A Leaderless Hierarchical Atomic Broadcast Algorithm

Research Paper

Lucas V. Ruchel
Federal Institute of Parana
Cascavel, PR, Brazil
lucas.ruchel@ifpr.edu.br

Luiz Antonio Rodrigues
Western Parana State University
Cascavel, PR, Brazil
luiz.rodrigues@unioeste.br

Rogério C. Turchetti
Federal University of Santa Maria
Santa Maria, RS, Brazil
turchetti@redes.ufsm.edu.br

Luciana Arantes
Sorbonne Université/CNRS/Inria/LIP6
Paris, France
luciana.arantes@lip6.fr

Elias P. Duarte Jr.
Federal University of Parana
Curitiba, PR, Brazil
elias@inf.ufpr.br

Edson T. de Camargo
Federal Technology University of Parana
Toledo, PR, Brazil
edson@utfpr.edu.br

ABSTRACT

This work presents *LHABcast*, a Leaderless Hierarchical Atomic Broadcast algorithm that is fully decentralized and allows all processes to send messages simultaneously. Processes use autonomic spanning trees built on top of the VCube overlay network to propagate message timestamps, i.e., local sequence numbers. Processes can fail by crashing. After each process receives the timestamps, it can make a decision about the message delivery sequence. The algorithm is shown to be correct.

CCS CONCEPTS

• **Networks → Network algorithms**; • **Computing methodologies → Distributed computing methodologies**; **Distributed algorithms**.

KEYWORDS

Atomic Broadcast, VCube, Leaderless, Hipercube, Virtual Topology

ACM Reference Format:
Lucas V. Ruchel, Luiz Antonio Rodrigues, Rogério C. Turchetti, Luciana Arantes, Elias P. Duarte Jr., and Edson T. de Camargo. 2022. A Leaderless Hierarchical Atomic Broadcast Algorithm: *Research Paper*. In *Proceedings of 11th Latin-American Symposium on Dependable Computing (LADC '22).* ACM, New York, NY, USA, 6 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Agreement is an essential problem of distributed systems that require any kind of collaboration between processes. Atomic broadcast, also known as total order broadcast, is an alternative agreement strategy that ensures that all processes in the system receive the same set and order of messages [3]. As a communication primitive, it is used to guarantee strong consistency while tolerating faults. Atomic broadcast algorithms typically rely on a leader, including the Paxos-based approaches [6] and *Raft* [8]. A leader can quickly become a bottleneck, and its failure directly affects the latency and throughput of the entire system.

Some atomic broadcast algorithms are based on destination agreement [3]. In this approach, the message delivery order is the result of an agreement between the destination processes. There are three variants: (1) agreement on a set of messages; (2) agreement on the adoption of a proposed message order; and (3) agreement on a sequence number. Both the first and second variants are implemented using a consensus algorithm, such as Paxos [6], and atomic confirmation protocols, such as those proposed in [7, 14]. Generally, they elect a leader who acts as a coordinator and is responsible for conducting rounds of consensus on the order of message delivery.

The third variant is completely decentralized: Processes agree on a unique (but non-consecutive) sequence number for each message and use it to determine the order of delivery. For example, *AllConcur* [11, 9] and *AllConcur+* [10] are leaderless atomic broadcast algorithms that use a digraph as an overlay network to broadcast messages. They are fully decentralized and assume perfect failure detectors P and $\diamond\mathcal{P}$, respectively. Although a larger number of messages are required for implementation, the third approach is distinguished by the fact that it allows all processes to broadcast messages simultaneously without compromising scalability, since there is no bottleneck caused by a central coordinator.

In this work, we present a fully decentralized atomic broadcast algorithm called *LHABcast*, where all processes send their messages simultaneously over autonomic, dynamically constructed spanning trees. All processes execute the same algorithm, no process is distinguished as a leader. The order of message delivery is determined by a sequence number (also called a timestamp) defined by the local logical clock of each process. In addition, the tree is rebuilt when a process crashes without the need for additional messages. The trees are created and managed using the VCube virtual topology [4]. The latter is a hypercube when all processes are fault-free, but it reconfigures itself when processes fail and maintains several logarithmic properties.

