIVER($m, sn(m)$) (linha 41). Em seguida, atualiza-se o registro $last_i$ de mensagens do processo de origem da mensagem e removem-se todos os *timestamps* associados à m da lista de mensagens recebidas $received_i$.

Na função DELIVER, a mensagem m é adicionada à lista de mensagens marcadas para entrega ($stamped_i$) com o maior *timestamp* recebido $sn(m)$ (linha 47). A mensagem é adicionada para entrega se, e somente se, para toda mensagem m' em $stamped_i$ não existir mensagem m'' recebida ($received_i$) com *timestamp* menor que m'. Caso contrário, m' é entregue somente após m'' (linha 49). As mensagens marcadas para entrega ($deliverable$) são ordenadas pelo *timestamp* associado à m e o processo de origem de m ($m.src$) (linha 51). Desta forma, as mensagens são entregues ordenadas e removidas de $stamped_i$.

A confirmação de recebimento de mensagens é realizada através da função CHECKACKS. A tupla $\langle p, *, \langle m, T \rangle \rangle$ expressa os ACKs que estão sendo aguardados para a mensagem m com *timestamp* T, recebidas de p e encaminhadas a qualquer processo ($*$) (linha 54). Por exemplo, na Figura 2a, p_1 é uma folha da árvore de p_0, portanto, não encaminhará nenhuma mensagem com o *timestamp* de p_0, o que permite o envio imediato do ACK para p_0. O mesmo ocorre com p_3 em relação à p_2. No entanto, p_2, ao receber o ACK de p_3, remove a mensagem pendente de sua lista e obtém o processo p que inicialmente enviou a mensagem com *timestamp* T. Com isso, ao executar CHECKACKS e verificado que não existe mensagem pendente de p_0, que deveria ter sido encaminhada por p_2, é enviado um ACK para p_0.

A detecção e notificação de falhas em processos é realizada pelo algoritmo de diagnóstico do VCube. Para todos os processos, quando i recebe a notificação de que j está falho (linha 58), j é removido da lista de processos corretos. Com isso, para cada mensagem m encaminhada a j não confirmada, ou seja, existe m enviada a j em $pending\_acks_i$ (linha 59), estas mensagens devem ser reencaminhadas ao próximo processo correto, se houver, no *cluster* a que j pertence (a função $FF\_neighbor_i(s)$ determina esta informação, sendo $s = cluster_i(j)$) (linha 61). Por exemplo, na Figura 2c em que p_0 é a origem da mensagem em um hipercubo de três dimensões, se p_4 falhar, durante a difusão da mensagem, p_0 não receberá a confirmação de recebimento do grupo três e, ao ser detectada a falha, encaminhará seu *timestamp* ao próximo processo correto deste *cluster* (p_5). Se não houver mais processos corretos naquele *cluster*, a função CHECKACKS é chamada para verificar se existe alguma mensagem pendente a ser encaminhada do processo p que enviou m ao processo i (linha 65). Além disso, o processo i pode não ter recebido o *timestamp* de j, logo ficará aguardando até que seja recebido. No entanto, j foi detectado como falho. Portanto, para cada mensagem m em $received_i$ é chamada a função CHECKDELIVERABLE (linha 67) para verificar se m pode ser entregue, visto que o *timestamp* de j não é mais necessário.

5. Resultados de Simulação

Esta seção apresenta os resultados de simulação em relação à latência e número de mensagem na execução do algoritmo proposto em cenários com e sem falhas. Os resultados foram obtidos utilizando o simulador Neko [Urban et al. 2001]. O número de mensagens corresponde a todas as mensagens trocadas entre os processos, incluindo as confirmações de entrega (*ACKs*). A latência corresponde à t unidades de tempo para que uma mensagem seja entregue a todos os processos corretos.

A estratégia hierárquica de difusão de mensagens foi comparada com uma estratégia todos-para-todos (chamada All2All). Na estratégia todos-para-todos, cada processo

envia o *timestamp* (ts) para a mensagem recebida a $n-1$ processos, assim cada processo comunica-se diretamente com os demais processos. Foram considerados diferentes cenários e tamanhos de sistema. Em cada experimento, o número de processos n variou de 8 a 1024 em potência de 2. Ao utilizar a estratégia All2All o detector de falhas se comunica diretamente com os processos e, portanto, possui menor latência para detecção de falhas. A estratégia de difusão hierárquica de mensagens conta com a estrutura do próprio VCube, que proporciona maior escalabilidade ao custo de aumentar a latência para detecção de falhas.

