J. Parallel Distrib. Comput. 115 (2018) 41–55

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

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

A distributed *k*-mutual exclusion algorithm based on autonomic spanning trees

Luiz A. Rodrigues [a,*], Elias P. Duarte Jr. [b], Luciana Arantes [c]

[a] *Western Paraná State University, Brazil*
[b] *Federal University of Paraná, Brazil*
[c] *Sorbonne Universités, UPMC Univ Paris 06, CNRS, Inria, LIP6, France*

HIGHLIGHTS

- An autonomic distributed *k*-mutual exclusion that tolerates $n - 1$ fault processes.
- A self-adapting spanning tree that covers the correct processes in the system.
- A hierarchical best-effort broadcast based on the spanning tree approach.

ARTICLE INFO

Article history:
Received 23 February 2017
Received in revised form 26 July 2017
Accepted 22 January 2018
Available online 3 February 2018

Keywords:
Mutual exclusion
Distributed applications
Fault-tolerance
Reliable broadcast
Hypercube-like topology

ABSTRACT

Distributed *k*-mutual exclusion ensures that at most a single process has permission to access each of the *k* copies of a critical resource. In this work we present an autonomic solution for distributed *k*-mutual exclusion that adapts itself after system changes. Our solution employs a hierarchical best-effort broadcast algorithm to propagate messages reliably and efficiently. The broadcast is based on another autonomic building block: a distributed algorithm for creating and maintaining spanning trees constructed in a fully distributed and adaptive way on top of a virtual hypercube-like topology, called VCube. The proposed solutions are autonomic in the sense that they reconfigure themselves automatically after the detection of faults given the set of correct processes in the system. All proposed algorithms are described, specified, and proofs of correctness are given. Results from simulation show that the proposed approach is more efficient and scalable compared to other solutions.

1. Introduction

Distributed systems allow a potentially large number of users to share multiple resources. Although some resources can be accessed by an arbitrary number of users at any time, there are resources for which it is necessary to guarantee exclusive access: those resources can be accessed at most by a single user at a given time instant (e.g. concurrent updates of shared data, which occurs both in replicated databases and in distributed shared memory systems, such as Linda [3] and Midway [6], among others). *Distributed mutual exclusion* [21,31,33] is employed to guarantee such exclusive access. Given a user process, the segment of code that accesses the shared resource is called the critical section (CS). Distributed mutual exclusion algorithms must ensure two properties [20]: *safety*, by which at most one process can access the resource per time instant, i.e., only a single process can be in the CS at a given time instant and *liveness*, by which all processes that request access to a resource eventually succeed.

There are basically two approaches to implement mutual exclusion in distributed systems: *permission-based* and *token-based* [30,38,20]. Permission-based mutual exclusion algorithms rely on the principle that a node (process) only enters a critical section after having received permission from all the other nodes (processes) (or a majority of them) [33,37]. The other family of algorithms is token-based, i.e. a system-wide unique token is shared among all processes, and its possession gives the process the exclusive right to execute the CS and thus access the critical resource [39,29,26,7].

An extension of the mutual exclusion problem is the *k*-mutual exclusion problem, which deals with *k* copies of the shared resources, i.e., there are *k* resource units, but a single unit can be accessed by at most one process at a given time instant. Thus, as there are *k* units, at most *k* processes can access these units simultaneously, with one process per unit. Therefore, a *k*-mutual exclusion algorithm must guarantee that at most *k* processes can be in the critical section at any time (*safety* property) and that every

* Correspondence to: Dept. Informatics, P.O. Box 719, 85819-110 Cascavel-PR, Brazil.
E-mail address: luiz.rodrigues@unioeste.br (L.A. Rodrigues).
URL: http://www.inf.unioeste.br/∼luiz (L.A. Rodrigues).

https://doi.org/10.1016/j.jpdc.2018.01.008
0743-7315/

request to enter the critical section execution is eventually satisfied (*liveness* property).

Raymond's algorithm [28] solves the k-mutual exclusion problem by extending the permission-based solution proposed by Ricart and Agrawala [33]. Whenever a process wants to use a unit of the shared resource, it sends a request message to the $n-1$ other processes and waits for at least $n-k$ replies. Note that if no other process is using or requesting the resource, the total number of responses is $n-1$. Therefore, in the worst case $2(n-1)$ messages are generated by each request. This means that there is no reduction in the number of messages of k-mutual exclusion compared to 1-mutual exclusion, but the process of acquiring the required permissions can be accelerated.

Distributed mutual exclusion algorithms should be resilient to failures. In permission-based mutual exclusion, a requesting process needs to be informed about which processes have crashed in order to avoid waiting indefinitely for permission from these processes [8]. Similarly, in token-based algorithms, if the process that holds the token fails, the failure must be detected and a new token must be generated. In both cases, a solution would be to use a monitoring mechanism that offers information about the state (correct or faulty) of all processes in the system [36]. It is worth pointing out that [11] proves the weakest failure detector to solve fault-tolerant distributed mutual exclusion, given a majority of correct processes, is the *trusting* $\mathcal{T}$ where, once a process is detected as alive, it will not be suspected of being faulty until it crashes. We should also point out that even if Raymond's algorithm [28] does not explicitly consider node failures, the fact that it does not need to wait for replies from all the other nodes (only $n-k$) implicitly renders it fault-tolerant to some extent. It tolerates up to $k-1$ faults. Thus, up to $k-1$ crashes, a node asking a unit of resource still gets it, but the effectiveness of the algorithm is reduced since the number of processes which can concurrently access the resource units decreases by one after each crash.

Besides failures, another important feature that has a direct impact on the scalability and performance of distributed mutual exclusion algorithms is the message dissemination mechanism used by the algorithm. This is particularly critical for permission-based algorithms in which processes broadcast their requests to all other processes. If the algorithm runs on a network that does not offer physical broadcast as service, the broadcast primitive called by processes can generate multiple copies of the request message that are sent to all other processes. Unfortunately, this solution is not efficient. If the number of processes is large, the sender will receive a large number of acknowledgments, which can possibly cause the occurrence of a *feedback implosion* problem at the sender [40]. To overcome this problem is one of the objectives of this work. We propose an efficient and scalable message dissemination mechanism based on a spanning tree [4] built on a logical hypercube that presents logarithmic message complexity.

The contributions of this work are threefold:

- By extending Raymond's algorithm [28], we introduce an autonomic distributed permission-based k-mutual exclusion algorithm which tolerates up to $n-1$ process crashes. Aiming at scalability and good performance, the algorithm exploits the underlying VCube monitoring mechanism [12]. In VCube, processes monitor each other and dynamically organize themselves on a virtual hypercube-like topology that keeps its logarithmic properties even in the presence of failures. Processes broadcast requests over adaptive and fully distributed spanning trees built on VCube. Our choice for a permission-based algorithm can be justified because, although token-based algorithms usually present good performance with respect to the number of messages, they suffer from poor resiliency while, due to redundancy of messages, the majority of permission-based algorithms inherently either tolerate failures or can be adapted to tolerate failures more easily.
- The distributed algorithm that dynamically constructs the above mentioned minimum spanning trees is also specified in this work. It is based on VCube and builds a spanning tree from any source process (which is the spanning tree root). A spanning tree built with the proposed algorithm contains all processes that are considered to be correct, according to the monitoring tests executed by VCube. The algorithm is fully distributed and autonomic in the sense that trees are built and maintained autonomically, despite changes in the system composition. These spanning trees are used to implement the broadcast primitive, providing scalability to the solution.
- Finally, we also present a best-effort broadcast algorithm responsible for disseminating messages to all processes of the system. Best-effort broadcast ensures that, if the sender does not fail, all fault-free processes will deliver the message from that sender. Such a broadcast is employed by the mutual exclusion algorithm to issue requests to access the resource to all other processes. The proposed best-effort broadcast algorithm employs the distributed minimum spanning tree algorithm, above described.

The three algorithms are formally specified and proofs of correctness are presented. Furthermore, the solution is experimentally evaluated. Simulation results confirm that our k-mutual exclusion algorithm executes efficiently in both fault-free and faulty scenarios presenting better performance when compare to two other permission-based k-mutual exclusion of the literature.

The rest of this paper is organized as follows. Section 2 summarizes related work. In Section 3 the system model is specified while in Section 4 VCube is described. The proposed spanning tree algorithm is in Section 5. Sections 6 and 7 present the best-effort broadcast and k-mutual exclusion algorithms, respectively. Experimental results obtained with simulation are reported in Section 8. Section 9 concludes the paper.