Several broadcast algorithms have been proposed for the VCube [13, 12, 5, 2]. The first one [12] defines both a best-effort and a reliable broadcast algorithm assuming a synchronous system model. These algorithms also dynamically build autonomic spanning trees

LADC '22, November 21–24, 2022, Fortaleza, BR
© 2022 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/XXXXXXX.XXXXXXX

that connect fault-free processes. VCube includes a failure detection service. After a process crash is detected, each process seamlessly redefines its tree edges. In [5], the authors used a similar approach to develop a reliable hierarchical broadcast algorithm, but assuming an asynchronous system. In order to ensure message delivery to those processes that are falsely suspected of having crashed, the algorithm keeps sending all messages to suspect processes in the tree. *LHABcast* stands out from these other broadcast algorithms as it is the first atomic broadcast proposed for the VCube.

The remainder of this paper is organized as follows. The system model is defined in Section 2. Section 3 presents the proposed algorithm and a proof of its correctness. Finally, Section 4 concludes the paper and discusses future work.

2 SYSTEM MODEL

We assume a distributed system that consists of a finite set P of $n > 1$ processes that communicate through message passing. The network is fully connected: any pair of processes can communicate with each other without employing intermediates. Processes can fail by crashing and, once a process crashes, it does not recover. If a process never crashes during a run, it is considered *correct*; otherwise, it is considered to be *faulty*. Links are reliable, i.e., messages exchanged between any two correct processes are never lost, corrupted or duplicated, and FIFO. The system is synchronous, i.e., relative processor speeds and message transmission delays are bounded [1].

Processes are organized on a virtual hierarchical topology, the VCube [4]. When all processes are correct, the VCube is a hypercube. Processes of a d-dimensional hypercube have identifiers consisting of d bits. Two processes are virtually connected whenever their binary addresses differ by only one bit. The major advantage of the VCube is that as processes crash, the virtual topology dynamically reorganizes itself, keeping multiple logarithmic properties.

VCube is a distributed diagnosis algorithm which organizes processes in increasingly large clusters. A diagnosis algorithm is used to determine which processes in the system are faulty and which are not. The virtual edges of a VCube correspond to tests that correct processes execute on each other. VCube allows processes to obtain diagnostic information from any process tested correct. Diagnostic information is timestamped to allow processes to distinguish recent events. A tester process i executes tests on a cluster $c_{i,s}$ of processes of size 2^{s-1}, such that $s = 1 \ldots \log_2 n$. Function $C_{i,s}$ (Equation 1) returns the ordered list of processes of each cluster:

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

where $\oplus$ is the bitwise exclusive or operator (*xor*). Table 1 shows the $c_{i,s}$ function for 8 processes. In order to determine the edges of the virtual topology, for each node i, there is an edge (j, i), such that j is the first fault-free node in $c_{i,s}$, $s = 1 \ldots \log_2 n$. After a process detects that any other process has crashed, the set of edges (tests) is recomputed. For instance, in the example shown in Figure 1, process p_4 originally tests process p_0, but after it has crashed, the tester of process p_0 in the cluster with $s = 3$ is p_5.

Table 1: The $C_{i,s}$ table for 8 processes

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

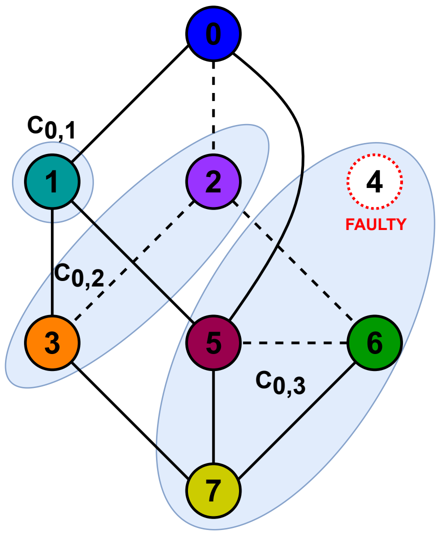


Figure 1: Clusters of a VCube with dimension $d = 3$.

