Journal of Parallel and Distributed Computing 184 (2024) 104789

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

journal homepage: www.elsevier.com/locate/jpdc

Scalable atomic broadcast: A leaderless hierarchical algorithm

Lucas V. Ruchel [a], Edson Tavares de Camargo [b], Luiz Antonio Rodrigues [c,*], Rogério C. Turchetti [d], Luciana Arantes [e], Elias Procópio Duarte Jr. [f]

[a] Federal Institute of Parana (IFPR), Av. das pombas, 2020, Cascavel, 85814-800, PR, Brazil
[b] Federal Technology University of Parana (UTFPR), Rua Cristo Rei, 19, Toledo, 85902-490, PR, Brazil
[c] Western Parana State University (UNIOESTE), Rua Universitária, 2069, Cascavel, 85819-110, PR, Brazil
[d] Federal University of Santa Maria (UFSM), Av. Roraima, 1000, Santa Maria, 97.105-900, RS, Brazil
[e] Sorbonne Université/CNRS/Inria/LIP6, 4 place Jussieu, Paris, 75252, France
[f] Federal University of Parana (UFPR), Rua Evaristo F. Ferreira da Costa, 383-291, Curitiba, 82590-300, PR, Brazil

ARTICLE INFO

Keywords:
Distributed systems
Fault tolerance
vCube
Atomic broadcast
Dependability

ABSTRACT

Atomic Broadcast is an essential broadcast primitive as it ensures the consistency of distributed replicas. However, it is notoriously non-scalable. In this work, we introduce the Leaderless Hierarchical Atomic Broadcast (LHABcast) algorithm, which has two properties to improve scalability. First, it is a fully decentralized algorithm that does not rely on a sequencer/leader, which is often a significant bottleneck. Processes running LHABcast send messages with local sequence numbers and order messages received from other processes using timestamps inspired on Lamport's logical clocks. A process that receives the required set of timestamps can make a decision about the overall sequence of message delivery. Second, the algorithm is hierarchical: processes are organized on a vCube logical overlay network, which has several logarithmic properties and allows the construction of autonomous spanning trees. vCube also works as a failure detector, assuming crash faults and an asynchronous system model. In this paper, LHABcast is described, specified, and proven to be correct. Both simulation and experimental results are presented. A comparison with an all-to-all strategy shows that the number of messages sent by LHABcast is significantly lower in both fault-free and faulty scenarios. An implementation of LHABcast in Akka.io achieved up to 3.9 times higher throughput in fault-free scenarios than an implementation of the Raft-based Apache Ratis.

1. Introduction

Agreement is an essential problem of distributed systems that requires processes to reach any kind of consensus. 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 of messages in exactly the same order [8,4]. As a communication primitive, it is used to guarantee strong consistency while tolerating faults. Atomic broadcast algorithms typically rely on a leader [18], including the Paxos-based approaches [12] and also those that employ Raft [16]. 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 [4]. According to this approach, the message delivery order is the result of an agreement between the destination processes. There are three variants: i) agreement on a set of messages; ii) agreement on the adoption of a proposed message order; and iii) agreement on a sequence number. Both the first and second variants are implemented using a consensus algorithm, such as Paxos or Raft, and atomic confirmation protocols, such as those proposed in [14,26]. In general, they elect a leader who acts as a coordinator and is responsible for conducting rounds of consensus to determine 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 [20,17] and AllConcur+ [19] 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 its implementation, the third approach has the advantage that it allows all

* Corresponding author.
E-mail address: luiz.rodrigues@unioeste.br (L.A. Rodrigues).

https://doi.org/10.1016/j.jpdc.2023.104789
Received 5 February 2023; Received in revised form 25 July 2023; Accepted 14 October 2023
Available online 23 October 2023

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 (Leaderless Hierarchical Atomic Broadcast). All processes running LHABcast 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 counters maintained by each process. In addition, the trees are autonomically rebuilt when a process crashes, without the need for any additional messages. The trees are created and managed using the vCube virtual topology [6], described in Section 2. The vCube is a hypercube when all processes are fault-free, but it reconfigures itself when processes fail, maintaining several logarithmic properties. Besides connecting processes on a virtual scalable topology, vCube is also a failure detector, notifying correct processes as a failure is detected.

Although LHABcast is the first atomic broadcast algorithm defined for vCube, several other types of broadcast algorithms have been proposed for that virtual topology [23,10,24,3]. The first algorithm [23] defines both a best-effort and a reliable broadcast algorithm assuming a synchronous system model. Those algorithms dynamically build autonomic spanning trees that connect fault-free processes. The tree autonomically heals itself upon receiving notifications from the vCube failure detection service: Each process redefines its tree edges independently and seamlessly. In [10], the authors developed a reliable hierarchical broadcast algorithm assuming an asynchronous system. In order to ensure message delivery to processes that are falsely suspected of having crashed, the algorithm keeps sending messages to suspect processes in the tree.

We simulated both LHABcast and an alternative approach based on destination agreement, where each process sends a message directly to all other processes (as opposed to using a hierarchical strategy). The results show that LHABcast outperforms the all-to-all algorithm in terms of both latency and number of messages as the number of processes increases. Furthermore, we implemented LHABcast in Java using the *Akka* [13] framework. The implementation was compared to *Apache Ratis*, an atomic broadcast solution based on an optimized implementation of the *Raft* consensus algorithm in terms of throughput. Experimental results show that LHABcast achieves up to 3.9 times higher throughput than *Raft* in fault-free scenarios.

The remainder of this paper is organized as follows. The system model is defined in Section 2, including a brief description of the vCube overlay network and failure detector in an asynchronous system. Section 3 presents the proposed algorithm and proof of correctness. Section 4 presents the simulation and results of the experimental comparison of the proposed algorithm. Related work is discussed in Section 5. Finally, Section 6 concludes the paper and discusses future work.

2. System model

We assume a distributed system consisting of a finite set P of $n > 1$ processes that communicate by message passing. The network is fully connected: Each pair of processes can communicate with each other without using intermediaries. 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 *faulty*. The communication channels are reliable, i.e., messages exchanged between two correct processes are never lost, corrupted, or duplicated. The system is asynchronous, i.e., the relative processor speeds and delays of message transmissions are unbounded [2].

Processes are organized on a virtual hierarchical topology, the vCube [6], described in the next subsection.

2.1. An asynchronous vCube algorithm

The hypercube-like topology of the virtual Cube (vCube) was originally proposed in the context of distributed system-level diagnosis as the Hierarchical Adaptive Distributed System-level Diagnosis (Hi-ADSD) algorithm. The process can be in one of two states: correct or faulty. The crash fault model is adopted. The processes perform tests on each other and exchange test information to determine which processes are faulty/correct. The virtual topology created by the algorithm is hierarchical and corresponds to a hypercube when all processes are correct. However, it retains several logarithmic properties when processes crash and the correct processes reorganize. In particular, the latency to detect the crash of a process is a logarithmic function of the total number of system processes. Later, it was shown that the original algorithm leads to a quadratic number of tests in some specific cases. As a result, the vCube algorithm [6] was proposed, which ensures that the number of tests is a logarithmic function of the total number of processes.

Recently, a new system-level diagnosis model for the specification of unreliable failure detectors was proposed [5]. The major difference between the two approaches is that system-level diagnosis [15] is based on tests (which correspond to the push-based category of failure detectors) and, most importantly, the original diagnosis model assumes perfect tests and thus implicitly assumes the synchronous system model. On the other hand, unreliable failure detectors [2] were proposed for asynchronous systems and allow false suspicions to occur, i.e. a correct process can be classified as faulty. Although an earlier system-level diagnosis model had assumed tests that were not perfect [1], only in the recent work mentioned above [5] diagnosis algorithms are specified as failure detectors, including vCube.

vCube organizes processes into increasingly large clusters. Processes of a d-dimensional vCube have identifiers consisting of d bits. Two processes are virtually connected if their binary addresses differ by a single bit. The major advantage of the vCube is that the virtual topology reorganizes itself dynamically when processes crash, preserving several logarithmic properties, in particular the number of neighbors and the distance between two processes.

The virtual edges of a vCube correspond to the tests that correct processes perform on each other. vCube allows processes to obtain diagnostic information from each correctly tested process. The diagnostic information is timestamped to allow processes to distinguish recent from older events. Initially, each node is assumed to be correct and the corresponding timestamp is zero. After an event is detected, i.e., a correct node has become faulty or vice versa, the corresponding timestamp is incremented by one.