Nos experimentos, considera-se que cada processo envia somente uma mensagem por vez com um atraso t_s. Ainda, um processo leva t_r unidades de tempo para receber uma mensagem. Se uma mensagem é enviada para mais do que um processo, t_s deve ser computado para cada cópia da mensagem a ser enviada. O tempo de transmissão na rede de uma mensagem é dado por t_t. Desta forma, definimos que $t_r = t_s = 0.1$ e $t = 0.8$ unidades de tempo para todas as execuções. O intervalo de teste do detector em cada experimento foi definido para 30.0 unidades de tempo. Um processo é considerado falho se o detector não obtiver a resposta em 4 * $(t_s + t_t + t_s)$[1] unidades de tempo. A seguir, são apresentados os resultados dos experimentos sem falhas e com falhas de processos.

5.1. Experimentos sem processos falhos

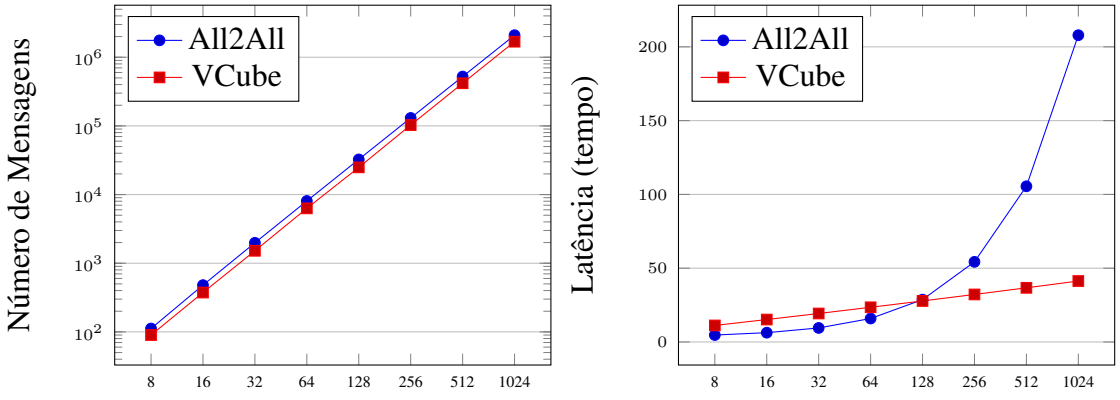
A Figura 3 apresenta o número de mensagens e a latência do envio de uma única mensagem pelo processo p_0. Na estratégia hierárquica, ao enviar a mensagem e seu *timestamp*, o caminho mais longo em uma sub-árvore gerada no VCube tem $\log_2 n$ arestas. Desta forma, no pior caso, um processo leva $2log_2 n$ para encaminhar seu *timestamp*, visto que é necessário o recebimento de *ACKs* para garantir que o *timestamp* e a mensagem foram recebidos. A entrega de uma mensagem é determinada pelo tempo que o último processo da maior sub-árvore de p_0 leva para encaminhar seu *timestamp*. Na estratégia *All2All*, a latência é dada pelo tempo que um processo leva para encaminhar seu *timestamp* aos demais processos após receber uma mensagem. Com isso, para poucos processos a latência da estratégia hierárquica é maior. No entanto, quando o número de processos é maior ou igual a 128, a estratégia hierárquica apresenta menor latência (ver Figura 3b). Desta forma, quanto maior o número de processos maior será o tempo para o envio das mensagens. A latência na difusão hierárquica utilizando o VCube mostrou-se linear. Já a latência da estratégia *All2All* apresentou comportamento exponencial.

A ordem total das mensagens é garantida com o recebimento do *timestamp* de todos os processos para a mensagem m. Ao utilizar a estratégia *All2All* todos os processos enviam diretamente o seu *timestamp* aos demais processos. Com a estratégia hierárquica, é possível que uma mensagem agregue mais do que um *timestamp*. Desta forma, o número de mensagens no algoritmo de difusão atômica proposto foi em média 21.45% menor (Figura 3a).

5.2. Experimentos com processos falhos

A seguir são apresentados resultados para a latência e número de mensagens a partir de três cenários distintos de falhas de processos. O primeiro cenário apresenta a falha do processo de origem da mensagem. O segundo cenário descreve o impacto da falha de um

[1]Este valor representa o dobro do tempo para o envio até a recepção de uma mensagem.