2. Related work

The first solution for distributed permission-based mutual exclusion was proposed by Lamport [21]. This algorithm uses three types of messages: REQUEST, REPLY and RELEASE. It also uses logical clocks to order the requests. When a process p_i wants to access the resource, it broadcasts a REQUEST message to all other processes and stores its request in a local queue. A process p_j, which receives the REQUEST message from p_i, enqueues the request locally and sends a timestamped REPLY message back to p_i. Process p_i can access the resource when two conditions are satisfied. First, it must have received REPLY messages with timestamps greater than that of its own request from all other processes. Second, the process's own request must be at the front of its queue. When p_i releases the resource, it broadcasts a RELEASE message.

Lamport's algorithm employs $3(n-1)$ messages, as $n-1$ messages are employed at each step: request, reply and release. Ricart and Agrawala [33] proposed an algorithm that requires $2(n-1)$ messages. This algorithm uses REQUEST and REPLY messages as Lamport's algorithm does, but does not employ RELEASE messages. This is possible because each process does not send REPLY messages while it is using the resource or if it is trying itself to use the resource and its request has higher priority. The REPLY messages are deferred and sent only when the process releases the resource. Thus a process only enters the critical section when it has received REPLY messages from all other processes.

Naimi [25] proposed two mutual exclusion algorithms tailored for hypercubes. The first solution is permission-based and requires

$d2^d$ messages, where d is the dimension of the hypercube. The second algorithm is token-based and requires $2d$ messages in the worst case. Different to our solution, the spanning tree is built using flooding, and no faults are considered.

In [28] is presented a token-based algorithm that employs a spanning tree for nodes to communicate. Th average number of messages is $O(\log_2 n)$, but in the worst case up to $2(n-1)$ messages are required. These mutual exclusion algorithms aim at minimizing the number of messages, and do not tolerate faulty processes.

The algorithm proposed by Agrawal and El Abbadi [1] uses the concept of *coterie* introduced by Garcia-Molina and Barbara [15] and hierarchical quorums to solve mutual exclusion in a scalable way. Another solution [2] presents a similar algorithm, but that includes fault tolerance. A quorum-based solution to k-mutual exclusion was proposed by [19] using the concept of k-coterie. To circumvent deadlocks, if any process in a quorum is blocked to another requester, the request is sent to other quorum. Park and Lee [27] present a fault tolerant quorum-based mutual exclusion algorithm that also uses a failure detector. Although they use a similar topology, the solutions are based on tokens.

Another token-based solution was proposed by [32]. It uses a dynamic tree based on reverse path approach, i.e., the tree is generated considering the source instead the destination. Faults are probabilistically tolerated using replication on successors and predecessors in the path.

Bouillaguet, Arantes, and Sens [8] adapted the permission-based approach proposed by [28] to tolerate up to $n-1$ faulty processes using the perfect failure detector $\mathcal{T}$. Later, the same authors proposed in [9] a solution that avoids the use of failure detectors and extra messages to detect faulty processes by including information about processes state into the messages of the mutual exclusion algorithm themselves and introducing a communication behavior property based on winning channels. The solution tolerates up to $k-1$ failures.

In a previous work [35] we presented a preliminary version of our k-mutual exclusion solution using spanning trees algorithm to propagate messages and a previous different version of the virtual hypercube-like topology used in this paper.

3. System model and definitions

We assume a distributed system consisting of a finite set of nodes named $P = \{p_0, .., p_{n-1}\}$, where $n > 1$. The set of participants are uniquely identified and known by all nodes. There is one process per node, hence we use the terms node and process interchangeably. Every pair of nodes is assumed to be connected by means of a bi-directional reliable communication channel (no loss, corruption or duplication of messages) and processes communicate by sending and receiving messages.

Processes can fail by crashing and a crash is permanent. A process is *correct* if it does not crash during a run, otherwise, it is *faulty*. The system is considered to be synchronous, i.e., there are known bounds on both message delays and process speed. Therefore, by correcting calibrating timeouts, false failure suspicions are avoided. We should remember that the weakest failure detector to solve fault tolerant distributed mutual exclusion on message-passing system is the $\mathcal{T}$ failure detector [11]. In this failure detector once a process is known by another, it is never suspected before it actually crashes. In other words, a fault-tolerant distributed k-mutual exclusion algorithm can only be implemented in a synchronous system.

There are k identical units of the shared resource. k is known in advance by all processes. We consider that processes behave correctly. A process requests only one resource unit at each time, i.e., a process cannot issue a new request while it is still using a resource unit. It calls function REQUEST to ask access to a unit of the resource, and function RELEASE, when it releases the resource unit. Whenever a process obtains a unit of the resource, it keeps it during a bounded interval, i.e., the duration of every critical section (CS) is bounded.

Processes are logically organized on an adaptive virtual topology which is based on a d-dimensional hypercube (d-cube) maintained by the distributed diagnosis algorithm VCube [12], described in the next section. A complete d-VCube consists of $n = 2^d$ processes. Each process i has a unique identifier from 0 to $n-1$ represented as a d-bit binary address $i_{d-1}, i_{d-2}, \ldots, i_0$. In a fault-free scenario, two processes are connected if their binary addresses differ by only one bit. Thus, each process is connected directly to $d = \log_2 n$ neighbors and the maximum distance between any two processes is $\log_2 n$. In faulty scenarios, correct processes can be reconnected to remove faulty processes in such way that the logarithmic properties are kept.

4. VCube: the virtual hypercube topology

VCube [12] is a logical topology that can be used to virtually interconnect network processes. Processes can leave the system as they become faulty (crash). When all processes are fault-free, the topology is a perfect hypercube of $d = \log_2 n$ dimensions and 2^d processes, named d-VCube. Such a topology presents important logarithmic properties. For example, the number of links from one process to other processes is $\log_2 n$ and the maximum distance between any two processes is $\log_2 n$. Upon the detection of processes failure, the remaining fault-free processes reorganize themselves in order to keep the logarithmic properties.

Nodes maintain the topology by running a distributed diagnosis algorithm [23], which offer fault-tolerant monitoring based on the execution of tests and distribution of test results. VCube is based on the Hi-ADSD algorithm proposed by [13], where fault-free processes of a system of n processes learn about new events of the system in at most $\log_2^2 n$ testing rounds. However, VCube presents even better performance, as it uses a new testing strategy that guarantees that the maximum number of tests executed per $\log_2 n$ testing rounds is $n\log_2 n$ and each fault-free process gets as much new information as possible at each test round. This is one of the main advantages of using VCube: after a network change, it takes a logarithmic number of testing rounds and a logarithmic number of messages (tests) for all nodes to update the information locally, including the routing table.

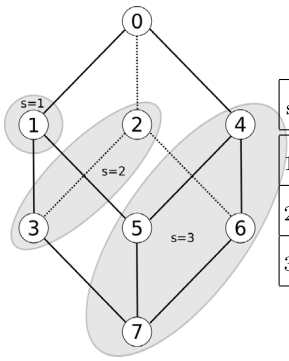
The principle of the hierarchical testing strategy is to organize the processes in progressively larger clusters. The list of ordered processes tested by process i in a cluster of size 2^{s-1}, is denoted by $c_{i,s}$, $s = 1..\log_2 n$. In the first testing interval, the tested cluster for process i is $c_{i,1}$; in the next testing interval, the tested cluster is $c_{i,2}$, and so on until the tested cluster is $c_{i,log_2 n}$. Then, in the next interval the tested cluster is again $c_{i,1}$ and the same testing process continues. When a fault-free process is tested, the tester obtains diagnostic information from the tested process.

A compact expression for computing $c_{i,s}$, $i = 0..n-1$, in which $\oplus$ corresponds to the exclusive or operation is:

$$c_{i,s} = i \oplus 2^{s-1} \parallel c_{i\oplus 2^{s-1},k} \mid k = 1, 2, \ldots, s-1. \quad (1)$$

Basically, in the above function, the hypercube neighbor of node i in cluster s is computed firstly. Note that the identifiers of the two neighbor nodes differ only by one bit: the bit which is equal to 1 in 2^{s-1}. Then, the remaining nodes in the cluster s are computed: they are the nodes in cluster $1, 2, \ldots, s-1$ of the neighbor, i.e., $c_{i\oplus 2^{s-1},1}, c_{i\oplus 2^{s-1},2} \ldots, c_{i\oplus 2^{s-1},s-1}$. The table of Fig. 1 shows the result of $c_{i,s}$ applied to a system with 8 nodes.

Process i tests another process in the $c_{i,s}$ to check whether it is correct or faulty. The tester executes a test procedure and waits for a reply. If the correct reply is received within an expected time



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

Fig. 1. Hierarchical cluster organization by 3-VCube and the respective $c_{i,s}$ table.

interval and is correct, the monitored process is considered to be correct. Otherwise, the tested process is considered to be faulty. Considering the results of the tests, each process i is connected to one correct process in each cluster s, if it exists. If there are no faults, a complete hypercube is created.

