Journal of the Brazilian Computer Society

RESEARCH Open Access

An autonomic hierarchical reliable broadcast protocol for asynchronous distributed systems with failure detection

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

Abstract

Reliable broadcast is a fundamental building block in fault-tolerant distributed systems. It consists of a basic primitive that provides agreement among processes of the system on the delivery of each broadcast message, i.e., either none or all correct processes deliver the message, despite failures of processes. In this work, we propose a reliable broadcast solution on top of VCube, assuming that the system is asynchronous. VCube is an autonomic monitoring layer that organizes processes on a hypercube-like overlay which provides several logarithmic properties even in the presence of processes failures. We consider that processes can fail by crashing, do not recover, and faults are eventually detected by all correct processes. The protocol tolerates false suspicions by sending additional messages to suspected processes but logarithmic properties of the algorithm are still kept. Experimental results show the efficiency of the proposed solution compared to an one-to-all strategy.

Keywords: Implementation of distributed system, Autonomic computing, Fault-tolerant broadcasts, Spanning trees

Introduction

Broadcast is a basic component to implement numerous distributed applications and services such as notification, content delivery, replication, and group communication [1–3]. A process in a distributed system uses broadcast to send a message to all other processes in the system. Formally, reliable broadcast is defined in terms of two primitives: BROADCAST(m), which is defined by the broadcast algorithm and called by the application to disseminate m to all processes, and DELIVER(m), which is defined by the application and called by the broadcast algorithm when message m has been received.

From an implementation point of view, the broadcast primitive sends point-to-point messages to each process of the system. Broadcast algorithms must ensure that, if a correct[1] process broadcasts a message, then it eventually delivers the message (*validity* property). Furthermore, every correct process delivers a message at most once and only if that message was previously broadcast by some process (*integrity* property) [4]. However, if the sender fails during the execution of the broadcast primitive, some processes might not receive the broadcast message. In order to circumvent this problem, *reliable broadcast* ensures, besides the validity and integrity, that even if the sender fails, every correct process delivers the same set of messages (*agreement* property).

There exists a considerable amount of literature on reliable broadcast algorithms, such as the one where all correct receivers retransmit all received messages guaranteeing then the delivery of all broadcast messages by the other correct processes of the system [5]. We are particularly interested in solutions that use failure detectors [6] which notify the broadcast algorithm about processes failures. Upon receiving such an information, the algorithm reacts in accordance to tolerate the failure. Another important feature of reliable broadcast algorithms concerns performance, which is related to how broadcast messages are disseminated to processes. Aiming at scalability and message complexity efficiency, many reliable broadcasts organize processes on logical spanning trees. Messages are then disseminated over the constructed tree, therefore providing logarithmic performance [7–11].

*Correspondence: luiz.rodrigues@unioeste.br
[2]Department of Computer Science, Western Paraná State University, Rua Universitária, 2069, Jardim Universitário, CEP 85.814-110 Cascavel-PR, Brazil
Full list of author information is available at the end of the article

This work presents an autonomic reliable broadcast algorithm in which messages are transmitted over spanning trees dynamically built on top of a logical hierarchical hypercube-like topology. Autonomic systems constantly monitor themselves and automatically adapt to changes [12]. The logical topology is maintained by the underlying VCube monitoring system which also detects failures [13]. VCube is a distributed diagnosis layer responsible for organizing processes of the system in a virtual hypercube-like cluster-based topology which is dynamically re-organized in case of process failure. When invoked, VCube also gives information about the liveness of the processes that compose the system.

We assume a fully connected *asynchronous* system in which processes can fail by crashing, and crashes are permanent. Links are reliable. A process that invokes the reliable broadcast primitive starts the message propagation over a spanning tree. This tree is built on demand with information obtained from VCube and is dynamically reconstructed upon the detection of a node crash (process failure).

In a previous work [14], we proposed an autonomic reliable broadcast algorithm on top of the Hi-ADSD, a previous version of VCube. The algorithm guarantees several logarithmic properties, even when nodes fail and allows transparent and efficient spanning tree reconstructions. However, for that solution, we considered a synchronous model for the system, i.e., there exist known bounds on message transmission delays and processors' speed and, consequently, VCube provides a perfect process failure detection. On the one hand, the advantage of such synchronous assumption is that there was no false failure suspicions and, thus, if VCube notifies the broadcast algorithm that a given process is faulty, the algorithm is sure that it can stop sending message to this faulty process and then removes it forever from the spanning tree constructions. On the other hand, the synchronous assumption considerably restrains the distributed systems and applications that can use the broadcast protocol since many of the current network environments are considered asynchronous (there exist no bounds on message transmission delay or on processors' speed).

Hence, considering the above constraints, we propose in this article a new autonomic reliable broadcast algorithm, using VCube in an asynchronous model. We assume that the failure detection service provided by VCube is unreliable since it can make mistakes by erroneously suspecting a correct process (false suspicion) or by not suspecting temporally a node that has actually crashed. However, upon the detection of a mistake, VCube corrects it. Furthermore, it also ensures that eventually all failures are detected (*strong completeness* property). Note that such false suspicions render a broadcast algorithm much more complex than the previous one since it can induce a violation of the reliable broadcast. For instance, the algorithm must ensure that a falsely suspected process must receive and deliver, only once, all broadcast messages, otherwise the agreement and integrity properties would be violated. In our solution, false suspicions are tolerated by sending special messages to those processes suspected of having failed. We must also emphasize that our aim is to provide a reliable broadcast algorithm which is efficient, i.e., that keeps, as much as possible, the logarithmic properties of the spanning tree dissemination over the hypercube-like topology. Our algorithm tolerates up to $n - 1$ node crashes.

In addition to the specification, we present experimental results in which the proposed algorithm was compared to a one-to-all point-to-point broadcast protocol and another algorithm based on hypercubes that do not employ failure detectors. The results confirm the efficiency of the proposed broadcast considering the following: (1) the latency to deliver the message to all correct processes; and (2) the total number of messages, including retransmissions in case of failures/false suspicion.

The rest of this paper is organized as follows: first, we present the Related work. Next, we define the System model and briefly describe VCube monitoring algorithm and the hypercube-like topology. Next, we present the proposed autonomic reliable broadcast algorithm for asynchronous systems, including proofs and performance issues. Experimental results are reported in sequence. We finish this article by presenting final remarks in the Conclusion section.

Related work

Many reliable broadcast algorithms of the literature exploit spanning trees such as [7–11].

Schneider et al. introduced in [7] a tree-based fault-tolerant broadcast algorithm whose root is the process that starts the broadcast. Each node forwards the message to all its successors in the tree. If one process p that belongs to the tree fails, another process assumes the responsibility of retransmitting the messages that p should have transmitted if it were correct. Like to our approach, processes can fail by crashing and the crash of any process is detected after a finite but unbounded time interval by a failure detection module. However, the authors do not explain how the algorithm rebuilds or reorganizes the tree after a process failure.

In [8], a reliable broadcast algorithm is provided by exploiting disjoint paths between pairs of source and destination nodes. Multiple-path algorithms are particularly useful in systems that cannot tolerate the time overhead for detecting faulty processors, but there is an overhead in the number of duplicated messages. On a star network with n edges, the algorithm constructs $n - 1$ directed edge-disjoint spanning trees. Fault tolerance is achieved by retransmitting the same messages through a number of edge-disjoint spanning trees. The algorithm tolerates up to $n - 2$ failure of nodes or edges and can be adjusted depending on the network reliability. Similarly, Kim et al. [9] propose a tree-based solution to disseminate a message to a large number of receivers using multiple data paths in a context of time-constrained dissemination of information. Thus, arguing that reliable extensions using ack-based failure recovery protocols cannot support reliable dissemination with time constraints, the authors exploit the use of multiple data paths trees in order to conceive a fast and reliable multicast dissemination protocol. Basically, the latter is a forest-based (multiple parents-to-multiple children) tree structure where each participant node has multiple parents as well as multiple children. A third work that exploits multi-paths spanning trees is [10] where the authors present a reliable broadcast algorithm that runs on a hypercube and uses disjoint spanning trees for sending a message through multiple paths.

