Distributed and Autonomic Minimum Spanning Trees

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

[1] Western Parana State University (UNIOESTE) – Cascavel – PR – Brazil

[2]Federal University of Parana (UFPR) – Curitiba – PR – Brazil

[3]Sorbonne Université – CNRS/LIP6 – Paris – France

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

Abstract. *The most common strategy for enabling a process in a distributed system to broadcast a message is one-to-all communication. However, this approach is not scalable, as it places a heavy load on the sender. This work presents an autonomic algorithm that enables the n processes in a distributed system to build and maintain a spanning tree connecting themselves. In this context, processes are the vertices of the spanning tree. By definition, a spanning tree connects all processes without forming cycles. The proposed algorithm ensures that every vertex in the spanning tree has both an in-degree and the tree depth of at most $log_2 n$. When all processes are correct, the degree of each process is exactly $log_2 n$. A spanning tree is dynamically created from any source process and is transparently reconstructed as processes fail or recover. Up to $n-1$ processes can fail, and the correct processes remain connected through a scalable, functioning spanning tree. To build and maintain the tree, processes use the VCube virtual topology, which also serves as a failure detector. Two broadcast algorithms based on the autonomic spanning tree algorithm are presented: one for best-effort broadcast and one for reliable broadcast. Simulation results are provided, including comparisons with other alternatives.*

Note: this preprint is an English translation and slightly extended version of the paper published in Portuguese at the 32nd Brazilian Symposium on Computer Networks and Distributed Systems (2014), reference [1].

1. Introduction

Spanning trees are used to solve various problems in distributed systems, such as mutual exclusion, clustering, message broadcasting, among many others [2, 3, 4, 5]. The main reason for being so important in this context is that a spanning tree represents a low-cost way to connect all processes of the system.

Consider, for instance, message broadcasting. If a message is sent from the source to all destinations, as the number of processes grows, there is a high load on the source. The alternative is to allow processes relay messages to other processes, which relieves the source. However, depending on the topology the processes form, multiple copies of the message may be received by each process. A spanning tree avoids that, as it connects all processes without forming any cycles. For a message to reach all nodes it only needs to be transmitted along the $n-1$ edges of the tree [6].

Another aspect that must be taken into account is the possibility of process failures, which is intrinsic to distributed systems. A fault-tolerant distributed application

must continue its correct execution even after process faults have occurred. Thus, the spanning tree must be resilient to process failures. Ideally, the failover procedure should be transparent. In this way, the system autonomically adapts itself to any fault situation [7]. Therefore, in addition to the algorithm employed to construct the tree, it is also paramount to have an algorithm for rebuilding or reconfiguring the tree as processes become faulty or recover.

In this work, we assume a fully-connected distributed system in which n processes communicate through message passing. Processes may fail by crashing, and can also recover and rejoin the system. The processes form a virtual topology called the VCube [8, 9]. The topology is virtual in the sense that processes only communicate across the VCube edges. Nevertheless, given the underlying fully-connected topology any process can directly communicate with every other process, without employing intermediates.

VCube is a failure detector [10, 11, 12] that organizes processes into progressively larger hierarchical clusters. A VCube is a hypercube when the number of processes is a power of two and all processes are correct. The virtual topology presents several logarithmic properties that ensure its scalability. Those properties are guaranteed even as processes fail and recover

We propose an autonomic spanning-tree algorithm constructed on a VCube in a distributed manner. Starting from any process that is called the source and which becomes the tree root, each process communicates with the first correct process in its clusters. Those processes in turn become roots of the subtrees in their clusters, and are connected to processes on their own clusters, and so on. When all processes are correct, the degree of a process is at most $log_2 n$, as well as the height of the spanning tree. A tree is reconstructed autonomously and transparently after failures and recoveries occur.

Two distributed broadcast applications were implemented using the proposed autonomic spanning trees: one for best-effort broadcast and the other for reliable broadcast. Simulation results confirm that the use of the virtual topology and the tree algorithm increases scalability in comparison with traditional one-to-all traditional alternatives. The message propagation latency is only slightly highly than that of a traditional all-communicates-with all strategy.

The rest of this work is organized as follows. The next section presents the system model and introduces the VCube topology. Section 3 presents the proposed spanning tree algorithm. Section 4 presents the tree-based broadcast algorithms. Simulation results are presented in Section 5. Section 6 describes related work, in particular presenting a survey of the main distributed algorithms that have been proposed for the VCube. The conclusion follows in Section 7.

2. System Model & The VCube Virtual Topology

A distributed system consists of a finite set Π of $n > 1$ independent processes $\{p_0, ..., p_{n-1}\}$ that collaborate to perform some task. Each process is considered to be executed on a distinct node. Therefore, the terms *node* and *process* are used interchangeably. The system under consideration is synchronous. Therefore, the temporal attributes related to process execution speed and message delay in communication channels have known bounds.

Let $G = (V, E)$ be the complete and undirected graph representing Π, in which there are $n = |V|$ vertices representing the processes and edge (i, j) indicates that process i can communicate directly with process j and vice versa. The graph is complete, and represents a fully connected system, in which any process can directly communicate with any other process. Thus $\exists(i, j), \forall i, j \in V, i \neq j$. A (*spanning tree*) of G is a connected and acyclic subgraph $T = (V, E')$ in which $E' \subseteq E$, that is, T, contains all the vertices of G and $|E'| = |V| - 1$. If edges have associated weights, a *minimum spanning tree* is one whose sum of the edge weights is minimal. If each edge has a different weight, there is a single minimal tree. If all edges have the same weight, all trees in the graph are minimal [13]. In this work, all links have the same weight which is equal to 1 and omitted in the rest of the paper.

Processes communicate by sending and receiving messages. The processes are organized in a virtual hypercube. A d-dimensional hypercube has $n = 2^d$ processes. The identifier i of a process satisfies $0 \leq i \leq n - 1$ and consists of a d-bit binary address $i_{d-1}, i_{d-2}, ..., i_0$. Two processes are connected if their addresses differ by exactly one bit. The transmission and receipt of messages are atomic operations, but *multicast* and *broadcast* are not supported as native operations. The links are reliable and never fail. Therefore, no messages are lost, corrupted, or duplicated during transmission. There is no network partitioning.

Processes can fail by crashing, and such failures are permanent. A process is considered to be "correct" or "fault-free" if it does not fail during the entire system execution. Otherwise, the process is considered to be "faulty". After it has crashes, a process does not perform any actions and does not respond to external stimuli; that is, it stops executing completely, and does not send or receive messages.

Faulty processes are detected by a perfect failure detector; that is, no faulty process q is suspected unless it is actually faulty, and if q is faulty, every correct process p will be notified by the detector module about the failure of p within a finite time [14].

Using a virtual topology to connect processes in a distributed system facilitates application development by abstracting the physical structure and facilitating system reconfiguration when necessary. The topology employed in this work, called VCube, organizes the processes into a virtual hypercube when there are no failures and the number of processes is a power of 2. However, if a process fails, the VCube reconfigures itself to adapt and maintain the topology connected. Even in the presence of failures, VCube maintains the following logarithmic properties: in a d-dimensional VCube with $n = 2^d$ vertices, the total number of edges is at most $n \log_2 n$ despite of which processes are faulty or correct, and the maximum distance between two vertices is $\log_2 n$.

To build the VCube processes are organized into progressively larger clusters [8], as illustrated in Figure 1. Each process i has $s = 1, .., \log_2 n$ clusters each with 2^{s-1} processes. Clusters in which $s = 1$ have one process. Clusters of in which $s = 2$ have 2 processes, and so on.