(a) Mensagens **(b) Latência**

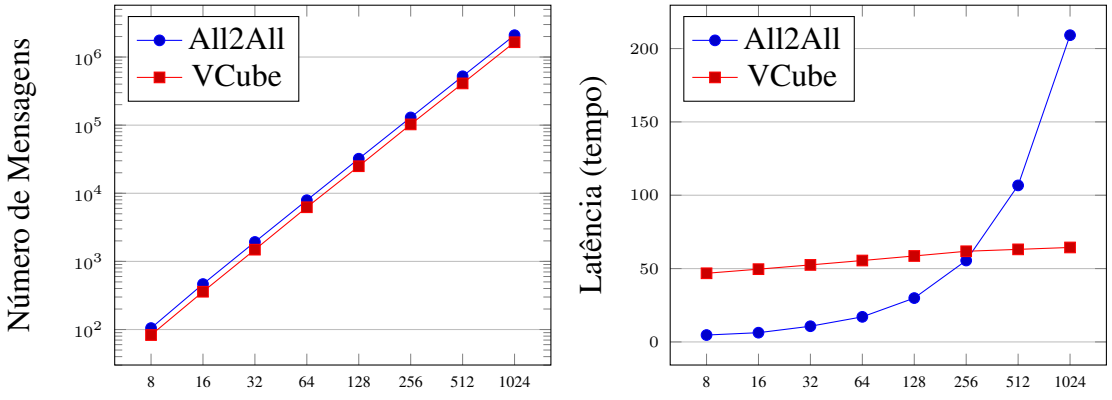
Figura 3. Execução sem processos falhos.

processo qualquer do sistema, antes de ter sido detectado como falho. Por fim, no terceiro cenário o algoritmo de difusão é executado após a falha de um processo ser detectada por todos os demais processos do sistema.

Execução com falha da origem. Considera-se que ao menos um processo correto recebeu a mensagem antes do emissor falhar, conforme a propriedade de terminação (*liveness*) herdada da difusão confiável. No algoritmo de difusão atômica proposto a falha da origem implica em todos os processos que receberam a mensagem desconsiderarem o *timestamp* do processo falho. Além disso, como o algoritmo utiliza difusão hierárquica de mensagens, os processos que não recebem a confirmação de recebimento das mensagens enviadas ao processo falho, devem reencaminhar a mesma mensagem ao próximo processo correto no *cluster* do processo falho, se houver. Para execução do experimento considerou-se a origem p_0 enviando mensagens no tempo 0 e falhando logo em seguida.

Conforme apresenta a Figura 4a, houve pequena variação no número de mensagens, pois a falha de um processo livra-o de obter os *timestamps*. Desta forma, o número de mensagens da abordagem hierárquica foi, em média, 21.74% menor em relação à topologia *All2All*, resultado semelhante à execução sem falhas. Já a latência para entrega de mensagens, ilustrada na Figura 4b, apresenta uma diferença significativa na presença de falhas. O algoritmo de diagnóstico do VCube executa em rodadas. Assim, para todos os processos corretos identificarem uma falha é necessária a execução de mais do que uma rodada do algoritmo. Desta forma, o intervalo para execução de rodadas do detector de falhas, definido em 30 unidades de tempo, impacta diretamente na latência para entrega de mensagens. A diferença de tempo para entrega de mensagens entre a abordagem hierárquica sem falha e na presença de uma falha é, em média, 29.63 unidades de tempo. Em relação à abordagem *All2All*, a partir de 512 processos apresentou maior latência para entrega de mensagens, visto que o tempo de envio e recebimento (t_s e t_r) das mensagens é mais significativo quando o número de processos é maior.

Execução com envio antes de processo ser detectado como falho. Considera-se a falha do processo p_1 durante a difusão de uma mensagem m por p_0. Um processo ao encaminhar m à um processo falho, nunca receberá a confirmação de recebimento deste processo e acabará por detectá-lo como falho. A falha de um processo qualquer, assim como a falha



(a) Mensagens **(b) Latência**

Figura 4. Execução com falha do processo de origem.