Raynal et al. [11] presented a reliable tree-based broadcast algorithm suited to dynamic networks in which message transfer delays are bounded by a constant of δ units of time. Whenever a link appears, its lifetime is at least δ units of time. The broadcast is based on a spanning-tree on top of which processes forward received messages to their respective neighbors. However, as the system is dynamic, the set of current neighbors of a process p may consists of a subset of all its neighbors and, therefore, p has to additionally execute specific statements when a link reappears, i.e., forwards the message on this link if it is not sure that the destination process already has a copy of it.

Similarly to our approach, many existing reliable broadcast algorithms exploit spanning trees constructed on hypercube-like topologies [10, 15, 16]. In [15], the authors present a fault-tolerant broadcast algorithm for hypercubes based on binomial trees. The algorithm can recursively regenerate a faulty subtree, induced by a faulty node, through one of the leaves of the tree. On the other hand, unlike our approach, there is a special message for advertising that the tree must be reconstructed and, in this case, broadcast messages are not treated by the nodes until the tree is rebuilt. The HyperCast protocol proposed by [16] organizes the members of a multicast group in a logical tree embedded in a hypercube. Labels are assigned to nodes, and the one with the highest label is considered to be the root of the tree. However, due to process failures, multiple nodes may consider themselves to be the root and/or different nodes may have different views of which node is the root.

Leitão et al. present in [1] the *HyParView*, a hybrid broadcast solution that combines a tree-based strategy with a gossip protocol. A broadcast tree is created embedded on a gossip-based overlay. Broadcast is performed by using gossip on the tree branches. Later, some of the authors proposed a second work [17] where they introduced *Thicket*, a decentralized algorithm to build and maintain multiple trees over a single unstructured P2P unstructured overlay for information dissemination. The authors argue that multiple trees approach allow that each node to be an internal node in just a few trees and a leaf node in the remaining of the trees providing, thus, load distribution as well as redundant information for fault-tolerance.

In [14], we presented a reliable broadcast solution based on dynamic spanning trees on top of the Hi-ADSD, a previous version of the VCube. Multiple trees are dynamically built that include all correct nodes, where each tree root corresponds to the node that called a broadcast primitive. Contrarily to the current work, that solution considers that the system model is synchronous and that VCube guarantees perfect failure detection. In the current work, false suspicions are possible and are dealt with to guarantee the correction of the algorithm.

System model

We consider a distributed system that consists of a finite set P of $n > 1$ processes $\{p_0, .., p_{n-1}\}$ that communicate only by message passing. Each single process executes one task and runs on a single processor. Therefore, the terms *node* and *process* are used interchangeably in this work.

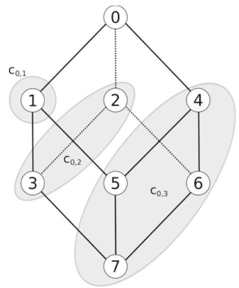
The system is asynchronous, i.e., relative processor speeds and message transmission delays are unbounded. Links are reliable, and, thus, messages exchanged between any two correct processes are never lost, corrupted or duplicated. There is no network partitioning.

Processes communicate by sending and receiving messages. The network is fully connected: each pair of processes is connected by a bidirectional point-to-point channel. Processes are organized in a virtual hypercube-like topology, called VCube (see next subsection). 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* or *fault-free*; otherwise, it is considered to be *faulty*. After any crash, the topology changes, but the logarithmic properties of the hypercube are kept.

We consider that the primitives to send and receive a message are atomic, but the broadcast primitives are not.

VCube topology and failure detection

Let n be the number of processes in the system P. VCube [13] is a distributed diagnosis algorithm that organizes the processes of the system P in a virtual hypercube-like topology. In a hypercube of d dimensions, called d-VCube, there are 2^d processes[2]. A process i groups the other $n - 1$ processes in $\log_2 n$ clusters, such that cluster number s has size 2^{s-1}. Figure 1 shows the hierarchical organization for a 3-VCube. The ordered set of processes in each cluster s is defined by the $c_{i,s}$, $i = 0, .., n - 1$, also



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)

$$c_{i,s} = i \oplus 2^{s-1} \; k \; c_{i \oplus 2^{s-1},k} \mid k = 1, .., s-1$$

Fig. 1 Hierarchical cluster-based logical organization in a 3-dimensional VCube

shown in Fig. 1, in which $\oplus$ denotes the bitwise exclusive *or* operator (xor).

To determine the state of each process, a process i tests another process in the $c_{i,s}$ executing a test procedure and waiting for a reply. If the correct reply is received within an expected time interval, the monitored process is considered to be correct or fault-free. Otherwise, it is considered to be *faulty* or *suspected*. We should point out that in an asynchronous model, which is the case in the current work, VCube provides an unreliable failure detection since it can erroneously suspect a correct process (false suspicion). If later it detects its mistake, it corrects it. On the other hand, according to the properties proposed by Chandra and Toueg [6] for unreliable failure detectors, VCube ensures the *strong completeness* property: eventually every process that crashes is permanently suspected by every correct process. Since there are false suspicions, VCube does not provide any *accuracy* property.

The virtual hypercube topology is created by connecting each process i to the first correct process in the $c_{i,s}$, $s = 1, .., \log_2 n$. For instance, considering all processes are correct in Fig. 1, process p_0 is connected to $\{1, 2, 4\}$, i.e., the first correct processes in each cluster $c_{0,1} = (1)$, $c_{0,2} = (2, 3)$, and $c_{0,3} = (4, 5, 6, 7)$.

In order to avoid that several processes test the same processes in a given cluster, process i executes a test on process $j \in c_{i,s}$ only if process i is the first faulty-free process in $c_{j,s}$. Thus, any process (faulty or fault-free) is tested at most once per round, and the latency, i.e., the number of rounds required for all fault-free processes to identify that a process has become faulty is $\log_2 n$ in average and $\log_2^2 n$ rounds in the worst case.

The proposed reliable broadcast algorithm

A reliable broadcast algorithm ensures that the same set of messages is delivered by all correct processes, even if the sender fails during the transmission. Reliable broadcast presents three properties [5]:

- *Validity*: if a correct process broadcasts a message m, then it eventually delivers m.
- *Integrity*: every correct process delivers the same message at most once (no duplication) and only if that message was previously broadcast by some process (no creation).
- *Agreement*: if a message m is delivered by some correct process p_i, then m is eventually delivered by every correct process p_j. Note that the agreement property still holds if m is not delivered by any process.

Our reliable broadcast algorithm exploits the virtual topology maintained by VCube, whenever possible. Each process creates, thus, a spanning tree rooted at itself to broadcast a message. The message is forwarded over the tree and, for every message that a node of the tree sends to one of its correct neighbor, it waits for the corresponding acknowledgement from this neighbor, confirming the reception of the message. Algorithm 1 presents the pseudo-code of our proposal reliable broadcast protocol for an asynchronous system with $n = 2^d$ processes. The dimension of VCube is, therefore, d. A process gets information, not always reliable, about the liveness of the other processes by invoking VCube. Hence, the trees are dynamically built and autonomically maintained using the hierarchical cluster structure and the knowledge about faulty (or falsely suspected) nodes. The algorithm tolerates up to $n - 1$ failures.