The processes belonging to each cluster can be computed with function $C_{i,s} = \{i \oplus 2^{s-1}, c_{i\oplus 2^{s-1},1}, ..., c_{i\oplus 2^{s-1},s-1}\}$, in which $\oplus$ represents the binary operation of *exclusive or* (*xor*).

Let i and j be two processes of the system. Next, we define three functions on the

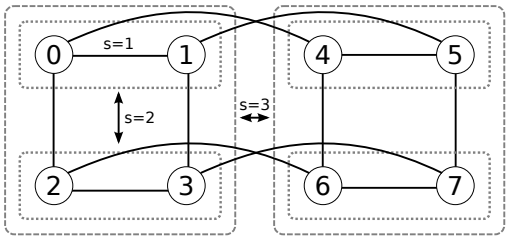


Figure 1. The clusters of a three-dimensional VCube.

VCube: $cluster_i(j)$, $FF\_neighbor_i(s) = j$, and $neighborhood_i(h)$, where h corresponds to the height of spanning tree as will be detailed below.

Function $cluster_i(j) = s$ determines which cluster s of process i process j belongs to. For example, in Figure 1 $cluster_0(1) = 1$, $cluster_0(2) = cluster_0(3) = 2$, $cluster_0(5) = 3$. Note that for every pair i, j, $cluster_i(j) = cluster_j(i)$.

Another function defined on the VCube is $FF\_neighbor_i(s) = j$. This function computes the first correct process j in cluster $C_{i,s}$. If all processes in the cluster are faulty, the function returns $\perp$. In the VCube in Figure 1, for example, in a fault-free scenario $FF\_neighbor_4(1) = 5$, $FF\_neighbor_4(2) = 6$, $FF\_neighbor_4(3) = 0$. On the other hand, if p_4 has detected p_6 faulty, $FF\_neighbor_4(2) = 7$. If p_6 and p_7 are faulty, $FF\_neighbor_4(2) = \perp$. It is important to emphasize that this function is dependent on which processes have been detected as faulty. Due to the latency required for detection, within the same timeframe, it is possible for two processes to have different views on which processes are correct or faulty.

Finally, function $neighborhood_i(h) = \{j \mid j = FF\_neighbor_i(s), j \neq \perp, 1 \leq s \leq h\}$ is defined. This function generates a set containing all correct processes virtually adjacent to process i according to $FF\_neighbor_i(s)$, for $s = 1, .., h$. Parameter h varies from 1 to $\log_2 n$. If $h = \log_2 n$ the resulting set contains all correct neighbors of process i. For any other value of $h < \log_2 n$, the function returns only a subset of the neighbors contained in the clusters $s = 1, .., h$. Like $FF\_neighbor_i$, this function depends on the local knowledge that process i has about the state of the other processes. As an example, for the VCube in Figure 1 and in a fault-free scenario, $neighborhood_0(1) = \{1\}$, $neighborhood_0(2) = \{1, 2\}$, and $neighborhood_0(3) = \{1, 2, 4\}$. If process p_4 is detected as faulty by process p_0, then $neighborhood_0(3) = \{1, 2, 5\}$. If p_1 is faulty, $neighborhood_0(1) = \emptyset$.

Thus, the system topology in VCube is formed by the connection of each process $i \in V$ with all its neighbors as determined by function $neighborhood_i(\log_2 n)$. Similarly, the neighbors of a process i restricted to the cluster s to which i belongs in relation to another process j are determined by $neighborhood_i(cluster_i(j) - 1)$.

3. The Spanning Tree Algorithm

This section presents the proposed autonomic and distributed spanning tree algorithm based on the VCube topology. The spanning tree is considered to be *autonomic* because it dynamically rebuilds itself as processes crash and recover. Moreover, in such cases the tree is regenerated locally, by the affected neighbor processes.

Consider that the distributed system is represented as a graph, of which the n processes are vertices. The spanning tree connects all those processes using the minimum number of edges: $n-1$. Algorithm 1 shows how a message called TREE is propagated from the source process (which represents the root of the spanning tree) to all other correct processes.

Algorithm 1 Propagation of Message TREE along the Spanning Tree

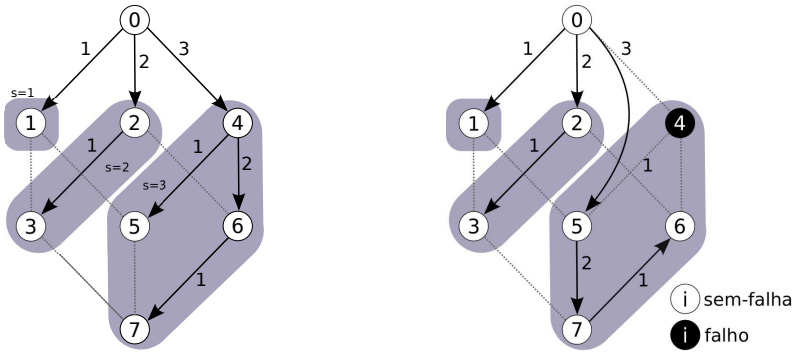
```
1:  correct_i ← {0, .., n − 1} // list of correct processes
2:  procedure STARTTREE( )
3:      // executed by the source, which is the tree root
4:      // the source sends message TREE to all neighbors
5:      for all k ∈ neighborhood_i(log_2 n) do
6:          SEND(⟨TREE⟩) to p_k
7:  upon receive ⟨TREE⟩ from from j
8:      // a process that is not the source receives message TREE
9:      if j ∈ correct_i then
10:         // retransmits to the neighbors in internal clusters
11:         for k ∈ neighborhood_i(cluster_i(s) − 1) do
12:             SEND(⟨TREE⟩) to p_k
13: upon notification crash(process j)
14:     // process i is notified that process j has been detected as crashed
15:     correct_i ← correct_i ∖ {j}
16:     if k = FF_neighbor_i(cluster_i(j)), k ≠ ⊥ then
17:         SEND(⟨TREE⟩) to p_k
```

Initially, consider a fault-free execution in which all processes are correct. The STARTTREE procedure is executed by the source process, which is the tree root. Note that any process can be the source. A TREE message is sent to the root's $\log_2 n$ correct neighbors, one in each cluster $s = 1, .., \log_2 n$ (lines 5-6).

Upon receiving message TREE from process j, process i forwards the message to its internal clusters, with respect to j and $cluster_i(j)$. Thus i forwards message TREE to the first correct process in clusters $s' = 1, .., cluster_i(j) - 1$ (line 11).

Figure 2(a) shows as an example a 3-dimensional VCube in which all processes are correct. Process p_0 is the source and sends message TREE to its neighbors, the first correct process in each of its $logn$ clusters, computed as follows: $neighborhood_0(3) = \{FF\_neighbor_0(1), FF\_neighbor_0(2), FF\_neighbor_0(3)\} = \{1, 2, 4\}$. Process p_1 receives the message but does not retransmit it, since $cluster_1(0) = 1$ and $neighborhood_1(0) = \perp$. Process p_2 receives the message and forwards it to its neighbor p_3 in cluster $s = 1$, since $neighborhood_2(1) = \{FF\_neighbor_2(1)\} = \{3\}$. Similar to p_1, When p_3 receives the message, it computes $cluster_3(2) = 1$ and does not retransmit it. In the case of p_4, the message is received and retransmitted to neighbors $5 \in c_{4,1}$ and $6 \in c_{4,2}$. Finally, since $FF\_neighbor_6(1) = 7$, process p_6 sends the message to process p_7.