Fig. 1 (left) shows the tests executed by process p_0 on a 3-VCube; clusters $c_{0,1}$, $c_{0,2}$, and $c_{0,3}$ are also shown. Considering that there are no faults, p_0 performs tests on processes $\{1, 2, 4\}$ which will then give information about the local state of all other processes in the system.

The VCube testing strategy is based on the following rule: before process i executes a test on process $j \in c_{i,s}$, it checks whether it is the first fault-free process in $c_{j,s}$ (please note that the index is j). After process i tests process $j \in c_{i,s}$ as faulty-free, it obtains new diagnostic information from tested process j. This information is not restricted to the cluster: the tester may obtain *any* new information that the tested process has—in comparison with its own local information. As a tester may obtain information about a given process from different tested processes, the algorithm employs timestamps to determine which information is newer. Initially every process is assumed to be fault-free and the corresponding timestamp is zero. As an event is detected, i.e., a process failure, the corresponding timestamp is incremented. Thus, an even timestamp corresponds to a fault-free process and an odd timestamp to a faulty process.

Node i keeps timestamp array $t_i[0..N-1]$. After obtaining information from process j tested as fault-free, process i only updates a local timestamp entry when the obtained value is greater than the current one. An advantage of such a strategy is that tester nodes can obtain new diagnostic information about every system process from any tested process. In order to avoid transferring the complete t_j array as process i tests process j as fault-free, it is possible to implement a simple solution in which process j only sends new information, i.e., information that has changed since the last time process j was tested by process i.

5. The spanning tree algorithm

Let $G = (V, E)$ be an undirected graph, where V is the set of vertices that correspond to the system's nodes, and E the set of edges, each of which represents a communication channel connecting pair of nodes. We assume that G consists of a single connected component. A *spanning tree* of G is an acyclic connected graph that contains all vertices of G [14]. If the edges are weighted, a *minimum spanning tree* is the tree for which the sum of all edge weights is the minimum for all possible trees. If G is unweighted or if all edges have the same weight, all trees of G are minimum.

A spanning tree is an efficient mechanism for nodes of a distributed system to disseminate information [5]. For instance, in a naive implementation of a broadcast algorithm every node (vertex) sends the message to all neighbors, thus messages are sent through $|E|$ links. Using a spanning tree, the messages need to be sent only to the neighbors in the tree, and thus $n-1$ messages are sent [16]. Whenever $|E| > n-1$, i.e., G is not a sparse graph, the spanning tree reduces the number of messages employed by the broadcast algorithm.

In this section we describe and specify a distributed spanning tree algorithm which is embedded on a VCube. Any process can be the root of the tree. Trees are dynamically built using the strategy described in Section 4. In particular, function $c_{i,s}$ is employed to determine the hierarchical clusters and the set of edges available.

5.1. Variables and functions

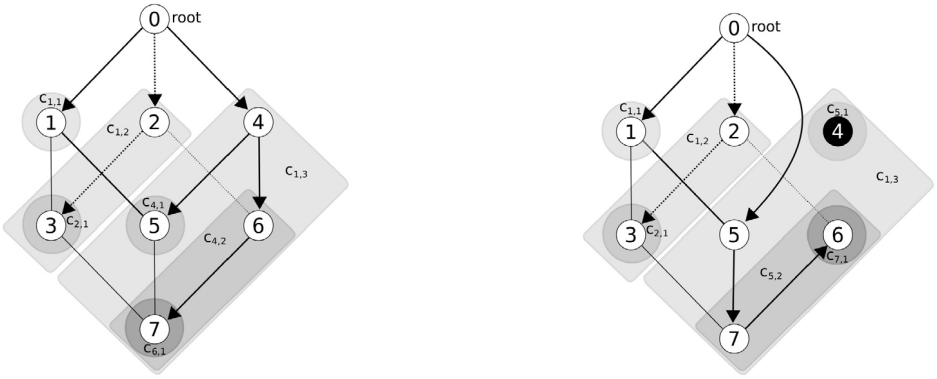
Let $correct_i$ be the set of nodes that node i considers to be correct, based on the tests executed and information received by node i through the VCube. Note that a given node j might be faulty but node i has not yet received information about the fault due to VCube's latency.

Algorithm 5.1 Spanning tree algorithm at process i

```
1:  correct_i ← {0, .., n − 1}        ▷ list of processes considered correct by i according VCube
2:  procedure StartTree( )
3:      for all k ∈ neighborhood_i(log_2 n) do        ▷ send the message to all neighbors of i
4:          Send(⟨TREE⟩) to p_k
5:  upon receive ⟨TREE⟩ from p_j
6:      if j ∈ correct_i then
7:          for k ∈ neighborhood_i(cluster_i(j) − 1) do        ▷ retransmit to neighbors in subclusters of i
8:              send(⟨TREE⟩) to p_k
9:  upon notifying crash(j)        ▷ j is detected as faulty by VCube at i
10:     correct_i ← correct_i ∖ {j}
11:     if ∃k = FF_neighbor_i(cluster_i(j)) then
12:         send(⟨TREE⟩) to p_k
```

The following functions are defined:

- $cluster_i(j) = s$: 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, $cluster_i(j) = cluster_j(i)$.
- $FF\_neighbor_i(s) = j$ returns the first node j in $c_{i,s}$ which also belongs to $correct_i$. If there is no such node, the function returns $\perp$ (no neighbor). For example, in the 3-VCube of Fig. 1



(a) Fault-free execution.

(b) Process 4 is faulty.

Fig. 2. Spanning trees in the 3-VCube rooted at process 0 (p_0).

in a scenario with no faulty nodes, $FF\_neighbor_4(1) = 5$ and $FF\_neighbor_4(2) = 6$. On the other hand, if node 6 is faulty and 4 is aware of that fault (i.e., 6 does not belong to $correct_4$), $FF\_neighbor_4(2) = 7$. If both 6 and 7 are faulty and 4 has been informed about their faults, $FF\_neighbor_4(2) = \perp$.

- $neighborhood_i(h)$ returns $\{j \mid j = FF\_neighbor_i(s), 1 \leq s \leq h\}$, 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 3-VCube of Fig. 1 and a fault-free scenario, $neighborhood_0(1) = \{1\}$, $neighborhood_0(2) = \{1, 2\}$ and $neighborhood_0(3) = \{1, 2, 4\}$. If node p_4 is detected as faulty by p_0, then $neighborhood_0(3) = \{1, 2, 5\}$; if p_1 is also detected as faulty by p_0, $neighborhood_0(1) = \perp$.

5.2. Description of the algorithm

The spanning tree algorithm can be started at an arbitrary *root* node i. Its children are returned by $neighborhood_i(\log_2 n)$. Each child j discovers that its parent is i with the similar function $neighborhood_j(cluster_j(i) - 1)$. Node i is a leaf if $cluster_i(j) = 1$ or if there is no correct processes in $c_{i,s}, s = cluster_i(j) - 1$. Note that the proposed spanning tree algorithm is autonomic because function *neighborhood* "automatically" adapts to the fault situation of each cluster.

All correct processes learn about the state of the other processes through VCube. If a crash is detected, all processes execute the event CRASH(j) (line 9). The process that crashed, say j, is removed from the local list of correct processes and a TREE message is sent to the next correct process k in the same cluster that contains j (if one exists). To discover to which cluster of a process i a process j belongs, we use function $FF\_neighbor_i(cluster_i(j))$, defined above.

First consider a fault-free execution. In the first step, the process that starts the algorithm sends $\log_2 n$ request messages, each to the first correct process of each cluster $s = 1, \ldots, \log_2 n$. Upon receiving message TREE from a process j, a process i retransmits it to its own clusters, from cluster 1 to cluster $s = cluster_i(j) - 1$, i.e., all internal clusters smaller than the cluster s from which process i received the TREE message. For example, consider the 3-VCube of Fig. 2(a). Process p_0 starts sending a message TREE to the first correct process of clusters $c_{0,1} = (1)$, $c_{0,2} = (2, 3)$ and $c_{0,3} = (4, 5, 6, 7)$. Process p_1 receives the message, verifies that $s = 1$ and does not forward the message. Process p_2 verifies that $s = 2$ and forwards the message TREE to p_3, which is the first correct process of cluster $c_{2,1} = (3)$. When receiving the message, p_3 does not retransmit it, since $s = 1$. Finally, process p_4 receives the message with $s = 3$ and forwards it to clusters $c_{4,2} = (6, 7)$ and $c_{4,1} = (5)$. The last message is sent by p_6 to p_7. Thus, all processes receive a single copy of the message sent initially by p_0 in the tree topology.