Let i and j be two different processes of the system. The function $cluster_i(j) = s$ returns the identifier s of the cluster of process i that contains process j, $1 \leq s \leq d$. For instance, in the 3-VCube as shown in Fig. 1, $cluster_0(1) = 1$, $cluster_0(2) = cluster_0(3) = 2$ and $cluster_0(4) = cluster_0(5) = cluster_0(6) = cluster_0(7) = 3$.

Message types and local variables

Let m be the application message to be transmitted from a sender process, denoted *source*, to all other processes in the system. We consider three types of messages:

- $\langle TREE, m \rangle$: message broadcast by the application that should be forwarded over VCube to all processes considered to be correct by the sender;

- $\langle DELV, m\rangle$: message sent to processes suspected of being faulty in order to avoid that false suspicions induce the no delivery of the message by correct processes. The recipient of the message should deliver it but not forward it;
- $\langle ACK, m\rangle$: used as an acknowledgement to confirm that a TREE message related to m was received.

For the sake of simplicity, we use TREE, DELV, and ACK to denote these messages.

Every message m carries two fielding containing: (1) the identifier of the process that broadcast m and (2) a *timestamp* generated by the process local counter which uniquely identifies m. The first message broadcast by a process i has timestamp 0 and at every new broadcast, i increments the timestamp by 1. The algorithm can extract these two parameters from m by respectively calling the functions $source(m)$ and $ts(m)$.

Process i keeps the following local variables:

- $correct_i$: the set of processes considered correct by process i.
- $last_i[n]$: an array of n elements to keep the last messages delivered by i. For each process j, $last_i[j]$ is the last message broadcasted by j that was delivered by i. If no message was received from j, $last_i[j] = \perp$.
- ack_set_i: a set with all pending acknowledgement messages which process i is waiting for. For each message $\langle TREE, m\rangle$ received by i from process j and retransmitted to process k, an element $\langle j, k, m\rangle$ is added to this set; The symbol $\perp$ represents a null element. The asterisk is used as a wildcard. For instance, $\langle j, *, m\rangle$ means all pending *acks* for a message m received from process j and forwarded to any other process.
- $pending_i$: list of the messages received by i that were not delivered yet because they are "out of order" with regard to their timestamp, i.e., $ts(m) > ts(last_i(source(m))) + 1$, ensuring the FIFO order.
- $history_i$: the history of messages that were already broadcasted by i. This set is used to prevent sending the same message to the same cluster more than once. $\langle j, m, h\rangle \in history_i$ indicates that the message m received from process j was already sent by i to the clusters $c_{i,s}$ for all $s \in [1, h]$.

Algorithm 1 Reliable broadcast primitives at process i

```
1: $\forall j \in [0,..,n-1] : last_i[j] \leftarrow \perp$          ▷ Initialization
2: $ack\_set_i \leftarrow \emptyset$
3: $correct_i \leftarrow \{0, ..., n-1\}$
4: $pending_i \leftarrow \emptyset$
5: $history_i \leftarrow \emptyset$

6: procedure BROADCAST(message $m$)          ▷ Invoked by the application
7:     wait until $ack\_set_i \cap \{\langle \perp, *, last_i[i]\rangle\} = \emptyset$
8:     $last_i[i] \leftarrow m$
9:     DELIVER($m$)
10:    BROADCAST_TREE($\perp, m, log_2 n$)

11: procedure BROADCAST_TREE(process $j$, message $m$, integer $h$)   ▷ Send $m$ to all first fault-free node in each cluster smaller than $h$
12:    $start \leftarrow 0$
13:    if $\exists x : \langle j, m, x\rangle \in history_i$ then
14:        $start \leftarrow x$
15:        $history_i \leftarrow history_i \setminus \{\langle j, m, x\rangle\}$
16:    $history_i \leftarrow history_i \cup \{\langle j, m, max(start, h)\rangle\}$
17:    if $start < h$ then
18:        for all $s \in [start + 1, h]$ do
19:            BROADCAST_CLUSTER($j, m, s$)

20: procedure BROADCAST_CLUSTER(process $j$, message $m$, integer $s$)   ▷ Send $m$ to the the first fault-free node and all suspect process in cluster $c_{i,s}$
21:    $sent \leftarrow false$
22:    for all $k \in c_{i,s}$ do
23:        if $sent = false$ then
24:            if $\langle j, k, m\rangle \in ack\_set_i$ and $k \in correct_i$ then
25:                $sent \leftarrow true$
26:            else if $k \in correct_i$ then
27:                SEND($\langle TREE, m\rangle$) to $p_k$
28:                $ack\_set_i \leftarrow ack\_set_i \cup \{\langle j, k, m\rangle\}$
29:                $sent \leftarrow true$
30:        else if $\langle j, k, m\rangle \notin ack\_set_i$ then
31:            SEND($\langle DELV, m\rangle$) to $p_k$

32: procedure CHECK_ACKS(process $j$, message $m$)
33:    if $j \neq \perp$ and $ack\_set_i \cap \{\langle j, *, m\rangle\} = \emptyset$ then
34:        SEND($\langle ACK, m\rangle$) to $p_j$

35: procedure HANDLE_MESSAGE(process $j$, message $m$)
36:    $pending_i \leftarrow pending_i \cup \{m\}$
37:    while $\exists l \in pending_i : source(l) = source(m) \wedge (ts(l) = ts(last_i[source(l)]) + 1$
38:        or $last_i[source(l)] = \perp \wedge ts(l) = 0)$ do
39:        $last_i[source(l)] \leftarrow l$
40:        $pending_i \leftarrow pending_i \setminus \{l\}$
41:        DELIVER($l$)
42:    if $source(m) \notin correct_i$ then
43:        BROADCAST_TREE($j, last_i[source(m)], log_2 n$)

44: upon receive $\langle TREE, m\rangle$ from $p_j$
45:    HANDLE_MESSAGE($m$)
46:    BROADCAST_TREE($j, m, cluster_i(j) - 1$)
47:    CHECK_ACKS($j, m$)

48: upon receive $\langle DELV, m\rangle$ from $p_j$
49:    HANDLE_MESSAGE($m$)
```

```
50: upon receive $\langle ACK, m\rangle$ from $p_j$
51:     for all $k = x : \langle x, j, m\rangle \in ack\_set_i$ do
52:         $ack\_set_i \leftarrow ack\_set_i \backslash \{\langle k, j, m\rangle\}$
53:         CHECK_ACKS($k, m$)

54: upon notifying crash($j$)
55:     $correct_i \leftarrow correct_i \backslash \{j\}$
56:     for all $p = x, m = y : \langle x, j, y\rangle \in ack\_set_i \cap \{\langle *, j, *\rangle\}$ do
57:         BROADCAST_CLUSTER($p, m, cluster_i(j)$)
58:         $ack\_set_i \leftarrow ack\_set_i \backslash \{\langle p, j, m\rangle\}$
59:         CHECK_ACKS($p, m$)
60:     if $last_i[j] \neq \perp$ then
61:         BROADCAST_TREE($j, last_i[j], log_2 n$)

62: upon notifying up($j$)
63:     $correct_i \leftarrow correct_i \cup \{j\}$
```

Algorithm description

Process i broadcasts a message by calling the BROADCAST(m) function. Line 7 ensures that a new broadcast starts only after the previous one has been completed, i.e., there are no pending *acks* for the $last_i[i]$ message. Note that some processes might not have received the previous message yet because of false suspicions. Then, the received message m is locally delivered to i (line 9) and, by calling the function BROADCAST_TREE (line 10), i forwards m to its neighbors in VCube. To this end, it calls, for each cluster $s \in [1, log_2 n]$, the function BROADCAST_CLUSTER that sends a TREE message to the first process k which is correct in the cluster (line 27). To those processes that are not correct, i.e, suspected of having crashed and placed before k in the cluster, a DELV message (line 31) is sent to them. Notice that in both cases, the messages are sent provided i has not already forwarded m, received from j, to k. For every TREE message sent, the corresponding ack is included in the list of pending *acks* (line 28). Note that in the notation used, upon-clauses cannot preempt the procedures nor vice-versa.