3 THE *LHABCAST* ALGORITHM

Atomic Broadcast ensures reliable broadcast while keeping the total order of messages delivered, i.e., all correct processes deliver the same set of messages in the same order. For this purpose, the following properties must be fulfilled [3]:

- *Validity*: if a correct process broadcasts a message m, then it eventually delivers m;
- *Integrity*: each message m is delivered once and only if it was previously broadcast (no creation);
- *Agreement*: if one correct process delivers m, then all correct processes deliver m;
- *Total order*: if two correct processes p and q deliver two messages m_1 and m_2, then p delivers m_1 before m_2 if and only if q delivers m_1 before m_2.

LHABcast is a leaderless hierarchical atomic broadcasts that disseminates all messages over the VCube topology described in Section 2. When process p broadcasts a message m, it becomes the root of a spanning tree over which m's timestamp will be sent and used to order messages. The tree is built dynamically and autonomously. Each correct processes p determines its edges, as described in the previous section. As soon as each process detects the failure of another process, the tree is reconfigured without any additional messages.

3.1 Algorithm Description

Briefly described, the *LHABcast* algorithm works as follows. A process i (source) broadcasts a timestamped message m to its neighbors (its children in the tree). Upon receiving m, each process j, which is not a leaf, becomes itself the root of a subtree and forwards m to its children. When a leaf of the tree receives m, it sends an acknowledgment message (*ACK*) back to the process that sent m to it (its parent). Every process, except the leaves and the source, after receiving an *ACK* from all of its children, sends an *ACK* back to its parent. Thus, when a process receives *ACK* from all its child processes, all correct processes in that cluster have surely received m too. When a process receives m for the first time, it updates m's timestamp before forwarding it. Message m is considered deliverable by a process provided it has received all timestamps assigned to it by the correct processes and there is no received message with a lower timestamp than m' highest timestamp ($sn(m)$). Message m will be delivered respecting the total order ($sn(m), source$).

Each process i that executes the *LHABcast* algorithm keeps the following local variables:

- LC_i: local logical clock that uniquely identifies messages broadcast by i. It is used to keep track of which messages have already been delivered by a particular source process;
- TS_i: *timestamp* used to establish the total order;
- $correct_i$: set of processes that are considered correct by process i;
- $last_i[n]$: the last ordered message delivered from each source process;
- $pending\_ack_i$: the set of pending ACKs. For each message $TREE\langle m, T\rangle$ (re)-transmitted by i from a process j to a process k, an element $\langle j, k, \langle m, T\rangle\rangle$ is added to this set where T is a set of *timestamps*;
- $received_i$: set of messages received by process i that cannot be delivered yet to the application. Each element in this set contains $\langle m, ts_k(m)\rangle$, i.e., the message m to be ordered and the timestamp $ts_k(m)$ associated with m by each process k;
- $stamped_i$: set of messages m for which all *timestamps* $ts_k(m)$ have been received, but cannot be delivered yet because they are out of order with respect to the other messages $m' \neq m$.

Message in the algorithm can be of the following types:

- $TREE\langle m, T\rangle$, which contains the message m to be transmitted and the set T of *timestamps* $ts_k(m)$. T has 1 to $log_2 n$ *timestamps* (the longest path in the tree).
- $ACK\langle m, T\rangle$, used to acknowledge the receipt of a message $TREE\langle m, T\rangle$.

Each message m has two parameters: (i) the identifier of the source process *m.src*; and (ii) a sequence number *m.seq*, set from the local logical clock LC_i of the source process i, which is used to uniquely and sequentially identify the messages sent by each process.

Algorithm 1 presents the pseudo-code of the proposed solution. Broadcasts start with the A-Broadcast(m) function (line 7), which adds the source and the local logical clock LC_i to the message. The Forward function is used to send *TREE* messages to the neighbors of process i. The parameters of this function are the number of *clusters* s of i to which the message m and the *timestamp* $ts_i(m)$ must be forwarded. In this case, $s = \log_2 n$, which is the dimension of the hypercube and corresponds to the number of clusters of the system In the Forward function, the message m is forwarded to the neighboring processes of i based on the value of s (line 15).

On receiving a message *TREE*, process i checks whether j is in the set of correct processes (line 19). If it is not, the message is ignored. Otherwise, i updates its local *timestamp* TS_i based on the received timestamp from process j (line 21). If the message was received for the first time, i.e., is not included in either the delivered, the received nor the stamped sets, the message is sent with $ts_i(m)$ to those processes in the tree of i that do not overlap with the tree of process j (line 25). To the others, m is forwarded with *timestamp* T(line 30), so that when m is first received, $ts_i(m)$ is aggregated to T (line 24).