Failure cases can be divided into two scenarios. First, consider that a process $j \in c_{i,s}$ has crashed and process i has already been informed of this fault by the de-



(a) fault-free (b) process p_4 is crashed

Figure 2. An example spanning tree built on a 3-dimensional VCube.

tector, that is, $j \notin correct_i$. In this case, process i sends the message to its correct neighbor $k = FF\_neighbor_i(s)$ and the message is propagated correctly by k to all its internal clusters (of k). In a second scenario, consider the same crashed process j, but process i has not yet been informed by the detector, that is, $j \in correct_i$. In this case, if $FF\_neighbor_i(s) = j$, process i sends the $TREE$ message to j, and thus the message is not received. Propagation stops prematurely and the internal subtree of cluster s does not get the message. However, as soon as the detector module informs i about the failure, a new $FF\_neighbor_i(s) = k$ is selected and the message is retransmitted to k, thus rebuilding the subtree and allowing the message to reach all processes (line 16).

Figure 2(b) illustrates a failure scenario. Let process p_0 be the root and p_4 be the crashed process. Since p_0 is aware of the failure of process p_4, instead of sending message TREE to p_4, p_0 sends the message to p_5, which is the first correct process in $c_{0,3}$. Process p_5, in turn, forwards the message to $p_7 \in c_{5,2}$. Finally, p_7 retransmits the message to $p_6 \in c_{7,1}$, completing the tree. If when p_0 starts the broadcast and $p_4 \in correct_i$, p_0 sends the message to p_4 and, when the detector informs it about the failure of p_4, the message will be retransmitted to p_5. From this point on, propagation is analogous to the previous case.

Lemma 1 proves that all correct processes receive the a message that is propagated across the tree built with Algorithm 1. In the next section, two broadcast algorithms based on the spanning tree are proposed, and this lemma is used in the proof of their correctness.

Lemma 1. *Let m be a message propagated by a correct source process src through the spanning tree. Every correct process receives m.*

Proof. The proof of this lemma is by induction. Consider as the basis of induction a system with $n = 2$ processes: p_0 is the src process that initiates the transmission of message m across the tree. According to the $c(i, s)$ function, $p_1 \in c_{0,1}$. If p_1 is correct, $FF\_neighbor_0(1) = 1$ and p_0 sends m to p_1 (line 5). Therefore, p_1 receives m and the lemma holds.

As the induction hypothesis, consider that the lemma holds for a system with $n = 2^k$ processes.

The induction step demonstrates that the lemma holds for a system with $n = 2^{k+1}$ processes. According to VCube's hierarchical topology, a system with $n = 2^{k+1}$

processes consists of two subsystems with $n = 2^k$ processes, as illustrated in Figure 3. The figure shows that src and j are the roots of those subsystems. Process src executes the algorithm and sends m to each process returned by $FF\_neighbor_{src}(s)$, $s = 1, .., k$ (line 5). Since $j = FF\_neighbor_{src}(k)$ is a correct process, j correctly receives m. If j is detected as crashed, a copy of the message is retransmitted to the next correct process in the same cluster as j (line 16). Thus, message m is transmitted in both subsystems (of $n = 2^k$ processes) and, by hypothesis, every correct process receives m in each subsystem. Since every process in Π belongs to one of those systems, every correct process in Π receives m.

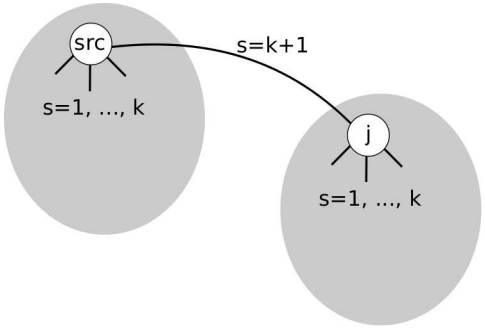


Figure 3. Dissemination of TREE messages in a system with $n = 2^{k+1}$ processes which consists of two subsystems with $n = 2^k$ processes.

4. Broadcast Algorithms Based on the Spanning Tree

Message broadcasting [15] is a fundamental building block used to implement several distributed algorithms. Examples include content delivery networks, publish/subscribe systems, distributed replication, and group communication. A source process broadcasts a message to all other processes in the system. However, if the source fails before completing the message transmission, some destination processes may not receive the message. If the required type of broadcast is *best-effort*, nothing further is needed; this ensures that only if the source is correct, all correct processes deliver the message. In contrast, if it is necessary to guarantee that all correct processes deliver the message even if the source fails, a different type of broadcast is required: *reliable broadcast*.

Fault-tolerant broadcast algorithms are typically implemented assuming underlying reliable point-to-point communication channels between processes. Processes communicate by sending and receiving messages executing the SEND and RECEIVE primitives. Processes invoke BROADCAST(m) and DELIVER(m) to broadcast and deliver messages to/from other application processes. A broadcast algorithm may rely on a failure detector to be notified if some process fails.

Next, two broadcast algorithms based on the VCube are presented. The first is a best-effort broadcast algorithm, while the second implements reliable broadcast. Both algorithms disseminate messages through the spanning tree described in Section 3.

4.1. Best-Effort Broadcast

Best-effort broadcast guarantees that all correct processes deliver the same set of messages as long as the sender (source) is correct. A best-effort broadcast algorithm must guarantee

three properties: validity, no duplication, and no creation of messages [16]. According to the validity property, if a process p_i sends a message m to a process p_j and neither of them fails, then p_j eventually delivers m. No duplication guarantees that no message is delivered more than once, and no creation guarantees that no message is delivered unless it has been previously broadcast by some correct process.

Algorithm 2 is a best-effort broadcast solution on the VCube spanning tree. Two message types are used: $\langle TREE, m\rangle$ for application messages and $\langle ACK, m\rangle$ which are acknowledgment messages (ACKs) used to confirm the receipt of application messages. The VCube failure detector notifies correct processes whenever some process is detected to have crashed. The algorithm works correctly even if up to $n-1$ processes crash. A preliminary version of this algorithm was published by Rodrigues [17] applied to distributed mutual exclusion.

Processes executing the best-effort broadcast algorithm maintain the following local variables:

- $correct_i$: set of processes that process i considers to be correct;

- $last_i[n]$: the last message received from each source process. Every message has a unique identifier, consisting of the source process id plus a timestamp. The timestamp is implemented by the source as local counter of messages. Function $ts(m)$ returns the timestamp of message m. Note that any process will only send a new message after it was successfully completed the broadcast of the previous message;

- ack_set_i: the set of ACKs that process i is waiting for (called pending ACKs). For each message $\langle TREE, m\rangle$ received by process i from process j and retransmitted to process k, an element $\langle j, k, m\rangle$ is added to this set.

The symbol $\perp$ represents a null element. The asterisk is used as a wildcard to select ACKs from the set ack_set. Thus, for example: an element such as $\langle j, *, m\rangle$ represents all pending ACKs for message m received from process j and retransmitted to any other process.