Upon reception of a message $\langle TREE, m\rangle$ from process j (line 44), process i calls the function HANDLE_MESSAGE. In this function, m is added to the set of pending messages a nd then all pending messages which were broadcast by the same process that broadcast m ($source(m)$) are delivered in increasing order of timestamps, provided no message is missing in the sequence of timestamps (lines 36 - 41). In the same HANDLE_MESSAGE function, if i suspects that $source(m)$ failed, it restarts the broadcast of $last_i[source(m)]$ (line 43) to ensure that every correct process receives the message even if $source(m)$ crashed in the middle of the broadcast. Otherwise, by calling function BROADCAST_TREE with parameter $h = cluster_i(j) - 1$ (line 46), m is forwarded to all neighbors of i in each sub-cluster of i that should receive m.

If process i is a leaf in the spanning tree of the broadcast ($cluster_i(j) - 1 = 0$) or if all neighbors of i (i.e., children of i in the tree) that should receive the message are suspected of being crashed, i sends an ACK message to the process which sent m to it, by calling function CHECK_ACKS (line 34).

If process i receives a $\langle DELV, m\rangle$ message from j (line 48), it means that j falsely suspects i of being crashed and has decided to trust another process with the forwarding of the message to the rest of the tree. Therefore, i can simply call the HANDLE_MESSAGE function to deliver the message and does not need to call BROADCAST_TREE.

Whenever i receives a message $\langle ACK, m\rangle$, it removes the corresponding *ack* from set of pending *acks* (line 52) and, by calling the function CHECK_ACKS, if there are no more pending *acks* for message m, i sends an ACK message to the process j which sent m to it (line 34). If $j = \perp$, the ACK message has reached the process that has broadcast m ($source(m)$) and the ACK message does not need to be forwarded.

The detection of the failure of process j is notified to i (crash(j)). It is worth pointing out that this detection might be a false suspicion. Three actions are taken by i upon receiving such a notification: (1) update of the set of processes that it considers correct (line 55); (2) removal from the set of pending *acks* of those *acks* whose related message m has been retransmitted to j (line 58); (3) re-sending to k, the next neighbor of j in the cluster of j (if k exists), of those messages previously sent to j. The re-sending of these messages triggers the propagation of messages over a new spanning tree (line 57).

Finally, in case j crashes, i has to re-broadcast the last message broadcast by j (line 61). Notice that, in this case, the *history* variable is used in order to prevent i from re-rebroadcasting the message to those clusters that i has already sent the same message.

If VCube detects that it had falsely suspected process j, it corrects its mistake and notifies i which then includes j in its set of correct processes (line 63).

Note that the restriction that to broadcast a new message it is only done after the previous broadcast has completed (line 7) was included to simplify the delivery procedure, so that it is not necessary to keep all messages received, but only one: the last broadcast. It also presents advantages when the source process is detected as faulty, because only the last message needs to be broadcast again. Note that it is not difficult to allow concurrent messages, but handling them will make the algorithm more difficult to present and understand.

Example of execution

Consider the 3-VCube topologies in Fig. 2 where process 0 (p_0) broadcasts a message. First, consider the fault-free scenario represented by Fig. 2a. Only TREE and ACK

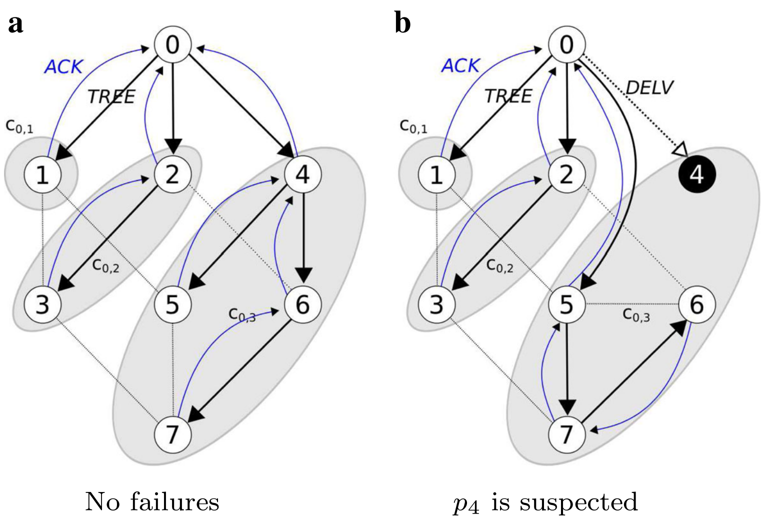


Fig. 2 Reliable broadcast started by process 0 (p_0). **a** Messages in a fault-free execution. **b** Messages when process p_4 is suspected

messages are involved. After delivering the TREE message to itself, p_0 sends a copy of the message to p_1, p_2, and p_4, which are neighbors of p_0 and the first correct processes on each of 0's clusters. Process p_1 receives the TREE message and, as p_1 is member of cluster 1 of process p_0 (which means it is a leaf in p_0's tree), it delivers the message and send the ACK to p_0. Process p_2 is in cluster 2 of process p_0. In this case, it must retransmit the message to its cluster 1, which contains process p_3 ($c_{2,1} = 3$). Process p_3 is also a leaf, since it is in cluster 1 of p_2. So, p_3 delivers the message and sends the ACK back to p_2, and p_2 in turn, sends the ACK back to p_0. Process p_4 is in cluster 3 of p_0. It means p_4 needs to retransmit the message to the first free processes in clusters $c_{4,1}$ and $c_{4,2}$, i.e., p_5 and p_6. Process p_6, on the other hand, still needs to retransmit the message to its cluster $c_{6,1}$ that contains only process p_7. Processes p_5 and p_7 operate in the same way as processes p_1 and p_3. Each ACK is propagated back to the source p_0 following the reverse path of the spanning tree. As soon as process p_0 receives the ACKs from p_1, p_2, and p_4, the broadcast is finished.

Now, consider the scenario represented by Fig. 2b, in which p_4 is considered suspected by the source p_0. In this case, process p_0 sends TREE messages to p_1 and p_2, as done in the fault-free scenario. Process p_4 is not the first fault-free process in the cluster 3 anymore. So, the TREE message is sent to p_5 ($c_{0,3} = (4, 5, 6, 7)$) and a DELV message is sent to p_4, since in the asynchronous environment there is no guarantee if p_4 is really faulty. TREE and ACK messages are propagated to p_2, p_7,and p_6, and forwarded to p_0 following the trivial rules already described.

In both scenarios, all correct processes receive a copy of the TREE message broadcast by p_0, as required by the broadcast properties.

Proof of correctness

In this section, we will prove that Algorithm 1 implements a reliable broadcast.

Lemma 1 *Algorithm* 1 *ensures the validity property of reliable broadcast.*

Proof If a process i broadcasts a message m, the only way that i would not deliver m is if i waits forever in line 7. This wait is interrupted when the set ack_set_i contains no more pending acknowledgements related to the message $last_i[i]$ previously broadcast by i.