Considering a three-dimensional VCube, Figure 2 shows the flow of sent messages from process p_0 in its spanning tree (see dark blue arrow), i.e., the forwarding of message m to p_0's neighbors in *clusters* 1, 2 and 3. The Figure 2a shows the behavior of process p_1 when it receives message m for the first time. Process p_1 recognizes through $\{cluster_1(0) - 1 = 0\}$ that it is in the last level of the p_0 tree (leaf). Therefore, p_1 broadcasts its *timestamp* to the processes $\{0, 3, 5\}$ (represented in the arrows by ts_1). Figure 2b highlights process p_2, which is a branch in p_0's tree ($cluster_2(0) - 1 = 1$). Therefore, p_2 forwards m with it *timestamp* to processes $\{0, 6\}$. In the p_0 tree, p_2 adds its own *timestamp* to m and forwards it to process p_3. Thus, m now contains the *timestamps* of p_0 and p_2.

Upon receiving the message *TREE* and forwarding the messages to the next process of the respective (sub)tree(s), process i adds each *timestamp* contained in T to the list of received *timestamps* of m (line 29) and calls the function CheckDeliverable(m). This function checks if the timestamps of all correct processes for m (line 39) have been received. If it is the case, i computes the highest *timestamp* of m (line 40), denoted $sn(m)$, and the function Deliver($m, sn(m)$) is called (line 41). Then, i updates the register of delivered messages $last_i$ and all *timestamps* associated with m are removed from the set $received_i$.

In the Deliver function, message m is added to the set of stamped messages ($stamped_i$) with the highest received *timestamp* – $sn(m)$ (line 47). On the other hand, a message m' in $stamped_i$ is added to the deliverable set ($deliverable_i$) only if there is no received message m'' in $received_i$ whose *timestamp* is lower than m' (line 49). Otherwise, m' should be delivered after m'' (line 49). Messages marked for delivery (*deliverable*) are ordered by the *timestamp* associated with m and the identifier of the originating process of m (*m.src*) (line 52). In this way, messages are delivered in order and then removed from $stamped_i$.

Confirmation of receipt of the message is performed by the CheckAcks function. The tuple $\langle p, *, \langle m, T\rangle\rangle$ represents the waiting *ACK*s for message m with *timestamp* received from p and forwarded to any process ($*$) (line 54). For example, in Figure 2a, as p_1 is a leaf in p_0's tree, it has not forwarded a message with p_0's timestamp. Therefore, an *ACK* can be sent to p_0 immediately. The same happens with p_3 relative to p_2. Nevertheless, after receiving the *ACK* from p_3, p_2 removes the message from the set of pending messages and notify p_0, which originally sent the message with *timestamp* T, with an *ACK*.

The VCube diagnostic algorithm performs the detection and notification of process failures. For all processes, when i receives

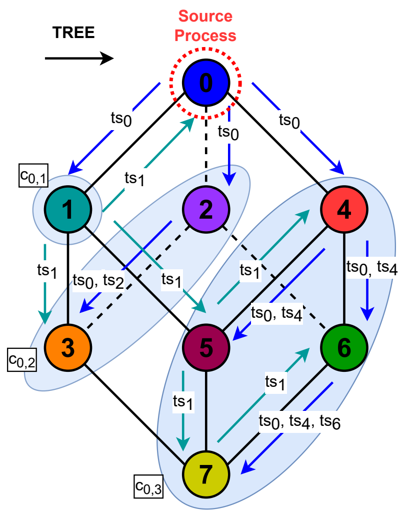
Algorithm 1 LHABcast Atomic Broadcast at Process i

```
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($i, \log_2 n, TREE\langle m, \{ts_i(m)\}\rangle$)

14: procedure FORWARD($from, s, TREE\langle m, T\rangle$)
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 ($last_i[m.src] = \perp$ or $m.seq > last_i[m.src]$)
            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($j, cluster_i(j) - 1, TREE\langle m, T\rangle$)
31:     CHECKDELIVERABLE($m$)
32:     CHECKACKS($j, \langle m, T\rangle$)

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

38: procedure CHECKDELIVERABLE($m$)
39:     if received $\langle m, ts_k(m)\rangle$ from all $k \in correct_i$
            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 $last_i[m.src] = \perp$ or $m'.seq = last_i[m.src] + 1$ 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:     Deliver all messages in $deliverable$ at order $(sn(m), m.src)$
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($m$)
```