Algorithm 2 The Hierarchical Best-Effort Broadcast Algorithm executed by process i

1: $last_i[n] \leftarrow \{\perp, .., \perp\}$
2: $ack\_set_i = \emptyset$
3: $correct_i = \{0, .., n-1\}$

4: **procedure** BROADCAST(message m)
5: **wait until** $ack\_set_i \cap \{\langle \perp, *, last_i[i] \rangle\} = \emptyset$
6: $last_i[i] = m$
7: DELIVER(m)
8: **for all** $j \in neighborhood_i(\log_2 n)$ **do** // sends the message to all neighbors
9: $ack\_set_i \leftarrow ack\_set_i \cup \{\langle \perp, j, m \rangle\}$
10: SEND($\langle TREE, m \rangle$) **to** p_j

11: **procedure** CHECKACKS(process j, message m)
12: **if** $ack\_set_i \cap \{\langle j, *, m \rangle\} = \emptyset$ **then**
13: **if** $\{source(m), j\} \subseteq correct_i$ **then**
14: SEND($\langle ACK, m \rangle$) **to** p_j

15: **upon** `receive` $\langle TREE, m \rangle$ **from from** p_j
16: **if** $\{source(m), j\} \nsubseteq correct_i$ **then**
17: **return**
18: // checks if m is a new message
19: **if** $last_i[source(m)] = \perp$ **or** $ts(m) = ts(last_i[source(m)]) + 1$ **then**
20: $last_i[source(m)] \leftarrow m$
21: DELIVER(m)
22: // retransmits m to neighbors in internal clusters
23: **for all** $k \in neighborhood_i(cluster_i(j) - 1)$ **do**
24: **if** $\langle j, k, m \rangle \notin ack\_set_i$ **then**
25: $ack\_set_i \leftarrow ack\_set_i \cup \{\langle j, k, m \rangle\}$
26: SEND($\langle TREE, m \rangle$) **to** p_k
27: CHECKACKS(j, m)

28: **upon** `receive` $\langle ACK, m \rangle$ **from from** p_j
29: $k \leftarrow x : \langle x, j, m \rangle \in ack\_set_i$
30: $ack\_set_i \leftarrow ack\_set_i \setminus \{\langle k, j, m \rangle\}$
31: **if** $k \neq \perp$ **then**
32: CHECKACKS(k, m)

33: **upon notification** `crash`(process j) // j is detected as crashed
34: $correct_i \leftarrow correct_i \setminus \{j\}$
35: **for all** $p = x, q = y, m = z : \langle x, y, z \rangle \in ack\_set_i$ **do**
36: **if** $\{source(m), p\} \nsubseteq correct_i$ **then**
37: // remove pending ACKs for $\langle j, *, * \rangle$ and $\langle *, *, m \rangle : source(m) = j$
38: $ack\_set_i \leftarrow ack\_set_i \setminus \{\langle p, q, m \rangle\}$
39: **else if** $q = j$ **then** // retransmits to new neighbor k, if exists
40: $k \leftarrow FF\_neighbor_i(cluster_i(j))$
41: **if** $k \neq \perp$ **and** $\langle p, k, m \rangle \notin ack\_set_i$ **then**
42: $ack\_set_i \leftarrow ack\_set_i \cup \{\langle p, k, m \rangle\}$
43: SEND($\langle TREE, m \rangle$) **to** p_k
44: $ack\_set_i \leftarrow ack\_set_i \setminus \{\langle p, j, m \rangle\}$
45: CHECKACKS(p, m)

4.1.1. Algorithm Description

A process i running Algorithm 2 invokes the BROADCAST primitive to broadcast a message m. A new message is broadcast only after the previous one has completed (line 5), that is, when there are no more pending ACKs for the message $last_i[i]$. In lines 8-10 the new message is sent to all neighbors j considered to be correct. For each message sent to j, a tuple $\langle \perp, j, m \rangle$ is added to the list of pending ACKs.

When process i receives a TREE message from process j (line 15), it first checks whether both the message source and j are correct. If one of those processes has crashed, the reception is aborted. If j has crashed, the process that had sent m to j will make resend that message after it detects that j has crashed. Then process i will receive the message through after the tree is rebuilt. On the other hand, if the source crashes, it is no longer necessary to continue the broadcast. If both the source and j are correct, process i checks whether the message is new by making a comparison of its *timestamp* with that of the last message stored in $last_i[j]$ and the received message m. If the message is a new one, then $last_i[j]$ is updated, and the the message is delivered to the application. Next, m is forwarded to the neighbors in each cluster of i. If there is no correct neighbor or if i is a leaf in the tree ($cluster_i(j) = 1$), there are no pending ACKs (CHECKACKS) then i immediately sends an ACK to j.

If a message $\langle ACK, m \rangle$ is received (line 28), set ack_set_i is updated, and if there are no more pending ACKs for message m, process i sends a $\langle ACK, m \rangle$ to the process k from which i previously received the TREE message (CHECKACKS). However, if $k = \perp$, the ACK has reached the source process and no longer needs to be propagated.

The detection of a crashed process j is handled by the CRASH event. Three actions are performed: (1) updating the list of correct processes; (2) removing pending ACKs for messages received from j or messages that contain process j as the destination; (3) messages that had been previously transmitted to j must now be retransmistted to another correct process k in the same cluster as j, if there is one. This retransmission triggers the propagation the message along a new subtree.

In Theorem 2, the correctness of the proposed best-effort broadcast algorithm is proven.

Theorem 2. *Algorithm 2 ensures that if the source process remains correct, every correct process will eventually deliver the broadcast message.*

Proof. The no duplication and no creation properties are derived from the underlying communication channels, which are perfect (reliable). Furthermore, every message has a unique identifier (timestamp), and even if some process receives that message multiple times, it is only delivered once (line 19).

Reliable delivery is guaranteed as follows. After the source broadcasts message m, it waits for acknowledgment messages (ACKs) from all correct neighbors. Lemma 1 proved that if the sender is correct, every correct process receives m, even if an intermediate process j crashes during retransmission. In this case, the message is retransmitted to the next correct neighbor k in the same cluster as process j (line 40).

Theorem 3. *The total number of messages generated in a fault-free execution of Algorithm 2 is* $2 * (n - 1)$*, including ACKs. If a process* i *that received a message from a*

process j crashes before sending back the ACK, the total number of extra messages depends on the number of processes in the cluster and their states. Recall that a cluster consists of 2^{s-1} processes and the largest cluster has $n/2$ processes.

Proof. In a fault-free execution of Algorithm 2, for each TREE message sent, an ACK message is returned. If $n-1$ is the number of edges in the tree that has n processes as vertices, then the total number of messages is twice the total number of edges.

If a process j is detected to have crashed by a process i after it has sent a TREE message was j, a new message will be sent to the next neighbor $k = FF\_neighbor_i(cluster_i(j))$. Let $s = cluster_i(j)$. In the best case, $k = \perp$ and no extra messages are sent (this is the case if $j \in c_{i,1}$ or there are no more correct processes in cluster s). However, if $k \neq \perp$, the number of extra messages depends on the number of processes detected as crashed/correct in cluster s. Let $n' = |c_{i,s}|$ denote the total number of processes in cluster $c_{i,s}$. In the worst case, all processes in the cluster are correct except j, and j has sent the message to all neighbors before it crashed. Thus, in this case the total number of extra messages is $1 + 2 * (n' - 2)$. An extra TREE message is sent for k and $(n' - 2)$ TREE messages + $(n' - 2)$ ACKs are sent in the subtree. In general, if there are f faulty processes in $c_{i,s}$ including j, the number of extra messages retransmitted is $1 + 2 * (n' - 1 - f)$.