Now consider the algorithm executed in scenarios with faulty nodes. Before sending a message, the process queries the monitoring system and checks if there is a correct process in the destination cluster. If all processes in the cluster are faulty no message is sent and the process tries the next cluster. Fig. 2(b) shows an example of execution. Process p_0 sends message TREE to p_1 and p_2, as described before in the fault-free scenario. However, in the last round, as process p_4 is faulty, p_0 sends message to p_5, that is the first correct process in cluster $c_{0,3} = (4, 5, 6, 7)$. Now consider p_5: for $s = 1$ the message would be forwarded to $c_{5,1} = (4)$, but p_4 is faulty and for $s = 2$ the message is sent to p_7. Since $cluster_7(5) = 2$ ($c_{5,2} = \{7, 6\}$), p_7 retransmits the message to its cluster $c_{7,1} = (6)$. The transmission thus completes and the tree is formed.

5.3. Proof of correctness

In Lemma 5.1 we show that the proposed autonomic spanning tree built with Algorithm 5.1 can be used to propagate a message to every process in the system.

Lemma 5.1. *Let m be a message broadcast by a correct source process src. Every other correct process in P receives m.*

Proof. We prove this lemma by induction. We first consider as the basis of the induction that $n = 2$: p_0 is the src process that broadcasts message m and $p_1 \in c_{0,1}$. If p_1 is correct, $FF\_neighbor_0(1) = 1$ and p_0 sends m to p_1 (line 3). Therefore, p_1 receives m, and the lemma holds.

We assume as the induction hypothesis that for $n = 2^k$ every correct process correctly receives m broadcast by src.

Now for the induction step consider a system with $n = 2^{k+1}$ processes. This system can be seen as consisting of two subsystems each of which with $n = 2^k$ nodes as shown in Fig. 3. The figure shows that src and j are the roots of these subsystems. As the src process executes the algorithm it sends m to every process returned by $FF\_neighbor_{src}(s)$, $s = 1, \ldots, k$ (line 3). Since $j = FF\_neighbor_{src}(k)$ is a correct process, it correctly receives m. If j is detected as faulty, m is sent to the next fault-free process in the same cluster of j (line 11). Thus, message m is broadcast in the two subsystems and, based on the above hypothesis, it is guaranteed that every correct process correctly receives m in each subsystem. As every process in P is in one of these systems, every correct process in P correctly receives m. □

The spanning tree described in this section is used by our best-effort broadcast algorithm, described next in Section 6. The

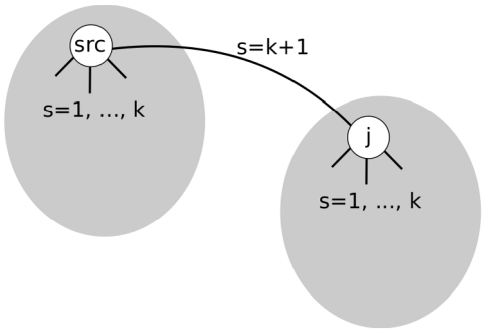


Fig. 3. Message propagation in the spanning tree built over VCube.

best-effort broadcast algorithm in its turn is used by the fault tolerant mutual exclusion algorithm, described in Section 7.

6. An autonomic best-effort broadcast algorithm

A process of a distributed system uses broadcast to send a message to all processes in the system. Broadcast is a basic building block used to implement several distributed algorithms and services such as event notification, content delivery, replication, and group communication [22,24]. Fault-tolerant broadcast algorithms [18] are particularly relevant as faults can affect the ability of disseminating messages in many ways. For instance, if a process fails as it is broadcasting a message, some processes can receive the message while others do not. Fault-tolerant broadcast algorithms are usually implemented by using reliable point-to-point links and primitives SEND and RECEIVE. It is common to define primitives BROADCAST(m) and DELIVER(m) to broadcast and deliver a message m. A failure detector can be used to notify the broadcast algorithm that some process has failed, and the algorithm can then react accordingly as faults are detected.

In this work we focus on a fault-tolerant broadcast algorithm that implements a best-effort solution. Best-effort broadcast ensures that, if the sender is correct, all correct processes deliver the message that it broadcasts. Thus, best-effort broadcast assumes that the sender does not fail. Other strategies such as reliable broadcast must be implemented if this assumption cannot be adopted [18]. In [34], we proposed an autonomic hypercube-based reliable broadcast. Best-effort broadcast is characterized by two properties: *validity* and *integrity* [17]. Validity is a liveness property and ensures that if a process i broadcasts a message m then every correct process eventually delivers m. Integrity is a safety property guaranteed by the *no-creation* and *no-duplication* properties. No-duplication ensures that no message is delivered twice, and no-creation guarantees that no message m sent by process i is delivered by any correct process unless it was previously broadcast by i.

Algorithm 6.1 presents a solution to best-effort broadcast using the spanning tree mechanism proposed in Section 5. It is then used by our autonomic k-mutual exclusion algorithm (Section 6). Processes get information about the state of other processes by using VCube. This broadcast algorithm tolerates up to $n-1$ processes crashes and still works correctly.

6.1. Variables and messages

Two types of messages are used:

- $\langle TREE, m\rangle$: identifies the application message m broadcast by a sender;
- $\langle ACK, m\rangle$: confirms the delivery of m by a receiver.

The local variables kept locally by each process are:

- $correct_i$: the set of processes considered correct by process i obtained from the underlying VCube;
- $last_i[n]$: last message delivered by each sender process;
- ack_set_i: set of pending ACK messages expected by process i. For each message $\langle TREE, m\rangle$ received by process i from a process j and retransmitted to a 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 wild card to select *acks* in the set *ack_set*. An element $\langle j, *, m\rangle$, for example, represents all pending *acks* for a message m received by process j and retransmitted to any other process ($*$).

Algorithm 6.1 The hierarchical best-effort broadcast executed by process i

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

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

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

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

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

32: **upon notifying** `crash`(j) ▷ j is detected as faulty by VCube at i
33: $correct_i \leftarrow correct_i \setminus \{j\}$
34: $k \leftarrow FF\_neighbor_i(cluster_i(j))$
35: **for all** $p = x, q = y, m = z : \langle x, y, z\rangle \in ack\_set_i$ **do**
36: **if** $\{source(m), p\} \nsubseteq correct_i$ **then** ▷ remove pending ACKs to $\langle j, *, *\rangle$ and $\langle *, *, m\rangle : source(m) = j$
37: $ack\_set_i \leftarrow ack\_set_i \setminus \{\langle p, q, m\rangle\}$
38: **else if** $q = j$ **then**
39: **if** $k \neq \perp$ **and** $\langle p, k, m\rangle \notin ack\_set_i$ **then** ▷ retransmit to the new neighbor k
40: $ack\_set_i \leftarrow ack\_set_i \cup \{\langle p, k, m\rangle\}$
41: SEND($\langle TREE, m\rangle$) **to** p_k
42: $ack\_set_i \leftarrow ack\_set_i \setminus \{\langle p, j, m\rangle\}$
43: CHECKACKS(p, m)

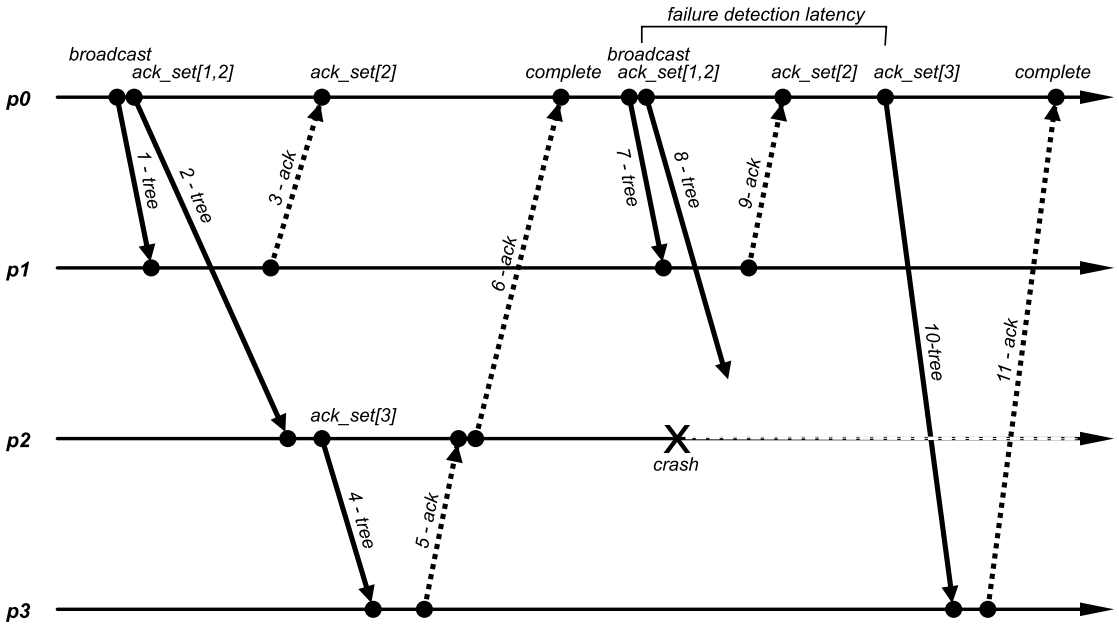


Fig. 4. Example of execution of the best-effort broadcast algorithm.