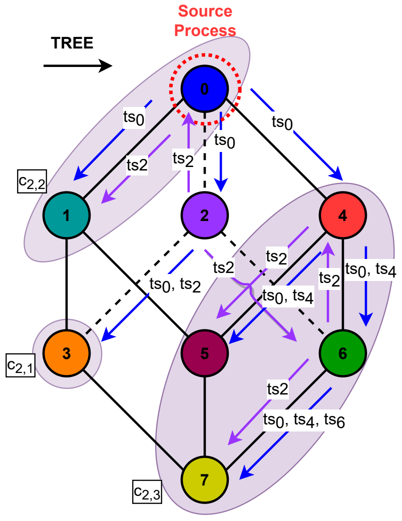
notification that j crashed (line 57), j is removed from the list of correct processes. Thus, for each message m forwarded to j not confirmed, i.e., there is m sent to j in $pending\_acks_i$ (line 59), this message m must be forwarded to the next correct process in the *cluster* to which j belongs, if it exists (the $FF\_neighbor_i(s)$ function determines this information, where $s = cluster_i(j)$) (line 61). For example, consider the three-dimensional hypercube in Figure 2c where p_0 is the source of the message. If p_4 fails during the broadcast, p_0 will not receive a confirmation from that cluster. When p_0 detects the failure, it forwards its *timestamp* to the next correct process of this *cluster* (p_5). If there are no more correct processes in the *cluster*, the CHECKACKS function is called to check if there are any pending messages to be forwarded from the p process that sent m to the i process (line 65). Also, the process i may not have received the *timestamp* from j. In this case i will keep waiting until the timestamp is received or j is detected as faulty. Therefore, for each m message in $received_i$, the function CHECKDELIVERABLE (line 67) is called to check if m can be delivered, since the *timestamp* of j is no longer needed.

3.2 Proof of correctness

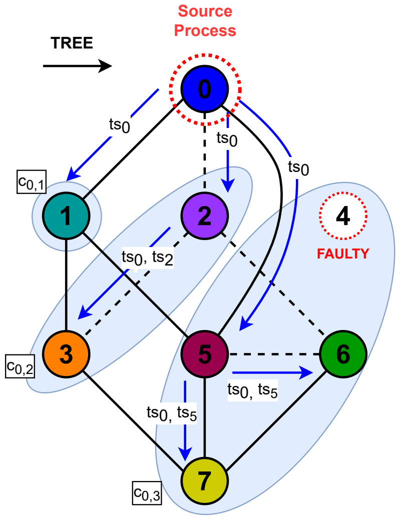
In this section, we will prove that Algorithm 1 ensures the validity, integrity, agreement, and total order properties of an atomic broadcast algorithm.



(a) p_1 as leaf



(b) p_2 as forwarder



(c) Tree when p_4 is faulty

Figure 2: Message forwarding – illustrative examples.

Lemma 3.1 (Validity). *Algorithm 1 ensures that if a correct process* A-Broadcast *a message m, then it eventually delivers m.*

Proof. When process i broadcasts a message m, it includes $\{\langle m, ts_i(m)\rangle\}$ into the set $received_i$ (line 12) and forwards m to all correct neighbors in the VCube topology (line 15). For each process j to which i sent m, i added a pending ack in $pending\_ack_i$. If j is correct, it responds with an *ACK* message (CheckAcks procedure) and i removes the pending ack in $pending\_ack_i$ on line 35. If j is faulty, i will eventually detect the crash and remove the pending ack in line 35.

As a result, all outstanding acks for m will eventually be removed from $pending\_ack_i$ and, after receives the timestamps to m from all correct process (line 39), i will deliver m on line 41. □

Lemma 3.2 (Integrity). *Algorithm 1 ensures that each message m is delivered at most once and only if it was previously* A-Broadcast *by a sender process i.*

Proof. A process only delivers a message m if it either has broadcast m itself (Lemma 3.1) or if m is in its $received_i$ set (line 41). Messages broadcast by the sender are added to $received_i$ in line 12. Messages received from other processes are included into the $received_i$ in line 29. Since links are reliable and do not generate messages, a message is delivered only if it was previously sent (no messages are generated).