4.2. Reliable Broadcast

A reliable broadcast algorithm ensures that the same set of messages is delivered to all correct processes, even if the source crashes as broadcast is being executed. The reliable broadcast algorithm proposed in this work uses the VCube spanning trees described in Section 3 to propagate messages hierarchically, and it works even if up to $n-1$ processes crash. Algorithm 3 is similar to the best-effort broadcast Algorithm 2, but must handle the crash of the source process differently. Algorithm 3 only shows the difference. The rest of the pseudo-code is identical and has been omitted.

A process i that is the broadcast source invokes primitive BROADCAST(m). If i never crashes (in this case i is the message source, i.e. $source(m) = i$) the algorithm is identical to the best-effort algorithm. After delivering the message locally, i sends the message to the first correct neighbor in each of its clusters. If i is not the source of the message, it only retransmits the message to its neighbors.

When a process i receives the message $\langle TREE, m \rangle$ from a process j, it first checks whether $j \notin correct_i$. If j has crashed, the message is discarded. Note that this check differs from that of the best-effort algorithm as the reliable broadcast algorithm does not discard messages received from a crashed $source(m)$. The second difference is in lines 16-18. If i receives a new message from a source process which has crashed, it initiates a new broadcast with the received message to ensure that the other processes receive m correctly. In this case, in line 2, $source(m) \neq i$ and the message is rebroadcast to the other processes through the spanning tree of j.

Crash notifications are handled similarly to best-effort broadcast, except that is necessary to rebroadcast the last message received from the process notified to have crashed (line 34). The retransmissions in line 17 ensure that all correct processes will receive and deliver the last message broadcast by the crashed source process j. That is guaranteed even if a single correct process received the message before j crashed.