For any process j that i sent $last_i[i]$ to, i added a pending ack in ack_set_i (line 28). If j is correct, then it will eventually answer with an *ACK* message (line 34) and i will remove $\langle \perp, j, last_i[j] \rangle$ from ack_set_i on line 52. If j is faulty, then i will eventually detect the crash and remove the pending ack in line 58.

As a result, all of the pending acks for $last_i[i]$ will eventually be removed from ack_set_i and i will deliver m on line 9.

Line 9 then ensures that i will deliver the message before broadcasting it. □

Lemma 2 *For any processes* i *and* j*, the value of* $ts(last_i[j])$ *only increases over time.*

Proof For the sake of simplicity, we take the convention that $ts(\perp) = -1$. The $last_i$ array is only modified in lines 8 and 39.

The first case can only happen when i broadcasts a new message m, and since timestamps of new messages sent by a same processes have to be increasing, $ts(m) > ts(last_i[i])$. When i calls BROADCAST(m)

procedure, $ts(last_i[\,i\,])$ will therefore increase in line 8.

The other way for $last_i$ to be modified is on line 39. $last_i\,[source(l)]$ will then be updated with message l if $last_i\,[source(l)] = \bot$ and $ts(l) = 0$ (and therefore $ts(last_i\,[source(l)]) = -1 < ts(l)$), or if $ts(l) = ts(last_i\,[source(l)]) + 1$. It follows that $last_i[\,source(l)$ is only updated if the new value of $ts(last_i[\,source(l)]\,)$ would be superior to the old one. □

Lemma 3 *Algorithm* 1 *ensures the integrity property of reliable broadcast.*

Proof Processes only deliver a message if they are broadcasting it themselves (line 9) or if the message is in their $pending_i$ set (line 41). Messages are only added to the $pending_i$ set on line 36 after they have been received from another process. Since the links are reliable and do not create messages, it follows that a message is delivered only if it was previously broadcasted (there is no creation of messages).

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

- **source(m) = i.** Process i called BROADCAST(m). As proved in Lemma 1, i will deliver m in line 9. Since the BROADCAST procedure is only called once with a given message, the only way that i would deliver m a second time is in line 41. Since $last_i[\,i]$ was set to m on line 8, it follows from Lemma 2 that m will never qualify to pass the test in lines 37 – 38.
- **source(m) ≠ i.** Process i is not the emitter of message m and did not call BROADCAST(m). Therefore the, the only way for i to deliver m is in line 41. Before i delivers m for the first time, it sets $last_i[\,source(m)]$ to m on line 39. It then follows from Lemma 2 that m will never again qualify to pass the test on lines 37 – 38, and therefore i can deliver m at most once.

□

Lemma 4 *Algorithm* 1 *ensures the agreement property of reliable broadcast.*

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

- *i is correct.* It can be shown by induction that every correct process receives m.
  As a basis of the induction, let us consider the case where $n = 2$ and $P = \{i, j\}$. It follows that $c_{i,1} = \{j\}$. Therefore, i will send m to j in line 31 if i suspects j or in line 27 otherwise. If j is correct, it will eventually receive m since the links are reliable and will deliver m in line 41. i will also deliver m by virtue of the validity property.
  We now have to prove that if every correct process receives m for $n = 2^k$, it 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 BROADCAST_TREE and BROADCAST_CLUSTER procedures ensure that for every $s \in [1, k+1]$, 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. If j is faulty and i detected the crash prior to the broadcast, i will send the message to j anyway in case it is a false suspicion (line 31) but it will also send it to another process in $c_{i,k+1}$ as a precaution (line 27). i will keep doing so until it has sent the *TREE* message to a non-suspected process in $c_{i,k+1}$, or until it has sent the message to all the processes in $c_{i,k+1}$.
  If j is faulty and i only detects the crash after the broadcast, the BROADCAST_CLUSTER procedure will be called again in line 57, which ensures once again that i will send the message to a non-suspected 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 P_2 subsystem in line 46.
  Since a correct process broadcasts m in both subsystems P_1 and P_2, and since both subsystems are of size 2^k, it follows that every correct process in P will eventually receive m.
- *i is faulty.* If i crashes before sending m to any process, then no correct process delivers m and the agreement property is verified. If i crashes after the broadcast is done, then everything happens as if i was correct. If i crashes after sending m to some processes and a correct process j receives m, then j will eventually detect the failure of i. If j detects the crash before receiving m, when it receives m it will restart a full broadcast of m in line 43. If j only detects the crash of i after receiving m, it will also restart a full broadcast of m in line 61. Since j is correct, every correct process will eventually receive m.

□

Theorem 1 *Algorithm* 1 *correctly implements reliable broadcast.*

Proof The proof follows directly from Lemmas 1, 3, and 4. □

Discussion on the performance

The goal of exploiting the VCube overlay in our solution is to provide an efficient broadcast where each process sends at most $log_2 n$ messages. However, this complexity cannot be ensured at all times in an asynchronous system where false suspicions can arise. Algorithm 1 aims to take advantage of VCube whenever possible while still ensuring the properties of a reliable broadcast despite false suspicions.

In the best case scenario where no process is ever suspected of failure, each process will send at most one message per cluster (line 27). Therefore $n-1$ *TREE* messages will be sent in total (since no process will be sent the same message twice) with no single process sending more than $log_2 n$ messages. This is the example presented in Fig. 2a.

If a process other than the source of the broadcast is suspected before the broadcast, there will be $n-2$ *TREE* messages and one *DELV* message sent. A single process might send up to $log_2 n$ *TREE* messages plus one *DELV* message per suspected process. This is the example of Fig. 2b.

If the source of the broadcast suspects everyone else, then it will send $n-1$ *DELV* messages. In this case, Algorithm 1 is equivalent to a *one-to-all* algorithm where one process sends the message directly to all others, losing, thus, the advantages of tree topology properties, such as scalability.

The main cost of suspicions lies in the fact that when a process is suspected, its last broadcast message must be resent. This is the purpose of lines 43 and 61. Such re-broadcast is an unavoidable consequence of the existence of false suspicions, necessary in order to ensure the agreement property of reliable broadcast.

Note that the fact that the information about a node failure is false or true has the same impact on the performance of the broadcast algorithm in terms of message complexity.

Experimental evaluation

In this section, we present simulation experimental results obtained with Neko [18], a Java framework to simulate and prototype distributed algorithms.

We compare our broadcast solution (VCube-RB) with two other approaches. In the one-to-all algorithm (All-RB), a source node sends a message directly to all other nodes in the system, i.e., without using a spanning tree. The sender process waits for acknowledgment messages to be sure that the related broadcast has been completed. After a failure is detected, the sender is notified to avoid waiting indefinitely. The second approach, named DPath-RB, was proposed by [8] and implements reliable broadcast in hypercube systems without needing failure detectors. Multiple copies of the messages are sent along disjoints paths. No acknowledgments messages are sent, since the multiple paths guarantee the deliver, but only if fewer than $log_2 n$ processes are faulty, which ensures that the graph representing the hypercube is connected.

The tests were organized into fault-free and faulty scenarios. Two metrics were used to evaluate the performance of both algorithms: (1) the mean latency to broadcast one message to all correct processes, and (2) the mean number of messages to complete a broadcast.

Simulation parameters

The simulation model is based on [19]. Each message exchanged between two processes consumes $t_s + t_t + t_r$ time units (t.u.): t_s t.u. are taken to send the message at the source, t_t t.u. are taken to transmit the message across the network, and t_r t.u. are taken to receive the message at the destination. If a message is sent to multiple destinations, the sender computes t_s to each copy sent.