Algorithm 1 presents the pseudocode of the vCube failure detector adapted from [6] for use in an asynchronous system. Process i maintains timestamps for the state of every other process in the $STATE_i[]$ array. The algorithm provides that in the case of a false suspicion, i.e., if a process i was mistakenly suspected by a process j, the process i, upon receiving this information, stops running and exits the system (line 8). Furthermore, if a process suspects all others it also leaves the system. This approach is similar to the fail-aware failure detectors proposed by [7], where a correct process, once suspected by others, suspects itself. The result is a $\diamond P$ failure detector that is eventually strongly perfect [2] if correct processes remain in a single connected component, i.e., there are no partitions in the testing graph.

A tester process i executes tests on a cluster $c_{i,s}$ of processes of size 2^{s-1}, such that $s = 1 \dots \log_2 n$. The function $C_{i,s}$ (Equation (1)) returns the ordered list of processes of each cluster, where $\oplus$ is the bitwise exclusive or operator (*xor*).

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

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 $c_{i,s}$, $s = 1 \dots \log_2 n$. After a process detects that any other process has crashed, the set of edges (tests) is

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

Algorithm 1 Asynchronous vCube Failure Detector executed by process i.

```
1: $STATE_i[j] \leftarrow correct, \forall j = 0, .., n-1$
2: **repeat**
3:     **for** $s \leftarrow 1$ **to** $\log_2(n)$ **do**
4:         **for all** $j \in c_{i,s}$ | $i$ is the first correct process $\in c_{j,s}$ **do**
5:             TEST(j)
6:             **if** $j$ is tested $correct$ **then** $i$ obtains $STATE_j[]$ and $j$ obtains $STATE_i[]$
7:                 **if** $STATE_j[i] = suspect$ **then**     ▷ $i$ was suspected by $j$ (false suspicion)
8:                     Halt execution and leave the system
9:                 **else**
10:                    Update $STATE_i[]$ with the received $STATE_j[]$
11:            **else**
12:                **if** $STATE_i[j] = correct$ **then** $STATE_i[j] \leftarrow suspect$
13:                **if** $\forall j = 0, .., n-1, j \neq i$: $STATE_i[j] = suspect$ **then**
14:                    Halt execution and leave the system
15:    Sleep until the next testing interval
16: **until** forever
```

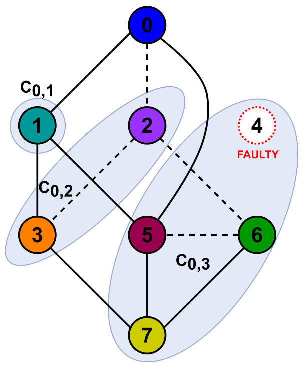


Fig. 1. Clusters of a three-dimensional vCube with $2^3 = 8$ processes; process 4 is faulty.

recomputed. For instance, in the example shown in Fig. 1, process p_4 originally tests process p_0, but after it has crashed, the tester of a process p_0 in the cluster with $s = 3$ is p_5.

The algorithm by Rodrigues and others [24,22] defines the construction of autonomic spanning trees for message dissemination in the vCube. Any vCube process can be the tree root. The following functions are defined in that work and used to specify the LHABcast algorithm:

- $cluster_i(j) = s$: Let i and j be two nodes of the system, $i \neq j$. Function $cluster_i(j) = s$ returns the index s of the cluster of node i that contains node j, $1 \leq s \leq \log_2 N$. For instance, in the 3-VCube shown in Fig. 1, $cluster_0(1) = 1$, $cluster_0(2) = cluster_0(3) = 2$ and $cluster_0(4) = cluster_0(5) = cluster_0(6) = cluster_0(7) = 3$. Note that for any i, j, $cluster_i(j) = cluster_j(i)$.
- $FF\_neighbor_i(s) = j$, the function returns the first node j in the cluster $c_{i,s}$ which is considered to be correct by i. If there is no such node, the function returns $\emptyset$ (empty, no neighbor). For example, in Fig. 1, $FF\_neighbor_0(1) = 1$, $FF\_neighbor_0(2) = 2$ and $FF\_neighbor_0(3) = 5$, since process 4 is faulty.
- $neighborhood_i(h)$ returns $\{\forall j \mid j = FF\_neighbor_i(s), 1 \leq s \leq h\}$, i.e., a set that contains all fault-free nodes virtually connected to a node i according to $FF\_neighbor_i(s)$, for $s = 1, .., h$. The parameter h can range from 1 to $\log_2 n$. If $\log_2 n$ is applied, the function returns all fault-free neighbors of node i in the hypercube. For any other value of $h < \log_2 n$, the function returns only a subset of the first fault-free neighbors that are connected to node i, i.e., those first fault-free neighbors whose respective cluster number $s <= h$. For example, considering the Fig. 1, $neighborhood_0(1) = \{1\}$, $neighborhood_0(2) = \{1, 2\}$ and $neighborhood_0(3) = \{1, 2, 5\}$. If node p_4 was considered correct by p_0, then $neighborhood_0(3) = \{1, 2, 4\}$; if p_1 is also detected as faulty by p_0, $neighborhood_0(1) = \emptyset$.

3. The LHABcast algorithm

Atomic Broadcast ensures reliable broadcast while ensuring 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 [4]:

- *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 broadcast algorithm that organizes processes using the vCube topology described in Section 2. vCube not only works as a failure detector, but also provides the means for message dissemination. When process i broadcasts a message m, it becomes the root of a spanning tree over which m is sent as well as the timestamps that are employed by the processes to reach total order. Those trees are built dynamically and autonomously within the vCube. Each correct process i determines its position in the tree and outgoing edges independently and autonomically, according to the topology rules. As soon as a process detects the failure of another process, the tree is reconfigured without the need for any additional messages. As the algorithm assumes the asynchronous system model, false suspicions can occur. However, they are equivalent to actual faults, as a suspected process eventually terminates its execution and leaves the system as soon as it discovers it has been suspected.

3.1. Algorithm description

The LHABcast algorithm works as follows. Let process i be the source, i.e., the process that executes the atomic broadcast primitive. Each process keeps a local sequential counter for the messages it broadcasts. After the source executes the atomic broadcast primitive, it updates the local sequence number and sends message m to its vCube neighbors, which are its children in the message dissemination tree. The source's set of neighbors consists of the first correct process of each of its vCube clusters. Upon receiving m, each process j, which is not a leaf, becomes itself the root of a subtree and in turn forwards m to its children. Those children are also the first correct processes of j's vCube clusters that are in that particular tree. When a leaf of the tree receives m, it sends an acknowledgment message (ACK) back to the

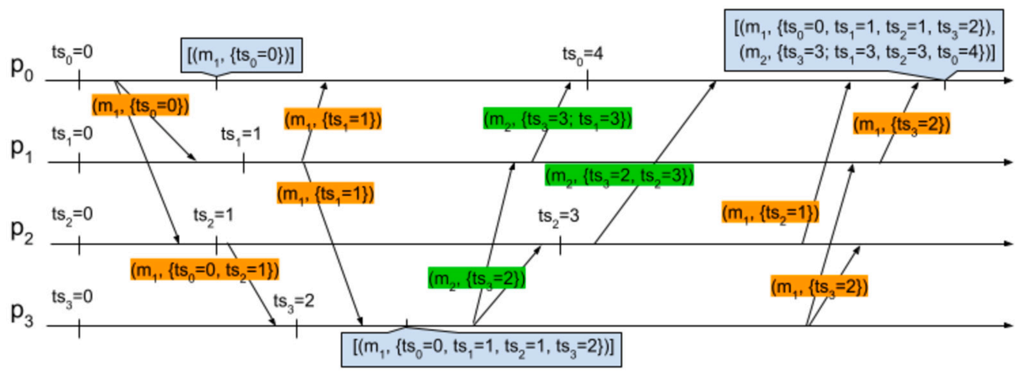


Fig. 2. Example of timestamp messages exchanged in a 2D-vCube.

process from which it had received m (its parent in the tree). 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 those clusters have surely received m too.

Besides the local sequence numbers, processes also maintain a timestamp that is assigned to each message. That timestamp was defined inspired on Lamport's logical clocks [11]. Initially, the timestamp is equal to the local sequence number. However, every time a process broadcasts or receives a new message, the timestamp is incremented to a value larger than both the current value and the timestamp of the received message. As a process receives message m for the first time, it updates and appends its timestamp for m before forwarding the message. The process then starts a spanning tree to forward the timestamp to all those processes which are not reached the broadcast message itself. All processes keep the complete set of timestamps received for each message until it is delivered. A message m is considered to be "deliverable" by process i after it has received timestamps assigned by all correct processes, and it has the lowest maximum timestamp of all messages waiting to be delivered.