Algorithm 3 Hierarchical Reliable Broadcast executed by process 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:     // sends the message to all neighbors
7:     ...
8: upon receive ⟨TREE, m⟩ from from p_j
9:     if j ∉ correct_i then
10:        return
11:    // checks if m is a new message
12:    if last_i[source(m)] = ⊥ or
13:     ts(m) = ts(last_i[source(m)]) + 1 then
14:        last_i[source(m)] ← m
15:        DELIVER(m)
16:        if source(m) ∉ correct_i then
17:            BROADCAST(m)
18:            return
19:    ...
20: upon notification crash(process j) // j is detected as crashed
21:    correct_i ← correct_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 pending ACKs for ⟨j, ∗, ∗⟩
25:            ack_set_i ← ack_set_i ∖ {⟨p, q, m⟩}
26:        else if q = j then // retransmits m to new neighbor k, if exists
27:            k ← FF_neighbor_i(cluster_i(j))
28:            if k ≠ ⊥ and ⟨p, k, m⟩ ∉ ack_set_i then
29:                ack_set_i ← ack_set_i ∪ {⟨p, k, m⟩}
30:                SEND(⟨TREE, m⟩) to p_k
31:            ack_set_i ← ack_set_i ∖ {⟨p, j, m⟩}
32:            CHECKACKS(p, m)
33:    if last_i[j] ≠ ⊥ then
34:        BROADCAST(last_i[j])
```

5. Simulation

The proposed broadcast algorithms were implemented using Neko [18], a Java framework for simulating distributed algorithms. The proposed hierarchical best-effort broadcast algorithm is called ATREE-B and the reliable broadcast algorithm ATREE-R. The algorithm was compared with two other approaches: the first is based on the classic all-to-all approach, and is called ALL-B (best-effort broadcast) and ALL-R (reliable broadcast). The second approach that was compared is also tree-based, but non-autonomic: NATREE-B (best-effort broadcast) and NATREE-R (reliable broadcast). We refer to the three different approaches as ATREE, ALL, and NATREE. All algorithms employ the failure detection service provided by VCube to learn about process crashes.

The ALL strategy also employs a $neighborhood_i$ function, but in this case it includes *all* correct neighbors of process i. Whenever process i disseminates a message, it sends the message to all those neighbors. The NATREE strategy builds a tree using flooding. The tree is created using the TREE broadcast message itself. Each process that receives the message for the first time joins the tree and retransmits the message to the other processes in the system, except the one from which it received the message. If a process that receives the message is already in the tree, it just sends back a NACK message and does not forward the message. Once the tree is created, further messages are sent only along the edges of the tree. In case of failure, a new tree is created from the source process using the same flooding procedure.

5.1. Simulation Parameters

When a process needs to send a message to more than one recipient, it must execute the SEND primitive sequentially. Thus, for each message, t_s time units are used to send the message and t_r time units to receive it, in addition to the transmission delay t_t. These intervals are computed for each copy of the message sent.

To evaluate the performance of the broadcast algorithms, two metrics are used: (1) *throughput*, given by the total number of broadcasts completed during a time interval; (2) the *latency* to deliver the broadcast message to all correct processes.

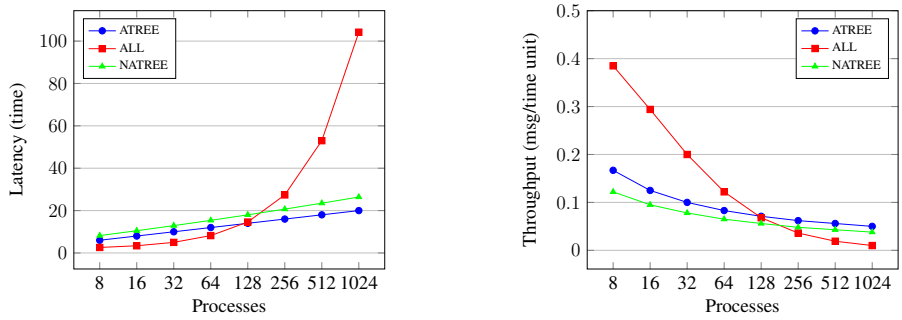
The proposed algorithms were evaluated in different scenarios, varying both the total number of processes and the number of crashed processes. The communication parameters were set to $t_s = t_r = 0.1$ and $t_t = 0.8$. The failure detector employs a testing interval set to 5.0 time units. A process is considered to have crashed if it does not respond to the test after $4 * (t_s + t_r + t_t)$ time units (timeout interval).

5.2. Simulation Results

The experiments were conducted in two stages. Initially, fault-free scenarios were used with systems with 8 to 1,024 processes. Furthermore, the latency and the number of messages sent by each application were computed individually in a system with 512 processes. Subsequently, failure scenarios were randomly generated for a system with 512 processes containing from 1% to 8% of crashed processes.

Experiments in fault-free scenarios. In fault-free scenarios, since there are no retransmissions, the proposed VCube-based best-effort and reliable broadcast algorithms present very similar performance. Figure 4 shows the latency and throughput considering that a single message is sent by process p_0. The longest path in a VCube with n processes is $\log_2 n$. Therefore, when n is small, the time to send the message via the longest path is greater than the time to send the $n - 1$ messages sequentially using the ALL strategy. Considering that the transmission time for each message is $t_s = 0.1$, the interval between sending a TREE message and receiving the corresponding ACK via the longest path in the tree is $2 \log_2 n(t_s + t_r + t_t)$. In the ALL strategy, to send $n-1$ messages, $(n-2)t_s + t_t + tr$ time units are required.

Thus, although ALL is more efficient for small systems, its performance declines as the system size grows. The NATREE non-autonomic tree solution presents similar behavior to ATREE, except for the extra latency that caused by processes having to wait



(a) Latency (b) Throughput

Figure 4. Best-effort broadcast: all processes are correct.

for ACK/NACK messages during tree setup. In fault-free scenarios and after the tree is complete, the performance of both solutions is the same.

Throughput was computed as $1/latency$, since only one process sends a single message. Similar to the latency, the ALL solution presents the best performance for small systems, up to 128 processes. However, as n increases, its performance decreases quickly. These results confirm the scalability of the proposed hierarchical strategy.

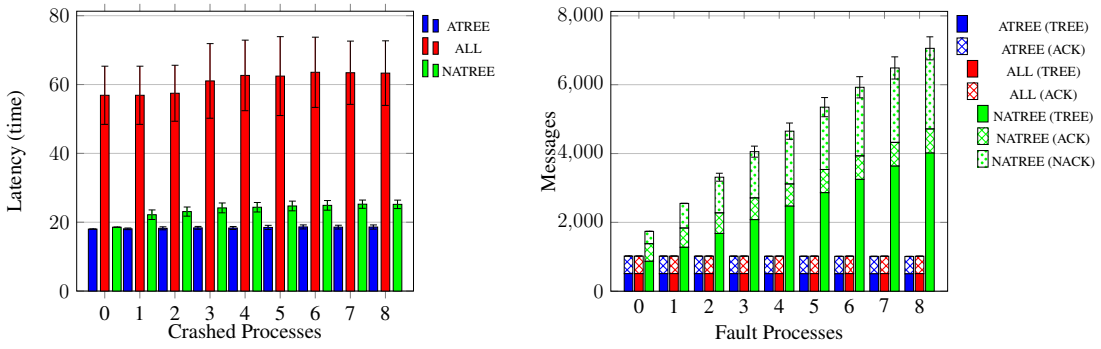
Experiments in scenarios with failures. Scenarios with failures were executed in a system with 512 processes. The the number of crashed processes varied from 1 to $\log_2 n$. For each number of crashed processes, 100 different scenarios were generated randomly following a normal distribution. Unlike the scenarios without failures, in which a single message is sent, the experiments with failures consisted of broadcasting 10 messages. The average latency is shown in Figure 5(a). The ALL solution presents higher latency than ATREE because it needs to send all messages sequentially, as explained above. In the NATREE solution, the result is similar to ALL since, for each failure, the tree needs to be rebuilt using flooding. ALL is more efficient both in the scenarios without and with few failures, but its performance degrades quickly as the number of failures increases. Regarding the number of messages, the results for ATREE and ALL are very similar, as shown in Figure 5(b). In the case of NATREE, the total number of messages sent is higher, as it is necessary to rebuild the tree from the root using flooding. These results demonstrate the efficiency of the proposed solution in creating and maintaining the tree autonomically.

6. Related Work

This section reviews work related both in terms of the construction of spanning trees in distributed systems and the VCube virtual topology. It is divided into two main subsections: the first focuses on spanning tree algorithms, while the second surveys VCube and its applications in distributed computing.

6.1. Spanning Tree Algorithms

Two of the most classic algorithms for building a minimum spanning tree from a graph are the algorithm by Kruskal and Joseph [19] and Prim's algorithm [20]. Kruskal's algorithm initially creates a forest in which each vertex is a tree. At each step, the trees are connected



(a) Latency (b) Total messages

Figure 5. Broadcast for $n = 512$ varying the numbers of crashed processes.

through the lowest-weight edges. Edges that do not connect two trees are discarded, thus avoiding cycles. In the end, a single connected component is generated, which is the minimum spanning tree. Prim's algorithm uses a different approach, employing minimal cuts to select the lowest-weight edges for inclusion in the tree.

Many distributed algorithms for building spanning trees are based on the centralized Kruskal-Joseph and Prim algorithms. The first of these was defined by Gallager et al. [13]. The process is similar to that used by Kruskal. Initially, each process is a tree. At each level, a node is elected as leader, and a minimum-weight edge connecting it to a process in another tree is added. The process is repeated until a single connected component is formed. The algorithm proposed by Dalal [21] uses Prim's algorithm to connect tree segments by choosing the lowest-weight edge connecting two segments.