For all executions, simulation parameters were set as follows: time to send/receive a message $t_s = t_r = 0.1$; time to transmit a message $t_t = 0.8$. In this case, we assume a scenario in which it takes longer for a message to be transmitted than it takes to be processed at each local node.

For the sake of improving of the evaluation coverage, we considered different scenarios and system sizes. The system parameters were set as follows. For each experiment, the number of processes n ranges from 8 to 1024 in a power of 2.

VCube-RB and All-RB use the same hierarchical monitoring strategy of VCube, and, therefore, the impact of failure detection is the same in the two solutions. The testing interval was set 30.0 t.u. A process is considered to be suspected if it does not respond to the test after $4 * (t_s + t_r + t_t)$ time units, that is, 4.0 t.u.

Experiments without faulty processes

Without loss of generality, a single broadcast message is sent by process 0 (p_0). Figure 3 presents the results obtained to fault-free scenarios.

In the All-RB strategy, latency is lower for systems up to 128 processes (compared to VCube-RB) and 64 processes (compared to DPath-RB), but increases faster after this threshold because of the processing delay for sending the copies of the TREE message to each of the $n-1$ correct processes in the system (remember that each sent message consumes $t_s = 0.1$ time units). On the other hand, the sender in VCube-RB sends only $\log_2 n$ messages, one for each neighbor, that are forward in parallel in the other levels of the tree. DPath-RB also uses a tree over the hypercube topology, but it is slightly faster than VCube-RB since it does not wait for acknowledgements. These results confirm the scalability of the proposed hierarchical strategy.

The number of messages per broadcast is the same for VCube-RB and All-RB, since $2(n-1)$ messages are sent by both algorithms ($n-1$ TREEs and $n-1$ ACKs). The difference is that in All-RB all TREE messages of

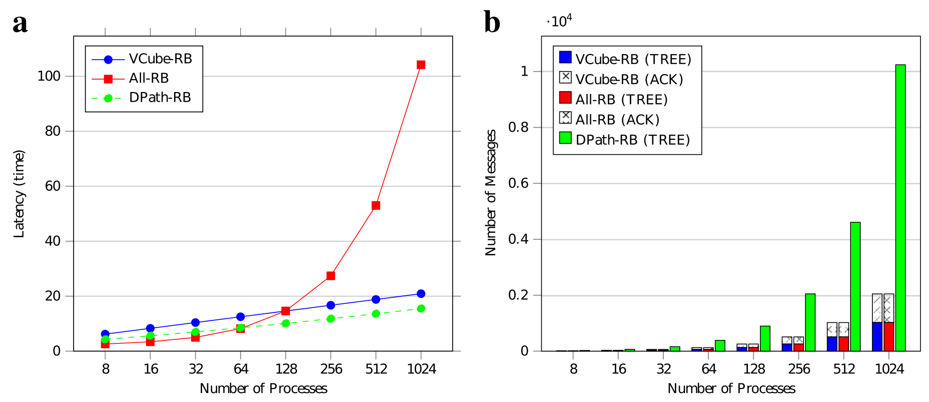


Fig. 3 Latency to complete a broadcast in a fault-free execution. **a** Latency. **b** Number of messages

the broadcast are sent by the same sender, while in the VCube-RB, the sender sends TREE messages to its neighbors that forward it to their neighbors and so on along the tree. DPath-RB sends much more messages, because each message is retransmitted by $log_2 n$ disjoint paths, and each process receives $log_2 n$ copies of the same message.

Experiments with faulty processes

We compared the performance of the three approaches when processes are faulty. The first experiment considers the failure of an intermediate process in the subtree of Vcube-RB. The second considers the failure of the sender process. Last, we perform tests with random fault processes. For the sake of simplicity, a single process (p_0) performs the broadcast.

Scenario with one faulty node in the subtree

We simulated one single crash of the process whose identifier is $n/2$. This process is the first process of the biggest cluster of process 0 in the VCube topology. In a VCube with eight processes, for example, when p_4 fails, extra messages can be retransmitted to p_5, p_6, and p_7, depending on the time the failure occurs. For All-RB and DPath-RB, the faulty process is the same, even if the scenario would be equivalent to any faulty process. For DPath-RB, a faulty process mans no retransmission along the paths that contains the process.

The failures of process $p_{n/2}$ were simulated at two different time instants. In the first scenario, the process crashes at time $t_0 = 0.0$. In this case, no extra message is generated, since the faulty process does not receive the TREE message and, in case of VCube-RB, upon detection of the failure, a new message will be retransmitted to the next correct process in the cluster, which will retransmit in the subtree only once. In the second experiment, the crash is set at time $t_1 = \log_2 n$. In the case of VCube-RB, this time is long enough to the process to receive and forward the TREE message before it fails. Thus, when the failure is detected, a new message will be sent to the next fault-free process in the same cluster that will forward it through the new subtree. Each correct process in that cluster will receive the message twice (although duplicated delivery is avoided by the timestamps). Note that the testing interval was set to 30.0 t.u., and the crash happens just after the first testing round. In this case, the latter is detected only in the second testing round. After that, the tree is rebuilt and the broadcast is completed. Such a behavior of the algorithm explains the performance of the experiments with t_1.

Figure 4a, b shows the latency and number of messages generated in the above two scenarios when the intermediate process of the tree, $p_{n/2}$, fails. We can observe that latency is higher in relation to the fault-free scenario due to the fault detection latency. Remember that the timeout was set to 4.0 t.u. In the case of All-RB, after the detection of the failure by the source process, the broadcast is immediately considered completed. Note that in order to ensure the correctness of the solution in case of mistakes, messages are also sent to processes considered to be faulty. In the case of VCube-RB, once the failure is detected, the tree is automatically reconfigured and the broadcast restarts in the cluster of the faulty process. In these scenarios, besides sending additional messages (Fig. 4d), there is a higher latency in VCube-RB in small systems, till close to 128. However, as the number of processes increases, the latency of All-RB is again jeopardized by the multiple copies of the message sent from the source process. In Fig. 4b with 128 processes, it is possible to note that All-RB presents a higher latency because the crash was detected only in the second round of the failure detector, i.e., 30 t.u. after the first round. For DPath-RB, one faulty process has no effect in terms of latency, and