6.2. Description of the algorithm

A process i broadcasts a message m by invoking BROADCAST(m). Line 5 ensures that a new broadcast is started only after the previous one has been concluded, i.e., when there are no more pending *acks* to the message in $last_i[i]$. So, a new message is sent to all neighbors considered correct by process i (lines 8–10). For each message sent, an *ack* is inserted in the list of pending acks.

When a process receives a message TREE from a process j (line 15), first it verifies if either the source or process j is considered correct. If the source and j are correct, process i checks if the message m is new by comparing m's timestamp and the timestamp of the message stored in $last_i[j]$ (line 19). If m is new, $last_i[j]$ is updated and the message is delivered to the application layer. Next, m is retransmitted to the neighbors in each internal cluster. On the other hand, if there is no correct neighbor or if i is a leaf ($(cluster_i(j) = 1)$, no *ack* is added to the set ack_set_i and CHECKACKS sends an ACK message to j, completing the broadcast in that branch. If either the original sender of j is faulty, the delivery is aborted because: (1) if j is faulty, the process that sent m to j will detect that j became faulty and will retransmit m to the next fault-free process in the same cluster of j, rebuilding the tree; and (2) if the source is faulty, by the validity property it is not necessary guarantee the delivery of m anymore.

Whenever a message $\langle ACK, m\rangle$ is received, the respective *ack* is removed from the set ack_set_i and, if there are no more pending *acks* to message m, CHECKACKS sends a $\langle ACK, m\rangle$ to the process k from which i had previously received the message TREE. However, if $k = \perp$, it means that the ACK message reached the source process and the ACK propagation from that branch is completed. The broadcast started by a source i is finished when all ACKs are received from each branch of the tree rooted at i.

If a process j is detected as faulty, the CRASH event is executed. Three actions are performed: (1) the list of correct process is updated; (2) the pending *acks* containing process j as source or destination are removed from the list; and (3) the messages in the list of pending *acks* sent to process j are retransmitted to the next fault-free process k in the same cluster of j, if one exists. This last action starts a propagation through the branch of the new tree rebuilt after the crash.

6.3. Example of execution

Fig. 4[1] shows an example of execution of the hierarchical best-effort broadcast algorithm proposed in a system with $n = 4$ processes. In the first request, all processes are correct. The broadcast is started by process p_0. Initially, p_0 sends the TREE message to its neighbor $p1$ and p_2 (messages 1 and 2). Process p_1 delivers the message and sends an ACK to confirm it has delivered the TREE message (message 3). Then, process p_2 retransmits the message to its neighbor p_3, completing the spanning tree. The ACK message is returned from p_3 to p_2, and finally to p_0, ending the broadcast.

In the second broadcast, process p_2 fails, but p_0 is not aware. In this case, p_0 sends the TREE message to p_1 and p_2, but only process p_1 returns the ACK message to p_0. After the detection latency, p_0 is notified by VCube about p_2's crash. Then, p_0 retransmits the TREE message to the next fault-free process in the same cluster, that is p_3. As soon as p_0 receives the ACK from p_3 (message 11) the broadcast is completed.

6.4. Proof of correctness

The correctness of Algorithm 6.1 as a solution to best-effort broadcast (Theorem 6.3) is guaranteed by Lemma 6.1 (validity property) and Lemma 6.2 (integrity property), presented next.

Lemma 6.1 (*Validity*)**.** *If a correct process i broadcasts a message m, then every correct process eventually delivers m, including i.*

Proof. Consider the procedure BROADCAST(m) of Algorithm 6.1. Process $i = src$ is the source process. It sends m to all neighbors, one from each cluster $s = 1..\log_2 n$ and waits from all ACK messages related to the last message broadcast by it, stored in $last_i[i]$ (line 5). As soon as the broadcast algorithm uses the spanning tree construction proposed in Section 5, it was proved in Lemma 5.1 that a message broadcast by a sender src is delivered by all correct processes in the system. Once the source has received all pending *acks*, $last_i[i]$ is updated and the source delivers the message itself (lines 5–7). □

[1] Time space diagram generated using http://pluxos.github.io/dis-sys-vis/live/.

Lemma 6.2 (*Integrity*). *Every correct process i delivers a message m from a given source process sr at most once (no-duplication property) and if and only if m was previously broadcast by src (no-creation property).*

Proof. It was proven by Lemma 6.1 that a message m broadcast by a source process *src*, m is delivered by all correct processes j in the system by using the spanning tree rooted at *src*. When a process j receives m, j verifies if m is new comparing timestamps (line 19) and, if m is new, j delivers m. In this way, even if a message m is retransmitted after a failure is detected and gets to a process j that has already received m, j will never deliver m twice. No-creation is derived from the properties of the reliable links. □

Theorem 6.3. *Algorithm 6.1 is a solution to best-effort broadcast: if sender i is correct, all correct processes will deliver the same set of messages broadcast by i.*

Proof. The theorem is correct due to the validity and integrity properties, formalized by Lemmas 6.1 and 6.2, respectively. □

7. An autonomic *k*-mutual exclusion algorithm

The distributed k-mutual exclusion algorithm we propose in this work is a permission-based algorithm that tolerates up to $n-1$ faulty processes. The algorithm obtains monitoring information from VCube, described in Section 4. The underlying VCube provides state information (correct or faulty) about every system process. Processes disseminate requests by using the best-effort broadcast mechanism described in Section 6. Processes keep global logical clocks [21] to keep track of event causality. Algorithm 7.1 presents our solution.

7.1. Variables and messages

The distributed k-mutual exclusion algorithm employs two types of messages:

- $\langle REQUEST, i, C_i \rangle$: a message issued by p_i to all other processes whenever it wants to access a resource unit. A message REQUEST is timestamped with the pair $(C_i; i)$, i.e., the current value of p_i's logical clock and its identifier. The timestamp provides Lamport's total order for the requests: $(C_i; i) < (C_j; j) \Leftrightarrow C_i < C_j$ *or* $(C_i = C_j$ *and* $i < j)$;
- $\langle REPLY, count \rangle$: message sent by process i to answer to one or more requests from p_j meaning that p_i gives permission to process j to access a resource unit. The *count* parameter informs the number of requests from p_j that p_i had postponed (details are given below).

Each process p_i keeps the following local variables:

- $correct_i$: the set of processes considered correct by process i according to VCube;
- $state_i$: stores the current state of the process, which can be *requesting*, *not_requesting* or *executing*;
- $clock_i$: used to implement the logical clock locally. Initially set to zero, $clock_i$ is updated whenever a new request is issued or a new REQUEST message is received. Upon reception of a REQUEST message, p_i updates its $clock_i$ with the maximum value between $clock_i$ value and the timestamp of the REQUEST message received;
- $last_i$: timestamp of the last REQUEST message sent;
- $perm\_count_i$: the total number of permissions received (REPLY messages) since its last request;
- $reply\_count_i[n]$: array that stores the number of replies expected from each process;
- $defer\_count_i[n]$: stores the number of deferred replies to each process. After releasing the resource, the process sends all outstanding permissions in a REPLY message.

7.2. Description of the algorithm

The interface of the distributed k-mutual exclusion algorithm consists of two functions that application processes use: `REQUEST()`, is employed by a process to ask for permissions to access a unit of the resource, and `RELEASE()`, executed when a process releases a resource unit. A process gets access to a unit of a resource if it gets $n - f - k$ permissions, where f is the number of current faulty processes informed by VCube to the requesting process.