In order to illustrate the use of timestamps to guarantee that messages are delivered in total order, consider the following example on a small (4 processes, 2 dimensional) vCube presented in Fig. 2.

Let the timestamps kept by the processes p_0, p_1, p_2 and p_4 be ts_0, ts_1, ts_2 and ts_3, respectively. Initially, all timestamps are set to zero. Consider that process 0 (p_0) broadcasts a message m_1 that carries $ts_0 = 0$. Initially, process 0 forwards the message to process 1 and process 2. Each of their timestamps is incremented to 1, i.e., a value greater than the current value (zero) and the maximum timestamp the message carries (zero). Process 1 is a leaf, but process 2 forwards the message to process 3 with timestamps $\{ts_0 = 0, ts_2 = 1\}$, which makes $ts_3 = 2$. All processes also send their timestamps to all others. However, consider that before any process receives all timestamps and can deliver m_1, they all receive another message m_2 broadcast by process 3 with timestamp $ts_3 = 2$. Process 3 forwards m_2 to processes 2 and 1, both of which increment their timestamps: $ts_2 = 3$ and $ts_1 = 3$. Process 2 in turn sends m_2 to process 0, which increments $ts_0 = 4$. Now consider that all processes have received timestamps of all processes for both messages. The highest timestamp for m_1 is $ts_3 = 2$, while the highest timestamp for m_2 is $ts_0 = 4$. Thus, all processes deliver first m_1 and then deliver m_2.

Now consider a case in which both messages are sent simultaneously, and reach different processes in different orders. Either one of the messages will be assigned a greater timestamp than the other (establishing the delivery order) or both messages end up with the same value for the largest timestamp. In case of a tie, the process id is used to determine the order, i.e. the messages are ordered according to the corresponding process identifier order, from lowest to largest. Note that if the two messages have exactly the same value for their corresponding largest timestamps, that value must have been assigned by different processes, as a single process necessarily assigns different values for the timestamps of different messages. As a final remark, note that in addition to the timestamps, each message also carries a sequence number generated by the broadcast source for the purpose of providing (together with the source id) a unique identifier for the message.

Before the algorithm is presented in pseudocode, we first describe the local variables maintained by each process and the two message types used. Each process i that executes the algorithm keeps the following local variables:

- $Correct_i$: the set of processes that are considered to be correct by process i according to information provided by the vCube failure detector;
- seq_i: local message counter incremented sequentially, which together with the source id (i.e., i) is used to identify each message uniquely. The message field $m.seq$ is set with the current value of seq_i and is used by the receivers to control which messages have already been delivered;
- $timestamp_i$: *timestamp* of the last message sent/received by i, so that each message transmitted by the process has a different timestamp. It is used to establish the total order after a process receives all timestamps for a message m;
- $lastMsg_i[n]$: the last message delivered by i from each source process, respecting the FIFO order;
- $PendingAcks_i$: the set of pending ACKs. For each message $TREE\langle m, T\rangle$ broadcast by i or received from a parent process j and retransmitted to a child process k, an element $\langle parent, child, \langle m, Timestamps_i^m\rangle\rangle$ is added to the $PendingAcks_i$ set; $Timestamps_i^m$ is the set of *timestamps* currently assigned to m;
- $ReceivedMsgs_i$: the set of messages received by process i that cannot yet be delivered to the application. Each element in this set contains $\langle m, ts_k^m\rangle$, i.e., the message m to be ordered plus the timestamps ts_k^m assigned to m by each process k;
- $TimestampedMsgs_i$: the set of messages m for which all timestamps ts_k^m have been received, but which still cannot be delivered because they are out of order with respect to the other messages $m' \neq m$.

The LHABcast algorithm employs two types of messages:

- $TREE\langle m, Timestamps_i^m\rangle$, this type is the used to broadcast a message m with the set $Timestamps_i^m$ of assigned timestamps ts_k^m. $Timestamps_i^m$ has from 1 to $log_2 n$ elements (the longest path in the tree).
- $ACK\langle m, Timestamps_i^m\rangle$, this message type is used to send acknowledgments that confirm the receipt of a $TREE$ message.

Each message m is broadcast with two fields that together work as a unique identifier of the message: (i) the source process identifier $m.src$; and (ii) the local sequence number $m.seq$ established by the source.

Algorithm 2 presents the proposed solution in pseudo-code. In order to atomically broadcast a message m, the source process, say i, executes the A-BROADCAST(m) primitive (line 9), which sets i as the $m.src$ and adds the local message counter seq_i to the message. The FORWARD function is used to send $TREE$ messages to the vCube neighbors of a process i. The parameters of this function are the *clusters* index s of i, which allows process i to locate itself along the dissemination tree, the message m and the timestamp. Initially, $s = \log_2 n$ which is the dimension of the vCube and corresponds to the total number of clusters of the system. In the FORWARD function, message m is forwarded only to the first fault-free neighbor of i in each cluster indexed by s (line 17).