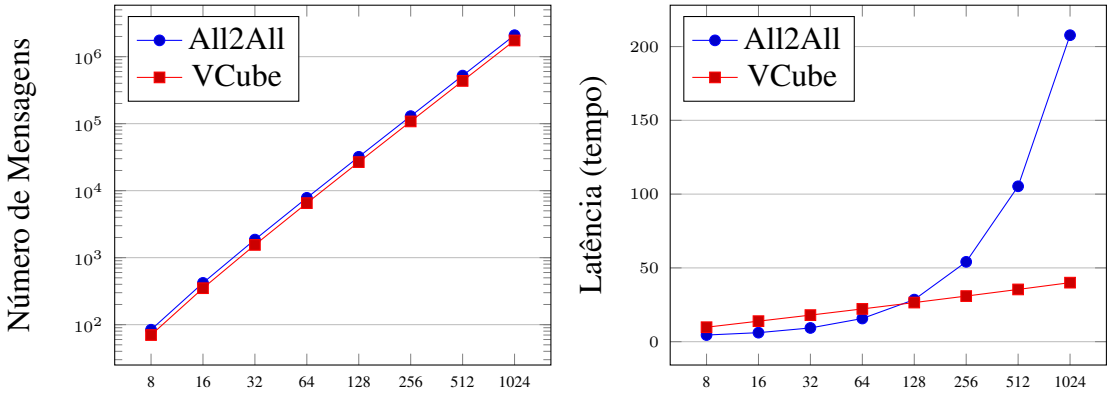
da origem, implica na remoção do *timestamp* do processo falho na variável $received_i$ e o reencaminhamento das mensagens com *ACKs* pendentes ao próximo processo correto do grupo, se houver. Desta forma, a latência e o número de mensagens para que uma mensagem m seja entregue é idêntica à execução com falha na origem.

Execução com envio após detecção de processo falho. Neste experimento, o processo p_1 falha no tempo 0. Somente após todos os processos detectarem a falha de p_1 a mensagem é difundida pela origem (processo p_0). Com isso, o algoritmo se reorganiza para que o processo falho seja removido da topologia. Os resultados obtidos são apresentados na Figura 5 e apontam que na topologia *All2All* o número de mensagem foi menor em relação a execução sem falhas, visto que existem menos processos ativos. No entanto, a abordagem hierárquica apresenta menor número de mensagens em relação à abordagem *All2All* (Figura 5a).

Devido à redução no número de processos corretos no sistema, a latência para entrega de mensagens é menor. Na abordagem *All2All* a latência é reduzida em 0.2 unidades de tempo para todas as execuções. Na abordagem hierárquica a latência para entrega é reduzida em 1.3 unidades de tempo. Assim como na execução sem falhas, a partir de 128 processos a latência utilizando a abordagem hierárquica é menor do que a latência utilizando a topologia *All2All*.

6. Conclusão

Este trabalho apresentou uma solução hierárquica, autonômica e totalmente descentralizada para difusão atômica. A principal contribuição é uma estratégia escalável que permite aos processos entregarem as mensagens em ordem sem a necessidade de um líder. A solução proposta foi comparada com uma estratégia todos para todos (chamada de *All2All*) em que os processos se comunicam diretamente. Resultados experimentais apresentaram cenários com e sem falhas de processos. Na presença de falhas, a abordagem hierárquica aproveita as propriedades do VCube para permitir que os processos da árvore se reorganizem automaticamente. Nos experimentos sem falhas o número de mensagens da estratégia hierárquica foi significativamente menor. A partir de 128 processos a latência foi maior na estratégia *All2All*. Nos cenários com falhas, não houve diferen-



(a) Mensagens **(b) Latência**

Figura 5. Execução com *broadcast* após a detecção de uma falha.

ças significativas no número de mensagens e observou-se que a latência é diretamente influenciada pelo intervalo de execução de rodadas do detector de falhas. Ainda assim, a abordagem hierárquica apresenta menor latência conforme o número de processos aumenta e supera a abordagem All2All quando a difusão atômica apresenta mais de 256 processos, conferindo uma tendência de maior escalabilidade à solução proposta.

Trabalhos futuros incluem a investigação de estratégias para diminuir o número de mensagens e também para aproveitar as propriedades do VCube para antecipar a entrega mesmo perante falhas. Além disso, pretende-se propor um algoritmo para outros modelos de sistema, como o parcialmente síncrono, e comparar a solução com a difusão atômica oferecida pelos algoritmos de consenso baseados em líder como o Paxos e Raft.

Referências

Birman, K. P. and Joseph, T. A. (1987). Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1):47–76.

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

Correia, M., Neves, N. F., and Veríssimo, P. (2006). From consensus to atomic broadcast: Time-free byzantine-resistant protocols without signatures. *The Computer Journal*, 49(1):82–96.

de Araujo, J. P., Arantes, L., Duarte, E. P., Rodrigues, L. A., and Sens, P. (2019). Vcube-ps: A causal broadcast topic-based publish/subscribe system. *J. Parallel Distributed Comput.*, 125:18–30.