Algorithm 7.1 The Autonomic k-Mutual Exclusion Algorithm at process i

```
1:  correct_i ← {0, .., n − 1}                                   ▷ Initialization
2:  state_i ← not_requesting
3:  ∀j ∈ n : reply_count_i[j] ← 0
4:  ∀j ∈ n : defer_count_i[j] ← 0
5:  perm_count_i ← 0
6:  clock_i ← 0
7:  last_i ← 0

8:  procedure REQUEST( )
9:      state_i ← requesting
10:     clock_i ← clock_i + 1
11:     last_i ← clock_i
12:     perm_count_i ← 0
13:     BROADCAST(REQUEST(i, last_i))                            ▷ using best-effort broadcast
14:     ∀j ≠ i, j ∈ correct_i : reply_count_i[j] + +
15:     wait until (perm_count_i ≥ |correct_i|−k)
16:     state_i ← executing

17: procedure RELEASE( )
18:     state_i ← not_requesting
19:     for all (j ≠ i : j ∈ correct_i) do
20:         if (defer_count_i[j] ≠ 0) then
21:             SEND(REPLY(defer_count_i[j])) to p_j
22:             defer_count_i[j] ← 0

23: upon receive REQUEST(j, last_j) from p_j
24:     clock_i ← max(clock_i, last_j)
25:     if (p_j ∈ correct_i) then
26:         if (state_i = executing or (state_i = requesting and (last_i, i) <
    (last_j, j))) then
27:             defer_count_i[j] + +
28:         else
29:             SEND(REPLY(i, 1)) to p_j

30: upon receive REPLY(j, count ) from p_j
31:     if (j ∈ correct_i) then
32:         reply_count_i[j] ← reply_count_i[j] − count
33:         if (state_i = requesting and reply_count_i[j] = 0) then
34:             perm_count_i + +

35: upon notifying crash(j)                                      ▷ j is detected as faulty by VCube at i
36:     if (state_i = requesting and reply_count_i[j] = 0) then
37:         perm_count_i − −
38:     correct_i ← correct_i ∖ {j}
```

The REQUEST() function starts by changing the state of process p_i to *requesting* (line 9). Then, a timestamped REQUEST message with p_i's local logical clock value is broadcast using the best-effort broadcast algorithm (line 13). If p_i receives a request from another process p_j while it is in the requesting state, it first checks whether the p_j's REQUEST message timestamp is smaller than p_i's or, in case

of a tie, if p_j's identifier is smaller than its own identifier (line 26). In these cases, p_j's request has higher priority than p_i's request and, therefore, p_i gives its permission for p_j to access the resource (it sends a REPLY message in line 29). Otherwise, p_i defers sending the REPLY message, incrementing $defer\_count_i[j]$ (line 27).

As mentioned above, by accessing the VCube monitoring system, the algorithm dynamically gets information about the number of faulty processes. This number is subtracted from the number of expected responses, allowing processes to get access to the resource even when k or more processes are faulty (line 15). Upon receiving the minimum number of permissions, process p_i gains access to a unit of the resource and changes its state to *executing*. Note that when a process is detected as faulty during the requesting phase, the algorithm checks if a permission has already been received from that process (lines 36–37). If this is the case, *perm_count* is decremented in order to keep consistent the condition in line 15, ensuring, thus, the safety property. We should point out that (for sake of simplicity the code for this part is not show), if a crash is detected, the condition in line 15 is checked only after the number of permissions received has been decremented in order to ensure that permissions sent by the crashed process is not considered in the final value of *perm_count*.

When p_i releases the resource, it changes its status to *not_requesting* and sends all deferred REPLY messages stored in *defer_count* (lines 18 –22) to the respective requesting processes. Hence, processes that are waiting for permissions can check the condition in line 15 and eventually get one unit of the resource.

7.3. Example of execution

Fig. 5 shows an execution of the k-mutual exclusion algorithm in a system with $n = 4$ processes and $k = 2$ resources. First, consider p_0 broadcasts a request(0) message by using the best-effort broadcast algorithm (ACKs are not presented). The message will be delivered to $p1$ and p_2. Then, process p_2 will retransmit the request message to its neighbor p_3, completing the spanning tree. As they are not requesting, everyone will reply immediately and p_0 will get the permission to access the critical section.

While p_0 is executing the critical section, process p_2 starts a request procedure. It sends a request message to its neighbors p_3 and p_0. Process p_3 is not requesting and replies immediately. On the other hand, p_0 is executing and defers the reply message, but retransmits the request to its neighbor p_1, following the spanning tree. Process p_1 also replies immediately, since it is neither executing nor requesting the critical section. At this point, p_2 receives two REPLY messages and can access the second resource. Finally, process p_1 starts a request procedure, sending request messages to p_0 and p_3 (that retransmits the request to p_2). Process p_3 replies immediately, but p_0 and p_2 do not, since both are using one copy of the resource. As soon as p_0 or p_2 releases the critical section, the deferred reply will be delivered to p_1, given access to a free resource (in this example, after p_0's release).

7.4. Proof of correctness

In order to ensure the correctness of the distributed k-mutual exclusion algorithm, two properties must be satisfied:

- *Safety*: at any time, at most k different processes can concurrently access the k resource units (one process per unit). In other words, at most k process are in the critical section at a given time instant;
- *Liveness*: a correct process that makes a request to access one unit of the resource, will eventually have access to it. This property ensures the absence of deadlock and starvation.

Besides the two properties above, the fact that a new request is broadcast only after the previous one was completed and that REQUEST messages timestamps define a total order for requests, the fairness for the satisfaction of the requests is guaranteed by the algorithm. In the context of mutual exclusion, the *fairness* property ensures that each process has the same chance to execute the critical section.

Lemma 7.1 (*Safety*)**.** *Algorithm 7.1 ensures that no more than k different processes get permission to use the k existing units of the shared resource at the same time instant.*

Proof. Consider that at time t, the k units of the resource are being used by k processes. Let $f = FD_i.crashed$ be the number of faulty processes detected by VCube and informed to process p_i. If process p_i wants to get access to the resource, it sends a REQUEST message to $n - f - 1$ other processes. For each process p_j that receives the request, four cases are possible, depending on p_j state:

1. if p_j is *not_requesting*, it sends a REPLY message to p_i immediately;
2. if p_j is *executing*, it defers the REPLY until it releases the resource;
3. if p_j is *requesting* and $(last_i, i) < (last_j, j)$, it sends the REPLY to p_i immediately, because in this case, p_i has a higher priority over p_j. Otherwise, it defers the reply until it gets and releases the resource;
4. If p_j is crashed, it does not send any reply.

If k processes are using the k units of resource, k replies will be deferred (line 27). Therefore, at most $n-k-f-1$ processes can be in the states of case (1) or (3), giving their respective permission to p_i immediately upon receiving its request. Since process p_i needs $n - f - k$ replies to get a unit of the resource, it will have to wait until one of the processes using a resource releases it by sending a REPLY message (line 20) or crashes. In the latter case, f will be incremented and p_i can get all permissions needed (line 15). Consequently, at most k processes can use the k resources at the same time.

Note that if a process p_j crashes after sending a REPLY message to a process p_i, the number of permissions must be decremented in p_i (line 37) in order to avoid that a faulty process contributes twice (in f and *perm_count*), breaking the safety property ensured by line 15. □

Lemma 7.2 (*Liveness*)**.** *Every request for a resource executed by a correct process i is eventually satisfied by Algorithm 7.1.*

Proof. Since pending requests are totally ordered by their timestamps and a new request has lower priority than the pending ones ensures that every request will eventually be the one with the highest priority. Let us consider a request from process p_i with the highest priority and the four possible cases enumerated in Lemma 7.1. Due to the safety property, at most k processes can be accessing the resource at the same time (in the CS) and, therefore, these processes will defer sending REPLY messages (2). If all other $n-f-k-1$ processes are requesting, but p_j has the highest priority, then p_j will receive $n-f-k-1$ replies according to (3). However, as soon as one of the k processes in CS releases one unit of the resource, the deferred REPLY message will be received by p_j. On the other hand, if one of the k processes fails, f will be incremented (4). In both cases, p_j will receive $n - f - k$ replies and will then obtain the necessary number of permissions to access the resource unit that is now free. □

Lemma 7.3 (*Fairness*)**.** *In Algorithm 7.1, requests are satisfied in the order they are issued, considering the priority defined by the logical clock.*