To show that there is no duplication of messages, let us consider two cases:

- **source(m) = i**. Process i called A-Broadcast broadcasting m. As proved in Lemma 3.1, i will deliver m on line 41. Since the procedure A-Broadcast is called only once for any given message, the only way that i could deliver m a second time would be on executing line 41. Since $last_i[i]$ was set to m in line 44 after the deliver on m, it follows that m will never be qualified to pass the test again.
- **source(m) ≠ i**. The process i is not the source of the message m, and did not call the procedure A-Broadcast with m. Therefore, the only way for i to deliver m is in line 41. Before i delivers m for the first time, it sets $last_i[m.src]$ to m in line 44. Thus, m will never pass the test again, and i may therefore deliver m at most once.

□

Lemma 3.3 (Agreement). *Algorithm 1 ensures that if a correct process delivers a message m, all correct processes eventually deliver m.*

Proof. Let m be a message broadcast by process i. We consider two cases:

- ***i* is correct**. It can be shown by induction that every correct process receives m.
  As a basis for induction, consider the case that $n = 2$ and $P = \{i, j\}$. It follows that $c_{i,1} = \{j\}$. Therefore, i will send m to j on line 15. If j is correct, it will eventually receive m since the links are reliable, and will deliver m on line 41. Process i will also deliver m due to the validity property.
  We now need to prove that if every correct process receives m for $n = 2^k$, this is also the case for $n = 2^{k+1}$. The system of size 2^{k+1} can be seen as two subsystems $P_1 = \{i\} \cup \bigcup_{x=1}^{k} c_{i,x}$ and $P_2 = c_{i,k+1}$ such that $|P_1| = |P_2| = 2^k$.
  The procedure Forward ensures that for each $s \in [1, k+1]$, i will send m to at least one process in $c_{i,s}$. Let j be the first process in $c_{i,k+1}$. If j is correct, it will eventually receive m. Process i will continue to do so until it has sent the *TREE* message to a non-suspect process in $c_{i,k+1}$.

If j is faulty and if i detects the crash only after the broadcast, the *Send* procedure will be called again in line 61, which ensures once again that i sends the message to a not faulty suspicious process in $c_{i,k+1}$. As a result, unless all the processes in $c_{i,k+1}$ are faulty, at least one correct process in $c_{i,k+1}$ will eventually receive m. This correct process will then broadcast m to the rest of the subsystem P_2 on the line. Since a correct process broadcasts m over both subsystems P_1 and P_2, the two of them have size 2^k, it follows that every correct process in P will eventually receive m.

- **i is faulty**. If i crashes before m is sent to any process, then no correct process delivers m and the agreement property is verified. If i crashes after having broadcast m to all its neighbors, then the broadcast of m happens as if i was correct. On the other hand, if i crashes while sending m and a correct process j receives m, then j will eventually detect the failure of i. If j detects the crash before receiving m, it waits to receive m from other correct processes because m is forwarded with local timestamp to all correct processes on lines 25 and 30. Since j is correct, each correct process will eventually receive m.

□

Lemma 3.4 (Total Order). *Algorithm 1 ensures that if two correct processes p and q both deliver messages m and m', then p delivers m before m', if and only if q delivers m before m'.*

Proof. We prove from Lemma 3.3 that all correct processes receive all broadcast messages. If we consider two messages m and m', we have two possible scenarios:

- $m.src = m'.src$. If both messages are from the same source process and m was sent before m', so $m.seq < m'.seq$ and $ts(m) < ts(m')$. By agreement, both messages and their respective timestamps are included in $received_i$ by all correct processes. Finally, both messages are delivered in the same order in line 52.
- $m.src! = m'.src$. When m and m' are sent by different processes, there is no direct precedence between their timestamps. Each correct process receives the messages in any order and assigns the timestamp according to the local order. Once all correct processes have received all timestamps from each message, both messages are delivered in the same order in line 52.