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

Delporte-Gallet, C. and Fauconnier, C. D. I. (1999). Real-time fault-tolerant atomic broadcast. In *Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems*, pages 48–55.

Duarte, E. P., Bona, L. C. E., and Ruoso, V. K. (2014). Vcube: A provably scalable distributed diagnosis algorithm. In *2014 5th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, pages 17–22.

Ekwall, R., Schiper, A., and Urban, P. (2004). Token-based atomic broadcast using unreliable failure detectors. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, 2004.*

Hao, W., Zeng, J., Dai, X., Xiao, J., Hua, Q., Chen, H., Li, K., and Jin, H. (2020). Towards a trust-enhanced blockchain p2p topology for enabling fast and reliable broadcast. *IEEE Transactions on Network and Service Management*, 17(2):904–917.

Jeanneau, D., Rodrigues, L. A., Arantes, L., and Jr., E. P. D. (2017). An autonomic hierarchical reliable broadcast protocol for asynchronous distributed systems with failure detection. *J. Braz. Comput. Soc.*, 23(1):15:1–15:14.

Kozhaya, D., Decouchant, J., and Esteves-Verissimo, P. (2019). Rt-byzcast: Byzantine-resilient real-time reliable broadcast. *IEEE Transactions on Computers*, 68(3):440–454.

Kshemkalyani, A. D. and Singhal, M. (2008). *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, USA, 1 edition.

Lamport, L. (1998). The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169.

Luan, S. . and Gligor, V. D. (1990). A fault-tolerant protocol for atomic broadcast. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):271–285.

Milosevic, Z., Hutle, M., and Schiper, A. (2011). On the reduction of atomic broadcast to consensus with byzantine faults. In *2011 IEEE 30th International Symposium on Reliable Distributed Systems*, pages 235–244.

Paznikov, A. A., Gurin, A. V., and Kupriyanov, M. S. (2020). Implementation in actor model of leaderless decentralized atomic broadcast. In *2020 9th Mediterranean Conference on Embedded Computing (MECO)*.

Pedone, F., Guerraoui, R., and Schiper, A. (1998). Exploiting atomic broadcast in replicated databases. In Pritchard, D. and Reeve, J., editors, *Euro-Par'98 Parallel Processing*, pages 513–520, Berlin, Heidelberg. Springer Berlin Heidelberg.

Poke, M., Hoefler, T., and Glass, C. W. (2017). Allconcur: Leaderless concurrent atomic broadcast. HPDC '17, page 205–218, New York, NY, USA. Association for Computing Machinery.

Rodrigues, L. A., Arantes, L., and Jr., E. P. D. (2014a). An autonomic implementation of reliable broadcast based on dynamic spanning trees. In *EDCC*, pages 1–12. IEEE Computer Society.

Rodrigues, L. A., Duarte Jr, E. P., and Arantes, L. (2014b). Árvores geradoras mínimas distribuídas e autonômicas. In *SBRC*, pages 1–14.

Rufino, J., Verissimo, P., Arroz, G., Almeida, C., and Rodrigues, L. (1998). Fault-tolerant broadcasts in can. In *Annual International Symp. on Fault Tolerant Computing - FTCS*.

Saramago, R. Q., Alchieri, E. A. P., Rezende, T. F., and Camargos, L. (2017). Algoritmo de difusão atômica rápido a despeito de colisões tolerante a falhas bizantinas. In *SBRC*, Porto Alegre, RS, Brasil. SBC.

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

Srivastava, P., Chakrabarti, P., and Panwar, A. (2014). Article: Rigorous design of moving sequencer atomic broadcast with malicious sequencer. *International Journal of Computer Applications*, 105(7):13–18.

Terra, A., Camargo, E., and Duarte, E. P. (2020). A caminho de uma alternativa hierárquica para implementação do algoritmo de consenso paxos. In *Anais do XXI W. de Testes e Tolerância a Falhas*, pages 15–28, Porto Alegre, RS, Brasil. SBC.

Thanisch, P. (2000). Atomic commit in concurrent computing. *IEEE Concurrency*, 8(4):34–41.

Urban, P., Defago, X., and Schiper, A. (2001). Neko: a single environment to simulate and prototype distributed algorithms. In *Proceedings 15th International Conference on Information Networking*, pages 503–511.