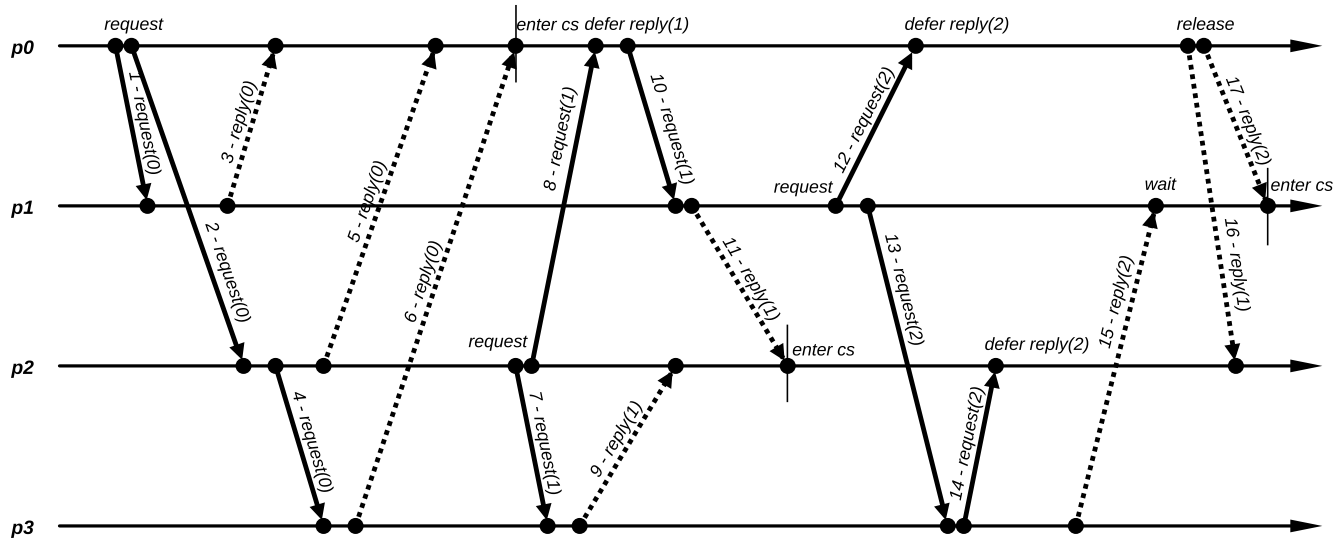


Fig. 5. Example of execution of the k-mutual exclusion algorithm.

Proof. Each request message sent by a process p_i is identified by a unique timestamp $last_i$ defined by its current logical clock value. A new request is broadcast only after the previous one has been completed and before issuing the former, the timestamp is incremented (line 10). In addition, for each REQUEST message received by a process, its logical clock is updated in line 24 with the highest value between its local clock value and the timestamp value of the REQUEST message. Furthermore, whenever p_i is waiting to get access to the resource and receives a REQUEST message from p_j with timestamp $last_j$, p_i checks if its last request has priority over the request from p_j, i.e., $(last_i, i) < (last_j, j)$ (line 26). Since process p_i only replies to requests with higher priority than its own current request, the systematic request ordering is ensured. □

Theorem 7.4. *Algorithm 7.1 is a solution for the k-mutual exclusion problem, tolerating up to $n - 1$ process failures.*

Proof. Theorem 7.4 follows directly from Lemmas 7.1 and 7.2. □

8. Experimental results

In this section we present results obtained by simulation on top of the Neko framework [41]. VCube was implemented using the Java classes available in Neko's fault tolerance library.[2] The proposed k-mutual exclusion algorithm, in this section denoted VCUBE, was compared with two other solutions: Raymond's algorithm [28] (RAY) and the solution proposed by Bouillaguet, Arantes, and Sens [8] (BAS). RAY does not use failure detection, but intrinsically tolerates $k - 1$ faults. However, each crash degrades the performance of the algorithm. BAS uses the failure detector $\mathcal{T}$ and tolerates $n - 1$ faults. See Section 2 for more details about these algorithms. Unlike our algorithm, both RAY and BAS use a one-to-all strategy to broadcast request messages, i.e., a process needs to send a copy of the message directly to every other process in the system and waits for replies.

8.1. Parameters

The simulation model is based on [10]. Each process executes a new request at τ time units (t.u.) after the previous request was completed. After getting permission to use a unit of the resource, a process uses the unit for e time units. 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. Considering that there is no broadcast facilities available in the system, if a message is sent to multiple destinations, t_s is multiplied by the number of copies sent.

The algorithms were compared under two different load scenarios. Under light load (LOW), only k processes request one of the k units of the resource. On the other hand, under high load (HIGH), all processes send request concurrently.

The following metrics were used to compare the algorithms:

- the *total number of resource units* allocated during the execution;
- the *obtaining time* for a process to get the resource, defined as the time interval from the instant that the process requests the resource until the moment it gets permission to access a resource unit;
- the *number of messages* (requests and replies) sent per request.

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 $t_t = 0.8$; time to use a resource $e = 0.0002$; time interval between two consecutive requests by a process $\tau = 0.1$; and total number of resources available $k = 3$. Each execution was performed for 1000 time units. The parameter f represents the total number of faulty processes.

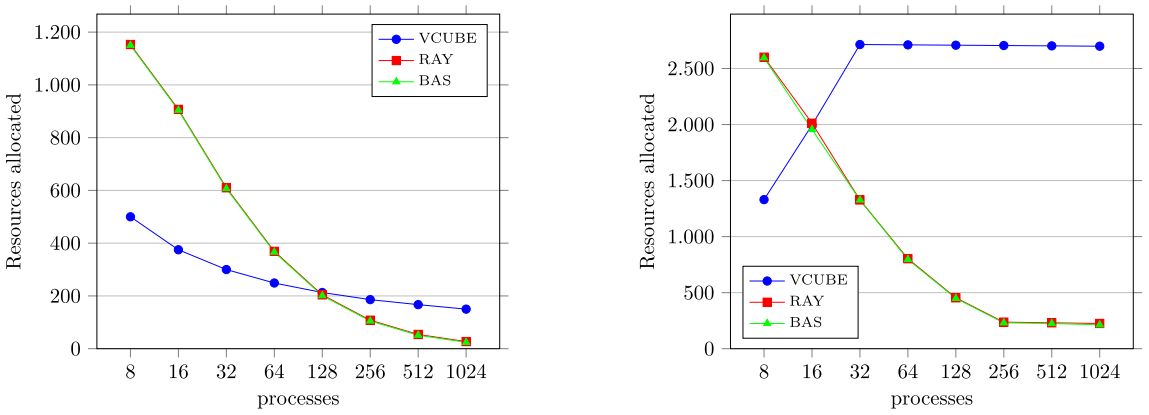
8.2. Results

Experiments were conducted in both fault-free scenarios, in which no process crashes, and in faulty scenarios, where a varying number of processes crash during the execution. The number of processes varied from 8 to 1024.

8.2.1. Experiments in fault-free scenarios

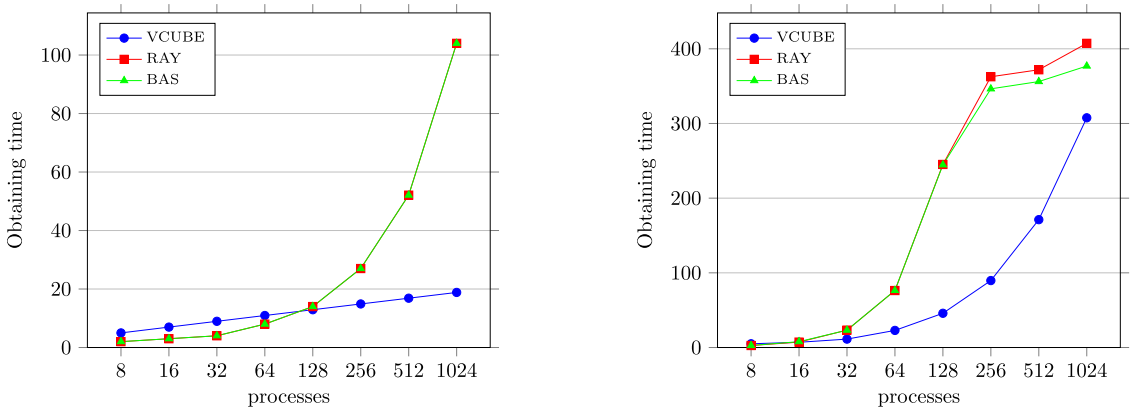
The mutual exclusion algorithms VCUBE, RAY, and BAS were executed in scenarios with different number of processes and under LOW and HIGH loads. In LOW load, only the first $k = 3$ processes with identifiers from 0 to $k - 1$ request a unit of the

[2] All source code and configuration files are available at www.inf.unioeste.br/~luiz.



(a) LOW load. (b) HIGH load.

Fig. 6. Resources allocated in fault-free scenarios with $k = 3$ during 1000 time units.



(a) LOW load. (b) HIGH load.

Fig. 7. Obtaining time in fault-free scenarios with $k = 3$ during 1000 time units.

resource. Under faulty-free scenarios, RAY, and BAS present similar results, since they use the same one-to-all message propagation algorithm.