Avresky [22] presents three algorithms for constructing and maintaining trees in fault-prone hypercube-based systems. The algorithms tolerate single faults but may block in certain combinations with multiple faults.

The work of Flocchini et al. [23] proposes a fault-tolerant solution for spanning trees that reconstructs the tree after failures using pre-computed alternative trees considering all failure possibilities.

6.2. VCube Applications

VCube [9] was proposed as a scalable virtual topology for large-scale distributed systems, enabling efficient and fault-tolerant communication over a hierarchical structure. VCube ensures that both latency and the number of tests are provably bounded by logarithmic functions. VCube evolved from the earlier Hierarchical Adaptive Distributed System-level Diagnosis algorithm (Hi-ADSD) [8], that was applied for network fault management based on the SNMP (Simple Network Management Protocol) [24]. Hi-ADSD presented a latency of $log_2^2 n$ testing rounds. Another version called isochronous [25] reduced the latency to the same of VCube $log_2 n$ rounds. VCube guarantees that latency by having each correct process test all $log_2 n$ clusters in all rounds. Hi-ADSD already exhibited several logarithmic properties, but in certain rare fault situations, it resulted in a quadratic number of failures. Later Hi-ADSD with Detours [26] was proposed to guarantee a logarithmic number of tests in all situations, but the proof of the worst case remains open. DiVHA (Distributed Virtual Hypercube Algorithm) was the first hierarchical diagnosis algorithm

to have a provably scalable number of tests [27], and was implemented as an overlay network [28], ensuring scalability and robustness under churn.

The original Hi-ADSD algorithm assumed static events, i.e. a process could only fail or recover after the previous event had been completely diagnosed. The algorithm presented in [29] allows dynamic events, and furthermore allows a correct process to obtain diagnostic information from multiple other processes, employing timestamps to allow more recent events to be identified.

Several broadcast algorithms have been designed based on the VCube. An autonomic reliable broadcast algorithm using dynamic spanning trees built within a VCube were presented in [30]. That algorithm was later extended to be able to work in an asynchronous systems in which it is impossible to determine whether a process has crashed or is simply slow [31]. Communication-efficient causal broadcast protocols based on multiple intersecting trees built on a VCube were introduced in [32] and later applied to implement a publish/subscribe system in VCube-PS [33, 34]. Message bundling techniques to reduce the communication overhead in VCube-based trees were explored in [35].

VCube has also been used as a substrate for autonomic spanning tree construction and self-adaptive distributed coordination. Distributed and autonomic minimum spanning tree algorithms over VCube were proposed in [1] (*in Portuguese*), which was later employed in several higher-level distributed algorithms such as distributed k-mutual exclusion [36] and majority quorum systems [37].

More recently [12, 38], VCube has be specified as an unreliable failure detector [10, 11, 39, 40]. Those works provide a framework for unifying distributed diagnosis and unreliable failure detectors. An eventually perfect hierarchical failure detector (Diamond-P-VCube) was proposed in [41]. A leader election algorithm based on the VCube assuming a crash-recovery model was addressed in [42]. Additionally, VCube have also been applied as a fault-tolerant parallel sorting method [43] for building fault-tolerant parallel algorithms.

Atomic broadcast has also been explored in the context of VCube. Leaderless and hierarchical atomic broadcast algorithms relying on autonomic VCube spanning trees were proposed in [44, 45]. In addition, VCube has been adopted as a substrate for blockchain systems. The permissioned vCubeChain architecture leverages the VCube topology to provide scalable and fault-tolerant consensus mechanisms [46]. Furthermore, efficient synchronization of conflict-free replicated data types (CRDTs) over VCube-based publish/subscribe systems has been proposed, demonstrating the suitability of the topology for large-scale eventual consistency services [47].

7. Conclusions

This work presented a solution for creating autonomic spanning trees in distributed systems subject to crash faults. The system's processes are organized in a logical topology called VCube. The trees are dynamically generated from a source process and reconfigure autonomically as processes fail. Two broadcast algorithms were implemented using the proposed tree strategy: the first for best-effort broadcast and the second for reliable broadcast. Simulation results are presented, demonstrating the efficiency of the proposed solutions and highlighting their scalability.

Future work includes extending the proposed algorithms to support the recovery of processes. Assuming a model with network partitions is also left as future work. Furthermore, an implementation of a network service [48, 49, 50] for message dissemination based on the proposed spanning tree algorithms that can be invoked by arbitrary applications over a network is also planned.

Acknowlegments

This work was partially supported by the Brazilian Research Council (CNPq) Grant 305108/2025-5.

References

[1] L. A. Rodrigues, E. P. Duarte Jr, and L. Arantes, "Árvores geradoras mínimas distribuídas e autonômicas," *Anais do XXXII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuıdos (SBRC), Porto Alegre, RS, Brasil. SBC*, 2014.

[2] C. T. Ryan, R. L. Smith, and M. A. Epelman, "Minimum spanning trees in infinite graphs: theory and algorithms," *SIAM Journal on Optimization*, vol. 34, no. 3, pp. 3112–3135, 2024.

[3] M. Elkin, "A faster distributed protocol for constructing a minimum spanning tree," *J. Comput. Syst. Sci.*, vol. 72, no. 8, pp. 1282–1308, Dec. 2006.

[4] D. England, B. Veeravalli, and J. Weissman, "A robust spanning tree topology for data collection and dissemination in distributed environments," *IEEE Trans. Parallel Distr. Syst.*, vol. 18, no. 5, pp. 608–620, 2007.

[5] S. Dahan, L. Philippe, and J. Nicod, "The distributed spanning tree structure," *IEEE Trans. on Parallel and Distributed Systems*, vol. 20, no. 12, pp. 1738–1751, 2009.

[6] F. C. Gärtner, "A survey of self-stabilizing spanning-tree construction algorithms," Swiss Federal Institute of Technology (EPFL), Tech. Rep., 2003.

[7] J. Kephart and D. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[8] E. P. Duarte Jr. and T. Nanya, "A hierarchical adaptive distributed system-level diagnosis algorithm," *IEEE Transactions on Computers*, vol. 47, no. 1, pp. 34–45, Jan. 1998. [Online]. Available: https://doi.org/10.1109/12.656078

[9] E. P. Duarte, L. C. E. Bona, and V. K. Ruoso, "VCube: a provably scalable distributed diagnosis algorithm," in *Proc. 5th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ser. ScalA '14. IEEE Press, 2014, p. 17–22. [Online]. Available: https://doi.org/10.1109/ScalA.2014.14

[10] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM (JACM)*, vol. 43, no. 2, pp. 225–267, 1996.

[11] M. Reynal, "A short introduction to failure detectors for asynchronous distributed systems," *ACM SIGACT News*, vol. 36, no. 1, pp. 53–70, 2005.

[12] E. P. D. Jr., L. A. Rodrigues, E. T. Camargo, and R. Turchetti, "A distributed system-level diagnosis model for the implementation of unreliable failure detectors," 2022. [Online]. Available: https://arxiv.org/abs/2210.02847

[13] R. G. Gallager, P. A. Humblet, and P. M. Spira, "A distributed algorithm for minimum-weight spanning trees," *ACM Trans. Program. Lang. Syst.*, vol. 5, pp. 66–77, 1983.

[14] F. C. Freiling, R. Guerraoui, and P. Kuznetsov, "The failure detector abstraction," *ACM Comput. Surv.*, vol. 43, pp. 9:1–9:40, Feb. 2011.

[15] V. Hadzilacos and S. Toueg, *Fault-tolerant broadcasts and related problems*. USA: ACM Press/Addison-Wesley Publishing Co., 1993, p. 97–145.

[16] R. Guerraoui and L. Rodrigues, Eds., *Introduction to Reliable Distributed Programming*. Berlin, Germany: Springer-Verlag, 2006.

[17] L. A. Rodrigues, "Fault-tolerant broadcast algorithms for the virtual hypercube topology," in *Student Forum - DSN-W'43*, 2013, pp. 1–4. [Online]. Available: https://doi.org/10.1109/DSNW.2013.6615520

[18] P. Urbán, X. Défago, and A. Schiper, "Neko: A single environment to simulate and prototype distributed algorithms," *Journal of Inf. Science and Eng.*, vol. 18, no. 6, pp. 981–997, November 2002.