□

Theorem 3.5. *Algorithm 1 implements an atomic broadcast.*

Proof. The proof follows directly from Lemmas 3.1, 3.2, 3.3 and 3.4. □

4 CONCLUSION

This paper has presented a hierarchical, autonomous and fully decentralized atomic broadcast algorithm. It uses a scalable strategy that allows processes to deliver messages in the same order without the need for a leader. In the event of a process failure, the hierarchical approach relies on the properties of VCube to automatically reorganize non-faulty processes into a new tree.

In the future, we will investigate optimizations aimed at reducing the number of messages required to fully order application messages, thereby improving performance. Other research directions include evaluating the impact of changing the *LHABcast* system model when the synchrony assumption is relaxed, and experimentally comparing *LHABcast* with other solutions such as *AllConcur* and *AllConcur+*.

REFERENCES

[1] Tushar Deepak Chandra and Sam Toueg. 1996. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43, 2, (Mar. 1996), 225–267. doi: 10.1145/226643.226647.

[2] João Paulo de Araujo, Luciana Arantes, Elias Procópio Duarte, Luiz A. Rodrigues, and Pierre Sens. 2019. Vcube-ps: A causal broadcast topic-based publish/subscribe system. *J. Parallel Distributed Comput.*, 125, 18–30. doi: 10.1016/j.jpdc.2018.10.011.

[3] Xavier Défago, André Schiper, and Péter Urbán. 2004. Total order broadcast and multicast algorithms: taxonomy and survey. *ACM Comput. Surv.*, 36, 4, (Dec. 2004), 372–421. doi: 10.1145/1041680.1041682.

[4] E. P. Duarte, L. C. E. Bona, and V. K. Ruoso. 2014. Vcube: a provably scalable distributed diagnosis algorithm. In *2014 5th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*. (Nov. 2014), 17–22. doi: 10.1109/ScalA.2014.14.

[5] Denis Jeanneau, Luiz A. Rodrigues, Luciana Arantes, and Elias Procópio Duarte Jr. 2017. An autonomic hierarchical reliable broadcast protocol for asynchronous distributed systems with failure detection. *J. Braz. Comput. Soc.*, 23, 1, 15:1–15:14. doi: 10.1186/s13173-017-0064-9.

[6] Leslie Lamport. 1998. The part-time parliament. *ACM Trans. Comput. Syst.*, 16, 2, (May 1998), 133–169. doi: 10.1145/279227.279229.

[7] S. -. Luan and V. D. Gligor. 1990. A fault-tolerant protocol for atomic broadcast. *IEEE Transactions on Parallel and Distributed Systems*, 1, 3, 271–285. doi: 10.1109/71.80156.

[8] Diego Ongaro and John Ousterhout. [n. d.] In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIXATC 14), pages=305–319, year=2014,*

[9] Alexey A. Paznikov, Andrey V. Gurin, and Mikhail S. Kupriyanov. 2020. Implementation in actor model of leaderless decentralized atomic broadcast. In *2020 9th Mediterranean Conference on Embedded Computing (MECO)*, 1–4. doi: 10.1109/MECO49872.2020.9134220.

[10] Marius Poke and Colin W. Glass. 2019. A dual digraph approach for leaderless atomic broadcast. In *2019 38th Symposium on Reliable Distributed Systems (SRDS)*, 203–20317. doi: 10.1109/SRDS47363.2019.00031.

[11] Marius Poke, Torsten Hoefler, and Colin W. Glass. 2017. Allconcur: leaderless concurrent atomic broadcast. In (HPDC '17). Association for Computing Machinery, Washington, DC, USA, 205–218. isbn: 9781450346993. doi: 10.1145/3078597.3078598.

[12] Luiz A. Rodrigues, Luciana Arantes, and Elias Procópio Duarte Jr. 2014. An autonomic implementation of reliable broadcast based on dynamic spanning trees. In *EDCC*. IEEE Computer Society, 1–12. doi: 10.1109/EDCC.2014.31.

[13] Luiz A. Rodrigues, Elias P. Duarte, and Luciana Arantes. 2014. Distributed and autonomic minimum spanning trees. In *2014 Brazilian Symposium on Computer Networks and Distributed Systems*, 138–146. doi: 10.1109/SBRC.2014.17.

[14] P. Thanisch. 2000. Atomic commit in concurrent computing. *IEEE Concurrency*, 8, 4, 34–41. doi: 10.1109/4434.895104.