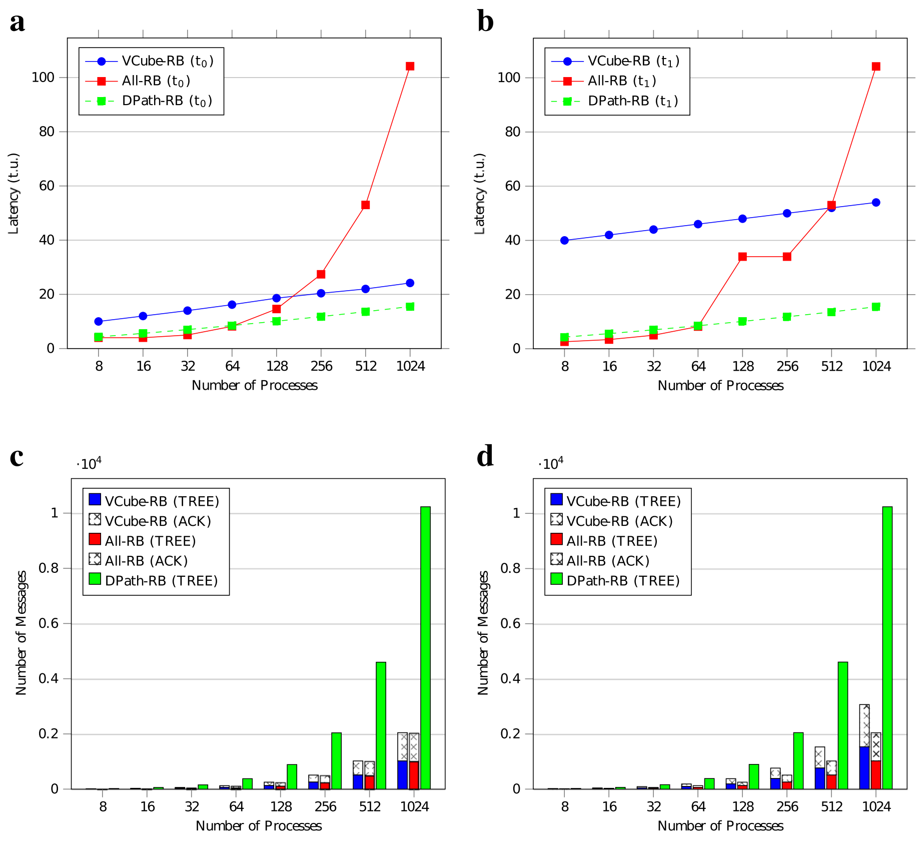


Fig. 4 Latency and total number of messages to scenarios with crash of the intermediate node $p_{n/2}$ at times $t_0 = 0.0$ and $t_1 = \log_2 n$. **a** Latency to t_0. **b** Latency to t_1. **c** Number of messages to t_0. **d** Number of messages to t_1

the number of messages can be slightly reduced if the process crashes before retransmiting the messages to the neighbors.

Failure of the sender process (source)

Considering the worst case in which the sender, process p_0, starts the broadcast and crashes after it has sent a message to each neighbor. This means that, in case of VCube-RB, all $\log_2 n$ neighbors of p_0 in the hypercube will receive a TREE message and forward it along the tree. In case of All-RB, all $n - 1$ copies of the TREE will be sent before the source crashes. In both algorithms, all correct processes will receive the TREE message. As we have explained, VCube monitoring system will eventually notify every correct process about the failure of p_0 and a new broadcast will be restarted at each correct process. Although they happen at different times for each algorithm, the source crash is simulated in the first round of VCube, not interfering with the detection time. For the DPath-RB implementation, considering the restriction of the number of faulty processes and the multiple disjoint paths, no re-broadcast is applied.

Figure 5 presents latency results considering different system sizes. The difference in latencies is due to the time taken for each process to propagate the TREE message to other members after being notified by the VCube failure detector about p_0's crash. In the All-RB strategy, each process sends a new copy of the last message received from p_0 directly to all other process of the system and waits for the respective acknowledgements. In the VCube-RB solution, copies of the last message are also sent by each process of the system, but using the tree rooted at each process.

Figure 5a compares the latency of the three solutions. Compared to All-RB, Vcube-RB has a lower latency as the number of processes grows; in relation to the total number of messages, it can be seen in Fig. 5b and Table 1 that VCube-RB uses a much smaller number of messages. Such a difference is the result of the retransmission mechanism in the tree, which prevents two equal messages from being propagated in the same subtree if it has already been forwarded and the corresponding ack is pending. The graph shows an imbalance between the number of TREE messages related to the ACK messages which can be explained since, in scenarios with failures,

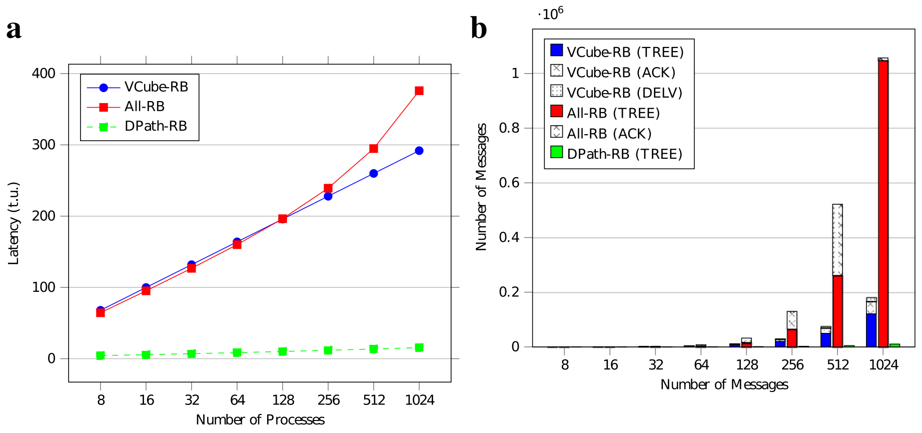


Fig. 5 Latency and total number of messages when sender process (source) crashes. **a** Latency. **b** Number of messages

when a process receives a TREE message from a faulty process (source or intermediary) it does not send an ACK message to the latter. In the case of the VCube-RB strategy, for example, the propagation of ACK messages to the source process, which is faulty, is canceled as soon as the failure is detected by a process in the reverse path of the tree. In addition, VCube-RB does not retransmit messages that have already been propagated in the tree. This considerably reduces the total number of retransmitted messages after failures compared to the All-RB strategy without this control. For DPath-RB, latency and number of messages are similar to the previous faulty scenarios. It is possible to conclude that DPath-RB has the best performance with single crash, but the restriction on the number of failures, as in DPath-RB, remains a limiting factor in systems subject to multiple failures.

Scenarios with random faulty nodes

In this subsection we evaluate the three approaches under scenarios with random faulty processes from 1 to $(log_2 n) - 1$ (process p_0 is the sender and never fails). The number of faulty processes were defined to $(log_2 n) - 1$, since this is the maximum number of faults tolerated by DPath-RB. Considering $d = log_2 n$ the dimension of the hypercube, for each scenario with different number of processes, we executed 32 experiments, each one with $d - 1$ random faulty processes at random time. With eight processes ($d = 3$), for example, two processes fail during each test. The time instant of the crashes were generated using a random integer between zero and d, which is the time that fits all latencies in fault-free executions. Results in Fig. 6a shows the mean latency with a 95% confidence interval using standard normal z-table.

The results confirm the scalability of VCube-RB compared to All-RB, and the efficiency in terms of messages compared to both All-RB and DPath-RB.

Conclusion

This paper presented a reliable broadcast algorithm for message-passing asynchronous distributed systems prone to crash failures. It tolerates up to $n - 1$ failures. For broadcasting a message, the algorithm dynamically builds a spanning tree over a virtual hypercube-like topology provided by an underlying monitoring system called VCube. In case of failure, the tree is dynamically reconstructed without any message overhead. To this end, the VCube provides information about node failures. However, as the system is asynchronous, it can make mistakes falsely suspecting fault-free nodes. Such false suspicions are tolerated by the algorithm by sending special messages to those processes suspected of having failed. In summary, whenever possible, the algorithm exploits the hypercube properties offered by the VCube while ensuring the properties of the reliable broadcast, even in case of false suspicions.