[19] J. B. Kruskal Jr., "On the shortest spanning subtree of a graph and the traveling salesman problem," in *American Mathematical Society*, 1956, pp. 48–50.

[20] R. C. Prim, "Shortest connection networks and some generalizations," *Bell System Technical Journal*, vol. 36, no. 6, pp. 1389–1401, nov. 1957.

[21] Y. Dalal, "A distributed algorithm for constructing minimal spanning trees," *Software Engineering, IEEE Transactions on*, vol. SE-13, no. 3, pp. 398–405, 1987.

[22] D. Avresky, "Embedding and reconfiguration of spanning trees in faulty hypercubes," *IEEE Trans. Parallel and Distributed Systems*, vol. 10, no. 3, pp. 211–222, 1999.

[23] P. Flocchini, T. Mesa Enriquez, L. Pagli, G. Prencipe, and N. Santoro, "Distributed minimum spanning tree maintenance for transient node failures," *IEEE Trans. Comput.*, vol. 61, no. 3, pp. 408–414, 2012.

[24] E. P. Duarte and L. E. De Bona, "A dependable snmp-based tool for distributed network management," in *Proceedings International Conference on Dependable Systems and Networks*. IEEE, 2002, pp. 279–284. [Online]. Available: https://doi.org/10.1109/DSN.2002.1028911

[25] A. Brawerman and E. P. Duarte Jr, "An isochronous testing strategy for hierarchical adaptive distributed system-level diagnosis," *Journal of Electronic Testing*, vol. 17, no. 2, pp. 185–195, 2001.

[26] E. P. Duarte, L. C. Albini, A. Brawerman, and A. L. Guedes, "A hierarquical distributed fault diagnosis algorithm based on clusters with detours," in *2009 Latin American Network Operations and Management Symposium*. IEEE, 2009, pp. 1–6.

[27] L. C. E. Bona, E. P. Duarte, and K. V. O. Fonseca, "A distributed virtual hypercube algorithm for maintaining scalable and dynamic network overlays," *Concurr. Comput.: Pract. Exper.*, vol. 27, no. 7, p. 1658–1678, May 2015. [Online]. Available: https://doi.org/10.1002/cpe.3321

[28] L. C. Bona, K. V. Fonseca, and E. P. Duarte, "Hyperbone: A scalable overlay network based on a virtual hypercube," in *NOMS 2008-2008 IEEE Network Operations*

and Management Symposium. IEEE, 2008, pp. 1025–1030. [Online]. Available: https://doi.org/10.1109/CCGRID.2008.61

[29] E. Duarte, A. Brawerman, and L. C. P. Albini, "An algorithm for distributed hierarchical diagnosis of dynamic fault and repair events," in *Proc. 7th Int'l Conference on Parallel and Distributed Systems*. IEEE, 2000, pp. 299–306. [Online]. Available: https://doi.org/10.1109/ICPADS.2000.857711

[30] L. A. Rodrigues, L. Arantes, and E. P. Duarte, "An autonomic implementation of reliable broadcast based on dynamic spanning trees," in *2014 Tenth European Dependable Computing Conference*, 2014, pp. 1–12. [Online]. Available: https://doi.org/10.1109/EDCC.2014.31

[31] É. Jeanneau, L. A. Rodrigues, L. Arantes, and E. P. Duarte Jr, "An autonomic hierarchical reliable broadcast protocol for asynchronous distributed systems with failure detection," *Journal of the Brazilian Computer Society*, vol. 23, no. 1, p. 15, 2017. [Online]. Available: https://doi.org/10.1186/s13173-017-0064-9

[32] J. a. P. de Araujo, L. Arantes, E. P. D. Júnior, L. A. Rodrigues, and P. Sens, "A communication-efficient causal broadcast protocol," in *Proc. 47th International Conference on Parallel Processing*, ser. ICPP '18. New York, NY, USA: ACM, 2018. [Online]. Available: https://doi.org/10.1145/3225058.3225121

[33] J. P. de Araujo, L. Arantes, E. P. Duarte, L. A. Rodrigues, and P. Sens, "VCube-PS: A causal broadcast topic-based publish/subscribe system," *Journal of Parallel and Distributed Computing*, vol. 125, pp. 18–30, 2019. [Online]. Available: https://doi.org/10.1016/j.jpdc.2018.10.011

[34] J. P. De Araujo, L. Arantes, E. P. Duarte, L. A. Rodrigues, and P. Sens, "A publish/subscribe system using causal broadcast over dynamically built spanning trees," in *29th Int'l Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 2017, pp. 161–168. [Online]. Available: https://doi.org/10.1109/SBAC-PAD.2017.28

[35] L. A. Rodrigues, E. P. Duarte, J. P. de Araujo, L. Arantes, and P. Sens, "Bundling messages to reduce the cost of tree-based broadcast algorithms," in *8th Latin-American Symposium on Dependable Computing*, 2018, pp. 115–124. [Online]. Available: https://doi.org/10.1109/LADC.2018.00022

[36] L. A. Rodrigues, E. P. Duarte, and L. Arantes, "A distributed k-mutual exclusion algorithm based on autonomic spanning trees," *J. Parallel and Distributed Computing*, vol. 115, pp. 41–55, 2018. [Online]. Available: https://doi.org/10.1016/j.jpdc.2018.01.008

[37] L. A. Rodrigues, L. Arantes, and E. P. Duarte, "An autonomic majority quorum system," in *30th Int. Conference on Advanced Information Networking and Applications (AINA)*, 2016, pp. 524–531. [Online]. Available: https://doi.org/10.1109/AINA.2016.73

[38] E. P. Duarte, L. A. Rodrigues, E. T. Camargo, and R. C. Turchetti, "The missing piece: a distributed system-level diagnosis model for the implementation of unreliable failure detectors," *Computing*, vol. 105, no. 12, p. 2821–2845, Aug. 2023. [Online]. Available: https://doi.org/10.1007/s00607-023-01211-8

[39] R. C. Turchetti and E. P. Duarte, "Implementation of failure detector based on network function virtualization," in *IEEE International Conference on Dependable Systems and Networks Workshops*. IEEE, 2015, pp. 19–25. [Online]. Available: https://doi.org/10.1109/DSN-W.2015.30

[40] R. C. Turchetti, E. P. Duarte Jr, L. Arantes, and P. Sens, "A QoS-configurable failure detection service for internet applications," *Journal of Internet Services and Applications*, vol. 7, no. 1, p. 9, 2016. [Online]. Available: https://doi.org/10.1186/s13174-016-0051-y

[41] G. Stein, L. A. Rodrigues, E. P. Duarte Jr., and L. Arantes, "Diamond-p-vcube: An eventually perfect hierarchical failure detector for asynchronous distributed systems," in *Proc. of the 12th Latin-American Symposium on Dependable and Secure Computing*, ser. LADC '23. New York, NY, USA: ACM, 2023, p. 40–49. [Online]. Available: https://doi.org/10.1145/3615366.3615420

[42] L. A. Rodrigues, A. E. S. Freitas, E. P. Duarte Jr., and V. Fulber-Garcia, "A hierarchical adaptive leader election algorithm for crash-recovery distributed systems," in *Proce. 13th Latin-American Symposium on Dependable and Secure Computing*, ser. LADC '24. New York, NY, USA: ACM, 2024, p. 136–145. [Online]. Available: https://doi.org/10.1145/3697090.3697102

[43] E. T. Camargo and E. P. D. Junior, "Algorithm-based fault-tolerant parallel sorting," *Int. J. Crit. Comput.-Based Syst.*, vol. 11, no. 1–2, p. 2–29, Jan. 2024. [Online]. Available: https://doi.org/10.1504/ijccbs.2024.139097

[44] L. V. Ruchel, L. A. Rodrigues, R. C. Turchetti, L. Arantes, E. P. Duarte Jr., and E. T. Camargo, "A leaderless hierarchical atomic broadcast algorithm," in *Proc. 11th Latin-American Symposium on Dependable Computing*, ser. LADC '22. New York, NY, USA: ACM, 2023, p. 61–66. [Online]. Available: https://doi.org/10.1145/3569902.3569914

[45] L. V. Ruchel, E. Tavares de Camargo, L. A. Rodrigues, R. C. Turchetti, L. Arantes, and E. Procópio Duarte, "Scalable atomic broadcast: A leaderless hierarchical algorithm," *Journal of Parallel and Distributed Computing*, vol. 184, p. 104789, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731523001594

[46] A. E. S. Freitas, L. A. Rodrigues, and E. P. Duarte, "vcubechain: A scalable permissioned blockchain," *Ad Hoc Netw.*, vol. 158, no. C, May 2024. [Online]. Available: https://doi.org/10.1016/j.adhoc.2024.103461

[47] L. F. Galesky, L. A. Rodrigues, E. P. Duarte Jr., and L. Arantes, "Efficient synchronization of crdts using vcube-ps," in *Proc. 12th Latin-American Symposium on Dependable and Secure Computing*, ser. LADC '23. New York, NY, USA: ACM, 2023, p. 50–59. [Online]. Available: https://doi.org/10.1145/3615366.3615421

[48] G. Venâncio, R. C. Turchetti, and E. P. Duarte Jr, "Nfv-coin: Unleashing the power of in-network computing with virtualization technologies," *Journal of Internet Services and Applications*, vol. 13, no. 1, pp. 46–53, 2022. [Online]. Available: https://doi.org/10.5753/jisa.2022.2342

[49] G. Venâncio, R. C. Turchetti, and E. P. Duarte, "NFV-RBCcast: Enabling the network to offer reliable and ordered broadcast services," in *2019 9th Latin-American Symposium on Dependable Computing (LADC)*. IEEE, 2019, pp. 1–10. [Online]. Available: https://doi.org/10.1109/LADC48089.2019.8995681

[50] R. C. Turchetti and E. P. Duarte Jr, "Nfv-fd: Implementation of a failure detector using network virtualization technology," *International Journal of Network Management*, vol. 27, no. 6, p. e1988, 2017. [Online]. Available: https://doi.org/10.1002/nem.1988