Algorithm 2 LHABcast Atomic Broadcast at process i.

```
 1: procedure INITIALIZATION( )
 2:   $Correct_i \leftarrow \{0, .., n-1\}$        ▷ set of processes considered correct by $i$
 3:   $seq_i \leftarrow 0$        ▷ local message counter for messages sent by process $i$
 4:   $timestamp_i \leftarrow 0$   ▷ local timestamp of the last message sent/received by $i$
 5:   $\forall j \in Correct_i : lastMsg_i[j] \leftarrow \perp$    ▷ array with the last message delivered
      by $i$ from each source process $j$
 6:   $PendingAcks_i \leftarrow \emptyset$ ▷ set of processes from which an acknowledgment has
      not been received
 7:   $ReceivedMsgs_i \leftarrow \emptyset$   ▷ set of all messages received but not yet delivered
 8:   $TimestampedMsgs_i \leftarrow \emptyset$ ▷ set of messages for which the timestamps have
      been received from all correct processes, but have not yet been delivered

 9: procedure A-BROADCAST($m$)
10:   $m.src \leftarrow i$
11:   $m.seq \leftarrow seq_i$
12:   $seq_i \leftarrow seq_i + 1$
13:   $ReceivedMsgs_i \leftarrow ReceivedMsgs_i \cup \{\langle m, timestamp_i \rangle\}$
                  ▷ Forward $m$ to all neighbors of $i$ in each cluster $s = 1..\log_2 n$
14:   FORWARD($i$, $\log_2 n$, $TREE\langle m, \{Timestamps_i^m = \{timestamp_i\}\}\rangle$)
15:   $timestamp_i \leftarrow$ MAX($timestamp_i$, $seq_i$)

16: procedure FORWARD($parent$, $s$, $TREE\langle m, Timestamps_i^m \rangle$)
                  ▷ Forward message $m$ to all neighbors in clusters $1..s$ of $i$
17:   for all $childprocess \in vCubeFD.neighborhood_i(s)$ do
18:     SEND($TREE\langle m, Timestamps_i^m \rangle$) to $child$
19:     $PendingAcks_i \leftarrow PendingAcks_i \cup \{\langle parent, child, \langle m, Timestamps_i^m \rangle\rangle\}$

20: upon receive $TREE\langle m, Timestamps_j^m \rangle$ from process $j$
21:   if $j \notin Correct_i$ then
22:     RETURN
23:   $timestamp_i \leftarrow$ MAX($\forall ts_k^m \in Timestamps_j^m, timestamp_i + 1$)
                  ▷ Next, check if $m$ is a new message, i.e never received before
24:   if ($lastMsg_i[m.src] = \perp$ or $m.seq > lastMsg_i[m.src]$)
          and $m \notin \{ReceivedMsgs_i \cup TimestampedMsgs_i\}$ then
25:     $Timestamps_i^m \leftarrow \{timestamp_i\}$
                  ▷ Next, forward $m$ to all neighbors of $i$ except those overlapping the
      subtree of $j$
26:     for all $p \in vCube.neighborhood_i(\log_2 n)$
            $\setminus vCube.neighborhood_i(vCube.cluster_i(j) - 1)$ do
27:       SEND($TREE\langle m, \{ts_i^m\}\rangle$) to $p$
28:       $PendingAcks_i \leftarrow PendingAcks_i \cup \{\langle i, p, \langle m, timestamp_i \rangle\rangle\}$
29:   $Timestamps_i^m \leftarrow Timestamps_i^m \cup Timestamps_j^m$
30:   for all $ts_k^m \in Timestamps_i^m$ do
31:     $ReceivedMsgs_i \leftarrow ReceivedMsgs_i \cup \{\langle m, ts_k^m \rangle\}$
      ▷ Next, forward the bundled timestamps in the overlapping subtrees of $j$
      and $i$
32:   FORWARD($j$, $vCube.cluster_i(j) - 1$, $TREE\langle m, Timestamps_i^m \rangle$)
33:   CHECKDELIVERABLE($m$)
34:   CHECKACKS($j$, $\langle m, Timestamps_i^m \rangle$)

35: upon receive $ACK\langle m, Timestamps_i^m \rangle$ from process $j$
36:   $p \leftarrow x{:}\langle x, j, \langle m, Timestamps_i^m \rangle\rangle \in PendingAcks_i$
37:   $PendingAcks_i \leftarrow PendingAcks_i \setminus \langle p, j, \langle m, Timestamps_i^m \rangle\rangle$
38:   CHECKDELIVERABLE($m$)
39:   CHECKACKS($p$, $\langle m, Timestamps_i^m \rangle$)

40: procedure CHECKDELIVERABLE($m$)
41:   if received $\langle m, ts_k^m \rangle$ from all $k \in Correct_i$ and $PendingAcks_i \cap \langle *, *, \langle m, * \rangle\rangle$
      $= \emptyset$ then
42:     $max\_timestamp_i^m \leftarrow$ MAX($ts_k^m \mid \langle m, ts_k^m \rangle \in ReceivedMsgs_i$)
43:     LHABDELIVER($m$, $max\_timestamp_i^m$)
44:     for all $\langle m' = m, ts_k^m \rangle \in ReceivedMsgs_i$ do
45:       if $lastMsg_i[m.src] = \perp$ or $m'.seq = lastMsg_i[m.src] + 1$ then
46:         $lastMsg_i[m.src] \leftarrow m'$
47:       $ReceivedMsgs_i \leftarrow ReceivedMsgs_i \setminus \{\langle m', * \rangle\}$

48: procedure LHABDELIVER($m$, $max\_timestamp_i^m$)
49:   $TimestampedMsgs_i \leftarrow TimestampedMsgs_i \cup \{\langle m, max\_timestamp_i^m \rangle\}$
50:   $DeliverableMsgs_i \leftarrow \emptyset$
51:   for all $\langle m', max\_timestamp_i^{m'} \rangle \in TimestampedMsgs_i$
          $\mid \forall \langle m'', ts_k^{m''} \rangle \in ReceivedMsgs_i : max\_timestamp_i^{m'} < ts_k^{m''}$ do
52:     $DeliverableMsgs_i \leftarrow DeliverableMsgs_i \cup \{\langle m', max\_timestamp_i^{m'} \rangle\}$
53:   Deliver all messages in $DeliverableMsgs_i$ in order ($max\_timestamp_i^m, m.src$)
54:   $TimestampedMsgs_i \leftarrow TimestampedMsgs_i \setminus DeliverableMsgs_i$

55: procedure CHECKACKS($parent$, $\langle m, Timestamps_i^m \rangle$)
56:   if $PendingAcks_i \cap \langle parent, *, \langle m, Timestamps_i^m \rangle\rangle = \emptyset$ then
57:     if $m.src \neq i$ and $\{parent\} \in Correct_i$ then
58:       SEND($ACK\langle m, Timestamps_i^m \rangle$) to $parent$

59: upon notifying crash(process $j$)
60:   $Correct_i \leftarrow Correct_i \setminus \{j\}$
61:   for all $parent = p{:}\langle p, j, \langle m, Timestamps_i^m \rangle\rangle \in PendingAcks_i$ do
62:     if $\{parent\} \in Correct_i$ then
        ▷ Next, check if there is a correct neighbor ($child$) in the same cluster of $j$
63:       if $child = vCube.FF\_neighbor_i(cluster_i(j)) \neq \emptyset$ then
64:         SEND($TREE\langle m, Timestamps_i^m \rangle$) to $child$
65:         $PendingAcks_i \leftarrow PendingAcks_i \cup \langle parent, child, \langle m, Timestamps_i^m \rangle\rangle$
66:     $PendingAcks_i \leftarrow PendingAcks_i \setminus \langle parent, j, \langle m, Timestamps_i^m \rangle\rangle$
67:     CHECKACKS($parent$, $\langle m, Timestamps_i^m \rangle$)
68:   for all $\langle m, * \rangle \in ReceivedMsgs_i$ do
69:     CHECKDELIVERABLE($m$)
```

Upon receiving a $TREE$ message from process j, process i checks whether j is in the set of correct processes (line 21). If it is not, the message is ignored. Otherwise, i updates its local timestamp $timestamp_i$ based on the newly received timestamp (line 23). Then i checks whether it is receiving the message for the first time, i.e., it is not yet in any of the delivered, received or timestamped sets of messages. In this case (the first time the message is received) then i forwards the message with the updated $timestamp_i$ along the tree, according to its tree location determined with the s index. Note that when m is first received by process i, $timestamp_i$ is aggregated to $Timestamps_i^m$ (line 29). Message m is forwarded with the updated set of timestamps $Timestamps_i^m$ (line 32). Furthermore, process i will start a new tree to send its timestamp for the message to the process that the message dissemination tree does not reach (line 26).

Fig. 3 shows an example execution of LHABcast on a three-dimensional vCube. Process p_0 is the source, the figure shows how message m is forward along its spanning tree. The dark blue arrows show p_0 forwarding message m to its neighbors in clusters $s = 1, 2, 3$. Fig. 3a shows process p_1 as it receives message m for the first time. p_1 determines through cluster index s $\{cluster_1(0) - 1 = 0\}$ that it is in the last level of the p_0 tree (it is a leaf). Therefore, p_1 starts another tree to send its timestamp to the processes $\{0, 3, 5\}$ (represented in the arrows by ts_1). Fig. 3b shows the algorithm executed by process p_2, which receives m from p_0, computes its location in the tree with cluster index s ($cluster_2(0) - 1 = 1$), and forwards m with its timestamp to processes $\{0, 6\}$. In p_0's 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 the timestamps contained in $Timestamps_i^m$ to the list of received timestamps of m (line 31) and calls function CHECKDELIVERABLE(m). This function checks if the timestamps of all correct processes for m (line 41) have been received. If it is the case, i computes the highest timestamp of m (line 42), denoted $max\_timestamp_i^m$, and function LHABDELIVER($m, max\_timestamp_i^m$) is called (line 43). Then, i updates the register

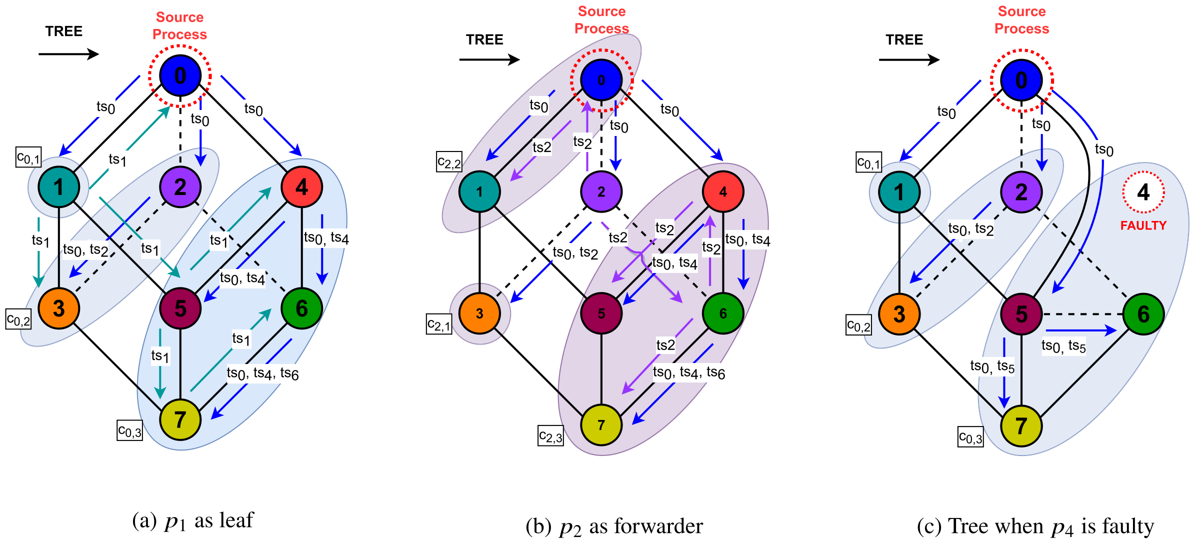


Fig. 3. Examples of LHABcast execution with 8 processes.

of delivered messages $lastMsg_i$ and all timestamps associated with m are removed from the set $ReceivedMsgs_i$.