Besides the formal proof of the algorithm, simulation results comparing the proposed solution with two other approaches show the scalability and efficiency of the algorithm in fault-free and faulty scenarios, especially for systems with more than 128 processes. The calculation of

Table 1 Total number of messages when sender process (source) crashes

Processes	VCube-RB	All-RB	DPath-RB
8	120	96	24
16	491	442	64
32	1589	1899	160
64	4582	7884	384
128	12242	32141	896
256	31104	129807	2048
512	76153	521741	4608
1024	181790	2092009	10240

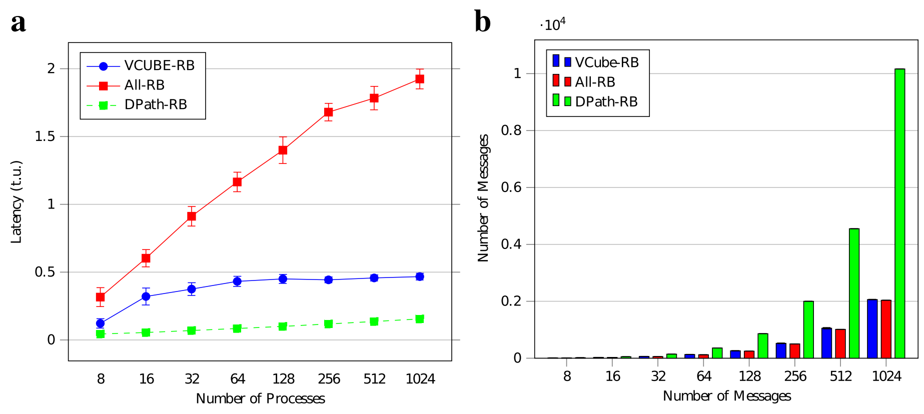


Fig. 6 Latency and total number of messages with multiple random faults. **a** Latency. **b** Number of messages

the tree on demand in each process and without the need to exchange messages, added to the control of messages already transmitted in that tree, decreases the latency and the total number of messages.

Future work includes developing causal and atomic broadcast algorithms based on VCube for asynchronous environments.

Endnotes

[1] A correct process is a process that does not fail during execution.

[2] Note that this is not a restriction. To deal with an arbitrary n, it is enough to avoid communications with processes that have identifiers greater than or equal to n.

Acknowledgements
We would like to thank the funding agencies and universities involved for the support provided. We also thank the many contributions from the reviewers.

Funding
This work has been partially supported by grants 45112/2015-144 from the Parana State Research Agency (Fundação Araucária/SETI)/CNRS, and by the Brazilian Research Agency (CNPq) under grants 311451/2016-0.

Availability of data and materials
Not applicable.

Authors' contributions
ÉJ is Ph.D. student at UMPC/Lip6 and has developed the reliable broadcast algorithm to asynchronous systems as part of her thesis. LAR has proposed the previous version to the algorithm to synchronous systems during his Ph.D. under the guidance of professors EPDJr. and LA, and conducted the implementation and experiments in this work. LA is Jeanneau's advisor and has written mainly the description and proofs to the algorithm. EPD Jr. has participated in all discussions and in the written phase of the final text. All authors read and approved the final manuscript.

Ethics approval and consent to participate
Not applicable.

Consent for publication
All authors agree to the submitted version.

Competing interests
The authors declare that they have no competing interests.

Publisher's Note

Author details
[1]Sorbonne Universités, UPMC Univ. Paris 06, CNRS, Inria, LIP6, 4 place Jussieu, 75252 PARIS CEDEX 05, Paris, France. [2]Department of Computer Science, Western Paraná State University, Rua Universitária, 2069, Jardim Universitário, CEP 85.814-110 Cascavel-PR, Brazil. [3]Department of Informatics, Federal University of Paraná, R. Cel. Francisco H. dos Santos, 100 Centro Politécnico, Caixa Postal: 19081, 81531-980 Curitiba, Brazil.

Received: 26 May 2017 Accepted: 22 November 2017
Published online: 20 December 2017

References
1. Leitão J, Pereira J, Rodrigues L (2007) HyParView: a membership protocol for reliable gossip-based broadcast. In: DSN. IEEE, Edinburgh. pp 419–429. doi:10.1109/DSN.2007.56
2. Yang Z, Li M, Lou W (2009) R-code: network coding based reliable broadcast in wireless mesh networks with unreliable links. In: GLOBECOM'09. IEEE, Honolulu. pp 1–6. doi:10.1109/GLOCOM.2009.5426175
3. Bonomi S, Del Pozzo A, Baldoni R (2013) Intrusion-tolerant reliable broadcast. Technical report
4. Hadzilacos V, Toueg S (1993) Fault-tolerant broadcasts and related problems:97–145. Chap. Distributed systems
5. Guerraoui R, Rodrigues L (eds) (2006) Introduction to reliable distributed programming. Springer, Berlin, Germany
6. Chandra TD, Toueg S (1996) Unreliable failure detectors for reliable distributed systems. J ACM 43(2):225–267. doi:10.1145/226643.226647
7. Schneider FB, Gries D, Schlichting RD (1984) Fault-tolerant broadcasts. Sci Comput Program 4(1):1–15
8. Fragopoulou P, Akl SG (1996) Edge-disjoint spanning trees on the star network with applications to fault tolerance. IEEE Trans Comput 45(2):174–185. doi:10.1109/12.485370
9. Kim K, Mehrotra S, Venkatasubramanian N (2010) FaReCast: Fast, reliable application layer multicast for flash dissemination. In: ACM/IFIP/USENIX 11th International Conference on Middleware. Middleware'10. Springer, Berlin, Heidelberg. pp 169–190
10. Ramanathan P, Shin KG (1988) Reliable broadcast in hypercube multicomputers. IEEE Trans Comput 37(12):1654–1657. doi:10.1109/12.9743
11. Raynal M, Stainer J, Cao J, Wu W (2014) A simple broadcast algorithm for recurrent dynamic systems. In: Proceedings of the 2014 IEEE 28th International Conference on Advanced Information Networking and Applications. AINA '14. IEEE, Victoria. pp 933–939. doi:10.1109/AINA.2014.115
12. Kephart JO, Chess DM (2003) The vision of autonomic computing. Computer 36(1):41–50

13. Duarte Jr EP, Bona LCE, Ruoso VK (2014) VCube: a provably scalable distributed diagnosis algorithm. In: 5th Work. on Latest Advances in Scalable Algorithms for Large-Scale Systems. ScalA'14. IEEE Press, Piscataway. pp 17–22. doi:10.1109/ScalA.2014.14, http://dx.doi.org/10.1109/ScalA.2014.14
14. Rodrigues LA, Arantes L, Duarte Jr EP (2014) An autonomic implementation of reliable broadcast based on dynamic spanning trees. In: 10th European Dependable Computing Conference. EDCC'14. IEEE, Newcastle. pp 1–12. doi:10.1109/EDCC.2014.31
15. Wu J (1996) Optimal broadcasting in hypercubes with link faults using limited global information. J Syst Archit 42(5):367–380
16. Liebeherr J, Beam TK (1999) HyperCast: a protocol for maintaining multicast Group Members in a Logical Hypercube Topology(Rizzo L, Fdida S, eds.). Springer, Berlin, Heidelberg. doi:10.1007/978-3-540-46703-8_5. http://dx.doi.org/10.1007/978-3-540-46703-8_5
17. Ferreira M, Leitão J, Rodrigues L (2010) Thicket: a protocol for building and maintaining multiple trees in a p2p overlay. In: Proceedings of the 29th IEEE International Symposium on Reliable Distributed Systems. IEEE, New Delhi. pp 293–302. doi:10.1109/SRDS.2010.19
18. Urbán P, Défago X, Schiper A (2002) Neko: a single environment to simulate and prototype distributed algorithms. J Inf Sci Eng 18(6):981–997
19. Bulgannawar S, Vaidya NH (1995) A distributed k-mutual exclusion algorithm. In: Proc. of the 15th Int'l Conf. on Distr. Comp. Systems. IEEE Computer Society, Los Alamitos. pp 153–160. doi:10.1109/ICDCS.1995.500014