In the LHABDELIVER function, message m is added to the set of timestamped messages ($TimestampedMsgs_i$) with the highest received timestamp – $max\_timestamp_i^m$ (line 49). On the other hand, a message m' in $TimestampedMsgs_i$ is added to the deliverable set ($DeliverableMsgs_i$) only if there is no received message m'' in $ReceivedMsgs_i$ whose timestamp is lower than m' (line 51). Otherwise, m' should be delivered after m'' (line 51). Messages in $DeliverableMsgs_i$ are ordered by the timestamp associated with m and the identifier of the source process of m ($m.src$) (line 54). In this way, messages are delivered in order and then removed from $TimestampedMsgs_i$.

Confirmation of receipt of the message is performed by the CHECKACKS function. The tuple $\langle parent, *, \langle m, Timestamped_i^m\rangle\rangle$ represents which ACKs are being waited for message m, given the timestamp received from the $parent$ process and forwarded to any process ($*$) (line 56). For example, in Fig. 3a, 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 holds for 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 notifies p_0, which originally sent the message with $Timestamped_i^m$, with an ACK.

The vCube failure detector notifies processes about failures. For all processes, when i receives a notification that j crashed (line 59), j is removed from the list of correct processes. Thus, for each message m forwarded to j not confirmed, i.e., there is a message m sent to j in $PendingAcks_i$ (line 61), this message m must be forwarded to the next correct process in the cluster to which j belongs, if there is one (the $FF\_neighbor_i(s)$ function determines this information, where $s = cluster_i(j)$) (line 63). For example, consider the three-dimensional hypercube in Fig. 3c 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*, in this case, 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 p, the process that sent m to i (line 67). Furthermore, 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 message m in $ReceivedMsgs_i$, function CHECKDELIVERABLE (line 69) 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 prove that Algorithm 2 ensures the validity, integrity, agreement, and total order properties of atomic broadcast.

Lemma 3.1 *(Validity). Algorithm 2 ensures that if a correct process* A-BROADCASTS *a message m, then it eventually delivers m.*

Proof. When process i broadcasts a message m, it includes $\{\langle m, timestamp_i\rangle\}$ into the set $ReceivedMsgs_i$ (line 13) and forwards m to all correct vCube neighbors (line 17). For each process j to which i sent m, i added a pending ack in $PendingAcks_i$. If j is correct, it responds with an ACK message (CHECKACKS procedure) and i removes the pending ack in $PendingAcks_i$ on line 37. If j is faulty, i will eventually detect the crash and remove the pending ack in line 37.

As a result, all outstanding acks for m will eventually be removed from $PendingAcks_i$ and, after receiving the timestamps to m from all correct processes (line 41), i will deliver m on line 43. □

Lemma 3.2 *(Integrity). Algorithm 2 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 $ReceivedMsgs_i$ set (line 43). Messages broadcast by the sender are added to $ReceivedMsgs_i$ in line 13. Messages received from other processes are included into the $ReceivedMsgs_i$ in line 31. 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:

- $\mathbf{source(m) = i}$. Process i called A-BROADCAST broadcasting m. As proved in Lemma 3.1, i will deliver m on line 43. 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 43. Since $lastMsg_i[i]$ was set to m in line 46 after the delivery on m, it follows that m will never be qualified to pass the test again.
- $\mathbf{source(m) \neq i}$. 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 43. Before i delivers m for the first

time, it sets $lastMsg_i[m.src]$ to m in line 46. Thus, m will never pass the test again, and i may therefore deliver m at most once. □

Lemma 3.3 *(Agreement). Algorithm 2 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 17. If j is correct, it will eventually receive m, since the links are reliable, and will deliver m on line 43. 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 $|P_1| = |P_2| = 2^k$.
  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 63, 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 the local timestamp to all correct processes on lines 26 and 32. Since j is correct, each correct process will eventually receive m. □

Lemma 3.4 *(Total Order). Algorithm 2 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 $ts_a^m < ts_b^{m'}$. By agreement, both messages and their respective timestamps are included in $ReceivedMsgs_i$ by all correct processes. Finally, both messages are delivered in the same order in line 54.
- $m.src! = m'.src$. 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 54. □

Theorem 3.5. *Algorithm 2 implements an atomic broadcast.*

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

4. Evaluation

This section presents the results of both a simulation of LHABcast done with Neko [27] and a Java implementation based on the Akka framework [13]. In the simulation, LHABcast was compared to an all-to-all strategy (called All2All), in which instead of employing the vCube to communicate, all processes send all messages to all processes. The Java Akka implementation of LHABcast was compared to a highly customizable implementation of the *Raft* protocol, called Apache Ratis [21].

4.1. Simulation

In the All2All strategy, each process communicates directly with the other processes. Upon receiving a message, a process sends the timestamp (ts) of the message to $n - 1$ processes. Different scenarios and system sizes were considered in the simulation. In each experiment, the number of processes n varied from 8 to 1024. The simulation was run on an AMD A8-5500B processor with 16 GB RAM. In the All2All strategy, the failure detector communicates directly with the processes and therefore has lower latency compared to LHABcast which relies on vCube to propagate messages, including failures, providing greater scalability but increasing latency for failure detection.

For the number of messages, we considered all messages exchanged between processes, including delivery acknowledgments (*ACKs*) while latency is defined as the unit of time t that a message takes to be delivered to all non-faulty processes.

In the experiments, when a message is broadcast, the time to send it to a given process takes t_s. Thus, if the message is broadcast to more than one process, t_s must be considered for each copy of the message sent. A process takes t_r time units to receive a message. The network transmission time of a message is given by t_t. Hence, as $t = t_r + t_s + t_t$, we have defined $t_r = t_s = 0.1$ and $t_t = 0.8$ time units for all executions. The test interval for failure detection was set to 30.0 time units in each experiment. A process is considered to have crashed if the failure detector does not receive a response within 4 * $(t_s + t_t + t_s)$[1] time units. Next, the results of the experiments conducted in scenarios with and without process failures are presented.

4.1.1. Simulation scenario in which all processes are correct

Fig. 4 shows the number of messages and the latency when process p_0 broadcasts a message m. In the hierarchical strategy, the longest path of a vCube consists of subtrees of size $\log_2 n$. Therefore, in the worst case, each copy of m and its timestamp is forwarded $2log_2 n$ times. Remember that p_0 must receive *ACKs* from all other processes to be sure that m and its timestamp were received by them. Hence, the time p_0 takes to deliver m is the time taken by the process in the longest subtree of p_0 to forward m and its timestamp. In the All2All strategy, latency is calculated based on the time a process takes to forward its timestamp to the other processes after receiving a message. Therefore, the latency is higher for the hierarchical strategy for systems with up to 128 processes. However, the situation reverses when the number of processes exceeds the threshold of 128. The hierarchical strategy actually exhibits lower latency (see Fig. 4b) because it takes longer for the single sender p_0 to send copies of m to all other processes. The larger the number of processes, the longer it takes p_0 to send individual messages to each of them. The latency to forward a message with the hierarchical strategy (vCube) grows roughly linearly with the number of neighbors of each process, while the All2All strategy shows an exponential behavior.

[1] This value is twice the time from sending to receiving a message.

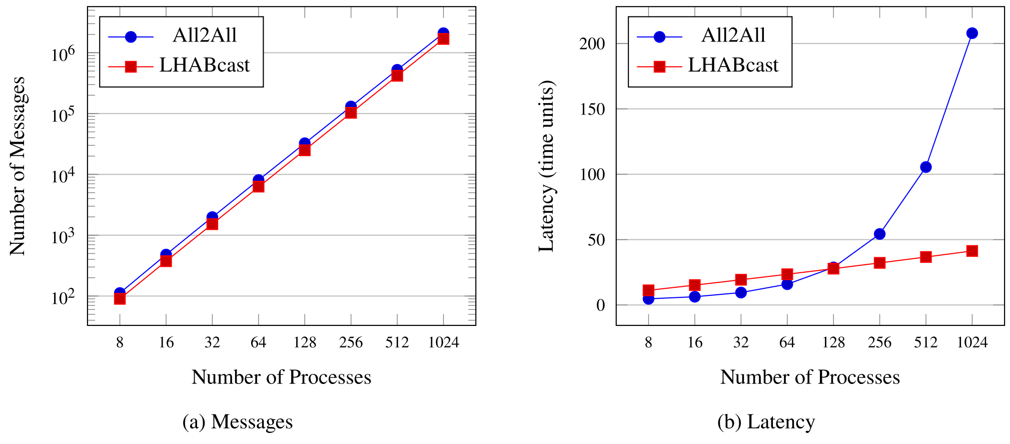


Fig. 4. Atomic broadcast executed in fault-free scenarios.

In order to ensure the total order delivery of all messages, all processes must receive the timestamp of each message m. When using the All2All strategy, all processes send their timestamps directly to the other processes. On the other hand, the hierarchical strategy aggregates multiple timestamps into a single message. Thus, the number of messages in LHABcast is on average 21.45% lower than in All2All (Fig. 4a).

4.1.2. *Simulation scenario with faulty processes*

The latency and number of messages from three different scenarios with faulty processes are presented next. In the first scenario the source process fails. The second scenario shows the impact of a failure of any system process before that process is detected as faulty. Finally, in the third scenario, the algorithms are executed after the detection of a faulty process by all other processes in the system.

The source fails. We assume that at least one fault-free process received the message before the source fails. This assumption is consistent with the termination property (*liveness*) inherited from the reliable broadcast. According to the LHABcast algorithm, if a process fails and this is detected, all processes simply ignore the timestamps from that process, even if the process is the source of the message. Furthermore, since the algorithm uses hierarchical message broadcasting, processes that do not receive acknowledgments for a message sent to a faulty process must forward the same message to the next non-faulty process in the cluster of that faulty process, if there is one. To perform the experiment, we assumed that p_0 broadcasts a message at time 0 and fails shortly thereafter.

Fig. 5a shows a small fluctuation in the number of messages because a failed process does not receive the timestamps. Consequently, the number of messages of LHABcast is 21.74% lower in comparison with All2All, a similar result to that of the fault-free scenario. However, the message delivery latency as shown in Fig. 5b is significantly different when there are process failures. vCube's failure detector runs in rounds. For all non-faulty processes to detect a failure, more than one round of the algorithm must be executed, which directly affects the latency of message delivery. The difference of the message delivery latency without failures and in the presence of a failure is of 29.63 time units in this experiment. The All2All approach exhibited higher latency when the total number of processes grew above 512, as the times to send and receive messages (t_s and t_r) increase significantly with a larger number of processes.

Sending a message before the failure of the receiver is detected. We consider that process p_1 fails during the broadcast of message m by p_0. A process that forwards m to a failed process will never receive an acknowledgment of receipt from that process and will eventually detect it as faulty. Note that the failure of any process implies the removal of the timestamp of that process from the set of received messages, and also on the transmission of *ACKs*, if any. Therefore, the latency and the number of messages in these scenarios are identical to the scenario in which the source fails.

Sending a message after the failure of the receiver is detected. In this experiment, process p_1 fails at time 0. In the simulated scenario, source process p_0 only starts the broadcast after all processes have detected that p_1 is faulty. Thus, all processes running the algorithm reorganize themselves in the dissemination tree excluding faulty process p_1 from the topology. The simulation results in Fig. 6 show that for the All2All approach, the number of messages is lower in comparison with the fault-free scenario, since there are fewer processes. On the other hand, LHABcast presents fewer messages than All2All (Fig. 6a).

4.2. *Implementation*

We implemented the LHABcast algorithm in Java using the Akka framework [13]. Akka automatically handles concurrency issues, since each component of the system is an actor. Each actor has a *mailbox* that is used to receive messages from other processes and each message received is retrieved and processed sequentially by the *mailbox* handler. Our implementation uses the default failure detector of the framework, which is the *phi accrual* detector [9].

The experiment was executed on a cluster consisting of 16 physical machines connected to an Ethernet gigabit network. Each machine was based on an AMD A8-5500B processor, each supporting 4 threads and equipped with 16 GB of RAM. Half of the machines (eight) were used to run either LHABcast or Apache Ratis, while the others were configured as clients that broadcast messages.

4.2.1. *Experiments executed in a fault-free scenario*

In this experiment, each client broadcasts a new message after receiving confirmation from the previous one. The number of clients ranged from 8 to 512, with each host running from 1 to 64 clients.

LHABcast significantly outperforms Apache Ratis in a fault-free scenario with up to 256 clients (Fig. 7a), showing a throughput of 390.01%, 301.72%, 174.38%, 119.78%, 55.45%, and 0.66% for 8, 16, 32, 64, 128, and 256 clients, respectively. However, for 512 clients and above, LHABcast performance is 53.45% lower compared to Apache Ratis.

The throughput of LHABcast decreases due to both the number of messages received by each client and the number of messages processed by the algorithms. As the number of clients increases, more messages are received simultaneously by each client as each of them receives all the messages in total order, which affects client performance. Moreover, the leaderless approach induces an additional cost for ordering messages and, in LHABcast, these costs are related to the number of

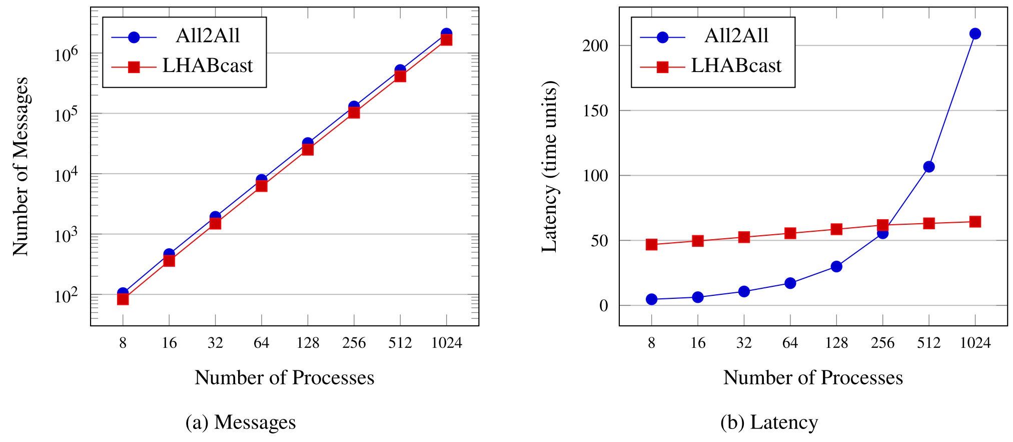


Fig. 5. Scenario in which the source fails.

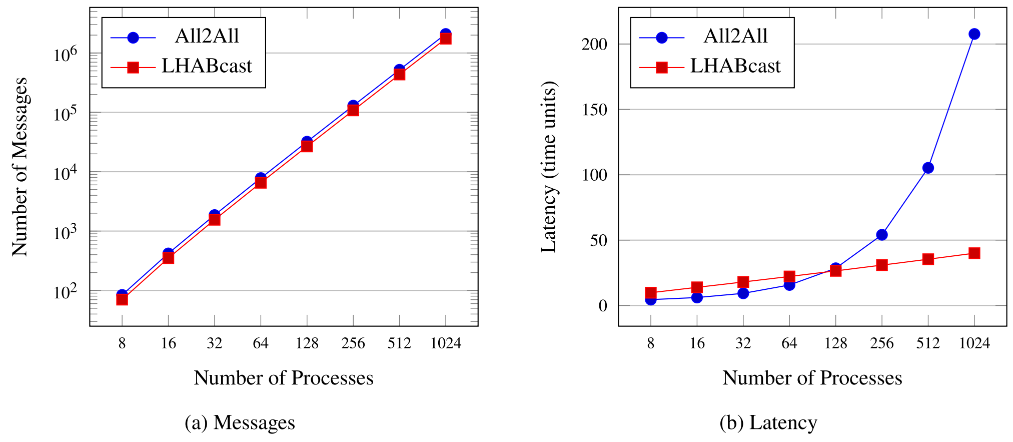


Fig. 6. Sending a message after the failure of the receiver is detected.

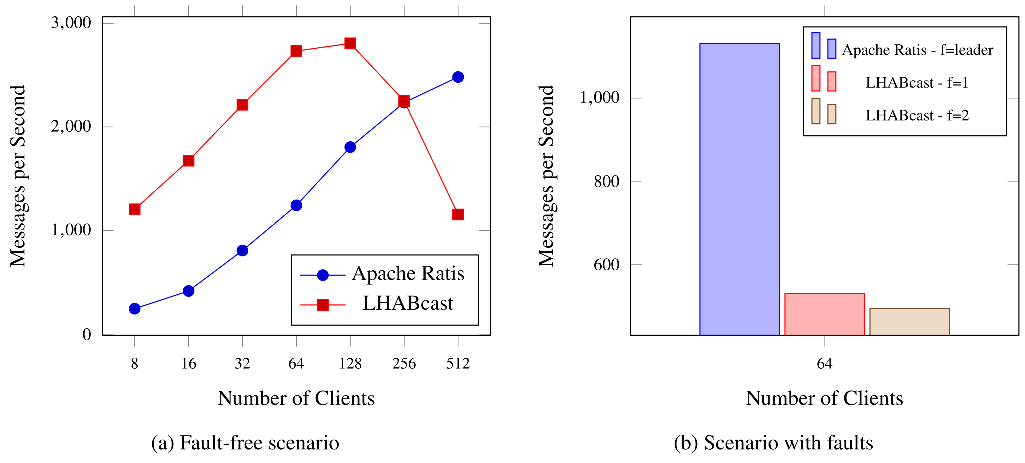


Fig. 7. Results of LHABcast and Apache Ratis implemented with *Akka.io*.

messages exchanged by the algorithm. As the number of clients increases, more messages are sent, and therefore LHABcast exchanges more messages among processes than Apache Ratis.

4.2.2. Experiments with faulty processes

We examined the impact of process faults on client requests in both Apache Ratis and LHABcast. The number of clients was set to 64, i.e. 8 machines with 8 clients each while the other 8 machines are dedicated to running the algorithms. This configuration was chosen because it presented one of the highest throughputs in the fault-free scenario, as shown above.

Fig. 7b shows the results obtained when running LHABcast with the failure of one and two processes. In the case of Apache Ratis, the leader fails. It is important to remember that processes in *Raft* can take one of three roles: *Follower*, *Leader*, and *Candidate*. The leader is responsible for receiving requests and coordinating consensus. If the leader fails, a new leader is elected.

In Fig. 7b, it can be seen that Apache Ratis presents little variation in the message delivery rate despite the leader failure, while LHABcast reaches 53.18% and 56.44% lower throughput with 1 and 2 failed processes, respectively. While *Raft* shows a decrease in the delivery rate of 8.97% compared to error-free execution, this drop is 80.61% and 81.96% when one and two processes failed, respectively.

For LHABcast, the decrease of the message delivery rate is due to the time it takes for the all processes to detect failures, which is affected by the values of the failure detector parameters set in the Akka framework. The latter defines how conservative the failure detector is. The more conservative it is, the lower the rate of false suspicions. However, despite using the recommended values for local networks, the performance levels observed cannot be considered satisfactory. In Apache Ratis, failure detection is performed by the algorithm itself. Thus, processes that detect the failure of the leader initiate the election of a new leader, and new messages can be processed by the newly elected leader.

5. Related work

AllConcur is a leaderless atomic broadcast algorithm that uses a digraph as an overlay network to disseminate messages [20]. Similar to the preliminary version of LHABCast [25], AllConcur assumes a perfect failure detector (P), which implies on a synchronous system model. Both algorithms (AllConcur and LHABcast) allow concurrent message dissemination by all processes. However, the initialization of AllConcur requires a reliable centralized service such as Zookeeper [28], which allows servers to agree on the initial configuration (i.e., the identity of participating servers and the digraph). In contrast, LHABCast uses vCube as a failure detector and dynamically reconfigures itself at runtime in the presence of failures.

AllConcur uses a technique called *early termination* to reduce message delivery latency. This technique requires each server to keep track of all messages in the system. In each round, each correct process broadcasts message m, registers the messages of the round using the *early termination* mechanism, and can deliver the messages received in the round in a deterministic order after message tracking is complete. When a server fails, its successors detect the failure and disseminate failure notifications to the other servers. The algorithm was implemented and compared with Libpaxos, an implementation of the Paxos consensus algorithm.

In the paper, the authors discuss the implications of relaxing the accuracy property of the detector, which implies on having to deal with false suspicions. In this case, the authors point to the use of additional messages that take advantage of the proposed topology to ensure that messages are received by all processes. However, there is still the problem of one process being isolated from the others, because such a constraint is imposed that only a majority *quorum* can deliver messages if some of the processes are isolated.

Later, an implementation of AllConcur was presented [17], based on the Java Akka framework. The authors showed that even when using a $\diamond\mathcal{P}$ detector (the default setting of the framework), the properties of the algorithm are preserved. Although this implementation has been tested for performance in shared and distributed memory environments, it is not compared to other consensus algorithms.

AllConcur+ [19] assumes a failure detector $\diamond\mathcal{P}$. In that algorithm, two overlapping networks are used for communication between processes. The authors assume that failures rarely occur in real environments, so an unreliable overlay network can be used to achieve higher performance. When failures are detected, the algorithm takes a more conservative stance and uses a reliable overlay network that ensures a vertex-connectivity larger than the maximum number of tolerated failures. The algorithm was evaluated in a simulated environment and was able to achieve up to 6.5x higher throughput and up to 3.5x lower latency than AllConcur and up to 318x higher throughput and 158x lower latency than Libpaxos.

6. Conclusion

In this work, we presented the LHABcast distributed algorithm, a hierarchical, autonomous and fully decentralized atomic broadcast algorithm. LHABcast employs a scalable strategy to allow processes to deliver messages in total order without the need for a leader. LHABcast is built on top of vCube, a scalable virtual topology that also works as a failure detector. In case of process failures, LHABcast relies on the properties of vCube so that fault-free processes autonomically reorganize themselves in a new configuration that preserves multiple logarithmic properties. LHABcast was described, specified and proven to guarantee the properties required of any atomic broadcast algorithm.

LHABcast was implemented and evaluated, both by simulation and through a Java implementation based on the Akka framework. We simulated scenarios both with and without process failures. LHABcast was compared to an all-to-all strategy (called All2All) where processes communicate directly. In scenarios without failures, the number of messages of LHABcast was significantly lower than that of the All2All strategy. As the number of processes grew above 128, the latency of the All2All strategy surpassed that of LHABcast. In scenarios with failures, there were no significant differences in the number of messages of both strategies. We also observe that the execution interval of the failure detector rounds has an impact on the latency. Nevertheless, LHABcast presented a lower latency as the number of processes increases and outperforms the All2All approach for more than 256 processes, confirming that the proposed solution tends to be more scalable.

The experimental evaluation conducted with the Java Akka implementation was also done on scenarios with and without failures. The comparison, in this case, was with Apache Ratis, an optimized implementation of Raft. Without failures, LHABcast showed significant performance gains over Apache Ratis. However, with more than 256 processes, the number of messages processed by LHABcast decreases compared to Apache Ratis. As a fault occurs as a message broadcast is still going on, the performance of LHABcast is affected by the failure detection latency, and there is a performance drop compared to Apache Ratis.

In the future, we will investigate optimizations aimed at further reducing the number of messages required to establish the total order of distributed application messages, thereby improving performance. We also intend to compare LHABcast with other solutions such as AllConcur and AllConcur+. Other research directions include evaluating the impact of changing the LHABcast in the crash-recovery model.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was partially supported by the Brazilian Research Council (CNPq – Conselho Nacional de Desenvolvimento Científico e Tecnológico) grant 308959/2020-5; FAPESP/MCTIC/CGI grant 2021/06923-0; and the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001.

References

[1] E.T.d. Camargo, E.P. Duarte, Running resilient mpi applications on a dynamic group of recommended processes, J. Braz. Comput. Soc. 24 (2018) 1–16, https://doi.org/10.1186/s13173-018-0069-z.

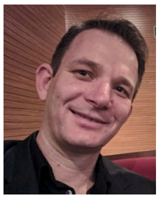
[2] T.D. Chandra, S. Toueg, Unreliable failure detectors for reliable distributed systems, J. ACM 43 (2) (1996) 225–267, https://doi.org/10.1145/226643.226647.

[3] J.P. de Araujo, L. Arantes, E.P. Duarte, L.A. Rodrigues, P. Sens, VCube-PS: a causal broadcast topic-based publish/subscribe system, J. Parallel Distrib. Comput. 125 (2019) 18–30, https://doi.org/10.1016/j.jpdc.2018.10.011.

[4] X. Défago, A. Schiper, P. Urbán, Total order broadcast and multicast algorithms: taxonomy and survey, ACM Comput. Surv. 36 (4) (2004) 372–421, https://doi.org/10.1145/1041680.1041682.
[5] E.P. Duarte Jr, L.A. Rodrigues, E.T. Camargo, R. Turchetti, The missing piece: a distributed system-level diagnosis model for the implementation of unreliable failure detectors, Computing 105 (12) (2023), https://doi.org/10.1007/s00607-023-01211-8.
[6] E.P. Duarte, L.C.E. Bona, V.K. Ruoso, VCube: a provably scalable distributed diagnosis algorithm, in: 2014 5th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems, 2014, pp. 17–22, https://doi.org/10.1109/ScalA.2014.14.
[7] C. Fetzer, M. Cristian, Fail-aware failure detectors, in: Proceedings 15th Symposium on Reliable Distributed Systems, 1996, pp. 200–209, https://doi.org/10.1109/RELDIS.1996.559722.
[8] V. Hadzilacos, S. Toueg, Fault-Tolerant Broadcasts and Related Problems, ACM Press, 1993, pp. 97–145, https://dl.acm.org/doi/10.5555/302430.302435.
[9] N. Hayashibara, X. Defago, R. Yared, T. Katayama, The /spl phi/ accrual failure detector, in: Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, 2004, pp. 66–78, https://doi.org/10.1109/RELDIS.2004.1353004.
[10] D. Jeanneau, L.A. Rodrigues, L. Arantes, E.P.D. Jr., An autonomic hierarchical reliable broadcast protocol for asynchronous distributed systems with failure detection, J. Braz. Comput. Soc. 23 (1) (2017) 15:1–15:14, https://doi.org/10.1186/s13173-017-0064-9.
[11] L. Lamport, Time, clocks, and the ordering of events in a distributed system, Commun. ACM 21 (7) (1978) 558–565, https://doi.org/10.1145/359545.359563.
[12] L. Lamport, The part-time parliament, ACM Trans. Comput. Syst. 16 (2) (1998) 133–169, https://doi.org/10.1145/279227.279229.
[13] LIGHTBEND, Akka: documentation, Available in: https://akka.io/docs/, 2020.
[14] S. Luan, V.D. Gligor, A fault-tolerant protocol for atomic broadcast, IEEE Trans. Parallel Distrib. Syst. 1 (3) (1990) 271–285, https://doi.org/10.1109/71.80156.
[15] G.M. Masson, D.M. Blough, G.F. Sullivan, System Diagnosis, Prentice-Hall, Inc., 1996, pp. 478–536.
[16] D. Ongaro, J. Ousterhout, In search of an understandable consensus algorithm, in: 2014 USENIX Annual Technical Conference (USENIXATC 14), 2014, pp. 305–319, https://dl.acm.org/doi/10.5555/2643634.2643666.
[17] A.A. Paznikov, A.V. Gurin, M.S. Kupriyanov, Implementation in actor model of leaderless decentralized atomic broadcast, in: 2020 9th Mediterranean Conference on Embedded Computing (MECO), 2020, pp. 1–4, https://doi.org/10.1109/MECO49872.2020.9134220.
[18] F. Pedone, A. Schiper, Modular Approach to Replication for Availability, Springer, 2010, pp. 41–57, https://doi.org/10.1007/978-3-642-11294-2_3.
[19] M. Poke, C.W. Glass, A dual digraph approach for leaderless atomic broadcast, in: 2019 38th Symposium on Reliable Distributed Systems (SRDS), 2019, pp. 203–20317, https://doi.org/10.1109/SRDS47363.2019.00031.
[20] M. Poke, T. Hoefler, C.W. Glass, Allconcur: leaderless concurrent atomic broadcast, in: 26th International Symposium on High-Performance Parallel and Distributed Computing, HPDC '17, Association for Computing Machinery, New York, NY, USA, 2017, pp. 205–218, https://doi.org/10.1145/3078597.3078598.
[21] Apache Ratis™, Open source java implementation for raft consensus protocol, Available in: https://ratis.apache.org/, 2021.
[22] L.A. Rodrigues, J. Cohen, L. Arantes, E.P. Duarte, A robust permission-based hierarchical distributed k-mutual exclusion algorithm, in: 2013 IEEE 12th International Symposium on Parallel and Distributed Computing, IEEE, 2013, pp. 151–158, https://doi.org/10.1109/ISPDC.2013.28.
[23] L.A. Rodrigues, L. Arantes, E.P.D. Jr., An autonomic implementation of reliable broadcast based on dynamic spanning trees, in: EDCC, IEEE Computer Society, 2014, pp. 1–12, https://doi.org/10.1109/EDCC.2014.31.
[24] L.A. Rodrigues, E.P. Duarte, L. Arantes, A distributed k-mutual exclusion algorithm based on autonomic spanning trees, J. Parallel Distrib. Comput. 115 (2018) 41–55, https://doi.org/10.1016/j.jpdc.2018.01.008.
[25] L.V. Ruchel, L.A. Rodrigues, R.C. Turchetti, L. Arantes, E.P. Duarte Jr., E.T. Camargo, A leaderless hierarchical atomic broadcast algorithm, in: Proceedings of the 11th Latin-American Symposium on Dependable Computing, LADC '22, Association for Computing Machinery, New York, NY, USA, 2023, pp. 61–66, https://doi.org/10.1145/3569902.3569914.
[26] P. Thanisch, Atomic commit in concurrent computing, IEEE Concurr. 8 (4) (2000) 34–41, https://doi.org/10.1109/4434.895104.
[27] P. Urban, X. Defago, A. Schiper, Neko: a single environment to simulate and prototype distributed algorithms, in: Proceedings 15th International Conference on Information Networking, 2001, pp. 503–511, https://doi.org/10.1109/ICOIN.2001.905471.
[28] P. Hunt, M. Konar, F. Junqueira, B. Reed, ZooKeeper: Wait-Free Coordination for Internet-Scale Systems, in: Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference, USENIX Association, USA, 2010, pp. 1–11, https://dl.acm.org/doi/10.5555/1855840.1855851.

Lucas V. Ruchel is a Laboratory Technician at the Instituto Federal do Paraná (IFPR), Cascavel, Brazil. He completed his master from the University of Western Paraná in 2022 in the area of Distributed Systems. He has a degree from the Federal University of Paraná in Internet Systems since 2017. His main research interests include Computer Networks, Distributed Systems, Software Defined Networks and Cybersecurity.

Edson Tavares de Camargo is an Associate Professor at Federal University of Technology – Paraná (UTFPR), Toledo, Brazil. He received a Ph.D. degree in Computer Science from the Federal University of Parana (UFPR) in 2017. During his Ph.D., he spent 1 year as a Ph.D. student at the Universitá della Svizzera Italiana (USI). He is a member of the Brazilian Computer Society where he served as chair of the Special Committee on Fault Tolerant Systems (CE-TF). Camargo coordinates projects in the area of Internet of Things and Smart Cities. His research interests include Computer Networks, Parallel Computing and Distributed Systems, their Dependability and Algorithms.



Luiz Antonio Rodrigues is an associate professor at Western Parana State University (Unioeste), Brazil, and a member of the GPISC (Computer Systems Research and Innovation Group). He received his PhD in Computer Science from the Federal University of Parana (UFPR) and spent a year in the Computer Science Laboratory (LIP6) at Sorbonne University. He is an effective member of the Brazilian Computing Society (SBC) and was the coordinator of the Special Committee on Fault-Tolerant Systems of the SBC (CE-TF 2019-2020). His main interests are in computer science, with a focus on computer networks, fault tolerance, distributed systems, and Dependability.

Rogério C. Turchetti is an Associate Professor at Federal University of Santa Maria, Santa Maria, Brazil. He received a Ph.D. degree in computer science from Federal University of Parana, Brazil, 2017, the M.Sc. degrees in production engineering with emphasis on information systems from Federal University of Santa Maria, Santa Maria, Brazil, in 2006. His research interests include Computer Network and Distributed Systems, their Dependability and Algorithms. His recent research is focused on the dependability in Network Function Virtualization and Software Defined Network.

Luciana Arantes Luciana Arantes received the PhD degree in computer science from Paris 6 University (UPMC), France, in 2000. She has been working as an associate professor at Sorbonne University (ex-UPMC), France, since 2001. She was a research member of INRIA/LIP6 Regal project-team from 2005–2017 and now research member of INRIA/LIP6 Delys Group. She was member of the program committee of several conferences in the area of distributed systems and parallelism (ICDCS, EDCC, ISSRE, NCA, SBAC-PAD, DAIS, etc.) and local organized chair of SBAC-PAD 2014 and EDCC 2015.

Elias Procópio Duarte Jr. is a Full Professor at Federal University of Parana, Brazil. His research interests include computer networks and distributed systems, their dependability and algorithms. With over 300 peer-reviewer papers, and 130 students supervised, Prof. Duarte is Associate Editor of the Computing (Springer) journal and IEEE Transactions on Dependable and Secure Computing, and has served as chair of more than 25 conferences and workshops, including TPC Chair of GLOBECOM'2024, SRDS'2018 and ICDCS'2021. He chaired the Brazilian National Laboratory on Computer Networks (2012–2016), and is a member of the Brazilian Computing Society and a Senior Member of the IEEE.