Fig. 6 shows the total number of resources allocated by all algorithms using the two types of load and a constant $k = 3$. Under light load, it can be observed in Fig. 6(a) that for small systems (8–64 processes) RAY and BAS are more efficient, since the requester sends messages directly to the other $n - 1$ processes which consume less time than the time to propagate messages over the VCUBE virtual tree. On the other hand, with 128 processes, the three algorithms present equivalent performance and with more than 128 processes, more unit of resources are used concurrently in VCUBE than in RAY and BAS. Considering the HIGH load, Fig. 6(b) shows that RAY and BAS have similar performance as in the previous case, because of the one-to-all propagation strategy. On the other hand, VCUBE increases the allocation rate quickly and surpasses the others from around 16 processes, when the hierarchical message propagation strategy starts to make a difference. The number of resources allocated peaks at about 64 processes for VCUBE and 256 processes for both RAY and BAS, in this case the limitation is posed by the propagation delay of broadcasting multiple copies of the request messages by all processes concurrently.

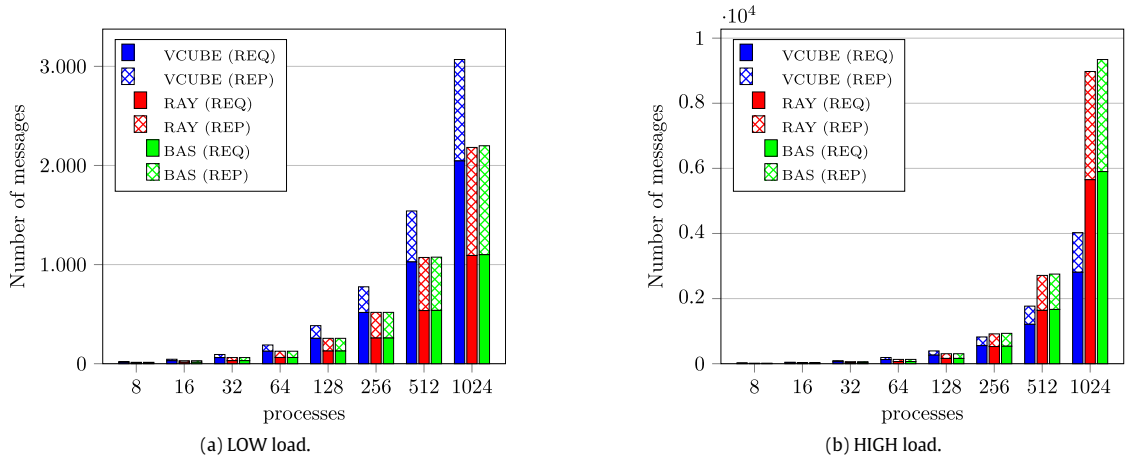
Fig. 7 shows the average time to obtain resource units under LOW and HIGH loads. BAY and BAS present similar performance, due to the one-to-all broadcast mechanism. Under light load, when the number of processes is greater than 128, VCUBE presents higher efficiency. Under high load, VCUBE presents linear growth, much smaller than the others. This behavior is due to the hierarchical broadcast strategy based on spanning trees used by VCUBE.

The mean number of messages (REQUEST and REPLY) per request is shown in Fig. 8. Algorithms RAY and BAS send the same number of messages, since they use the same one-to-all broadcast strategy. On the other hand, VCUBE sends more messages because of the tree-based strategy in which messages (specially REQUEST messages) are sent along the tree. Even with a larger number of messages, the best-effort algorithm used by VCube prevents the ACK implosion problem in the broadcast layer. In HIGH scenarios, large numbers of REQUEST/REPLY messages are transmitted by VCUBE, RAY, and BAS. However, considering that VCUBE presents a resource allocation rate much higher than the other algorithms, in average VCUBE uses fewer number of messages than the other algorithms, especially for 512 and 1024 processes.

Taking into account the results for the three parameters described above, we can conclude that, as the number of processes grows, VCUBE is more efficient (more unit of resources are allocated and lower obtaining time) than RAY and BAS both in light and high loads, even if more messages are needed per request.

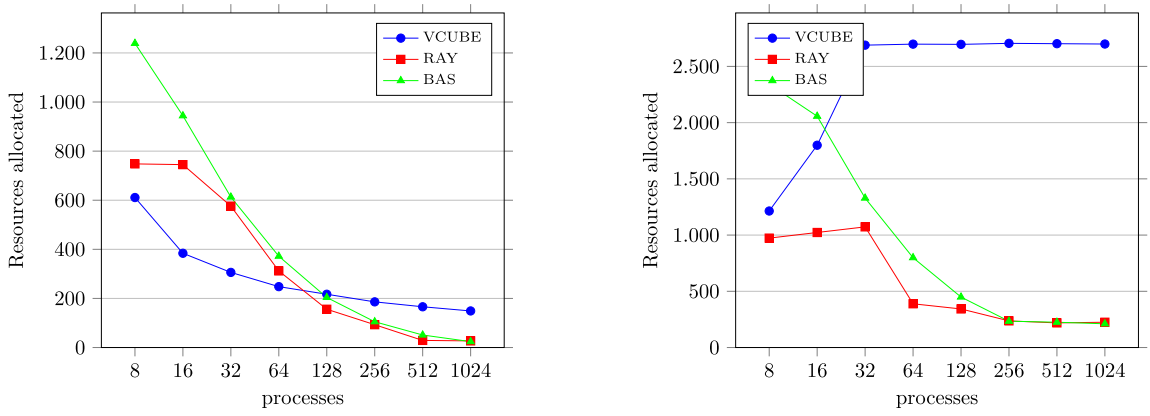
8.2.2. Experiments in faulty scenarios

Similarly to the experiments conducted in the fault-free scenarios, experiments with fault processes were also conducted under LOW and HIGH loads. We set $k = 3$ resources to all experiments. Considering that RAY does not tolerate more than k faulty process,



(a) LOW load. (b) HIGH load.

Fig. 8. Mean number of messages per request in fault-free scenarios with $k = 3$ during 1000 time units.



(a) LOW. (b) HIGH.

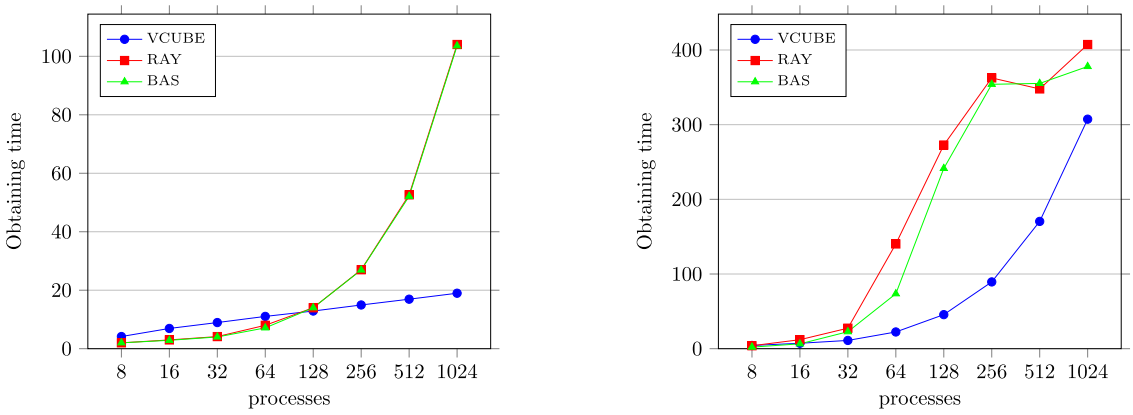
Fig. 9. Resources allocated in faulty scenarios with $k = 3$ during 1000 time units and $f = 3$ random faults.

three random processes were set to fail at random time instants. In the LOW load scenarios, processes from 0 to 2 never fail, since they are the ones which send requests. Each run was simulated for 1000 time units. The total number of resources allocated is shown in Fig. 9. Different from the fault-free results, under the faulty scenario, the performance of RAY is worse than BAS's, since it does not use a failure detector to identify crashes, i.e., it is not aware that the system size has decreased when processes fail. However, as the system size grows, the one-to-all broadcast time compensates the waiting time of RAY and, therefore, their performance is similar, as shown in the systems from 256 processes under LOW and HIGH loads, shown by Figs. 9(a) and 9(b). We should point out that the reason for the fast allocation rate growth of VCUBE in Fig. 9(b) is the same as explained for Fig. 6(b).

The time to obtain a unit of resource unit in the faulty scenarios is shown in Fig. 10. Considering both LOW and HIGH loads, VCUBE and BAS present the same behavior that they presented in the fault-free scenarios, and the reason is that they are informed about failures. On the other hand, RAY presents a poor performance for small systems (8–32 processes) as processes get blocked waiting for replies from processes that are actually faulty. As the system size increases, the time to broadcast the REQUEST messages by RAY and BAS compensates the waiting time and the performance of RAY becomes equivalent to BAS'. The performance results shown in the figures confirm clearly the benefits of the hierarchical propagation and the logarithmic detection delay of the proposed solution.

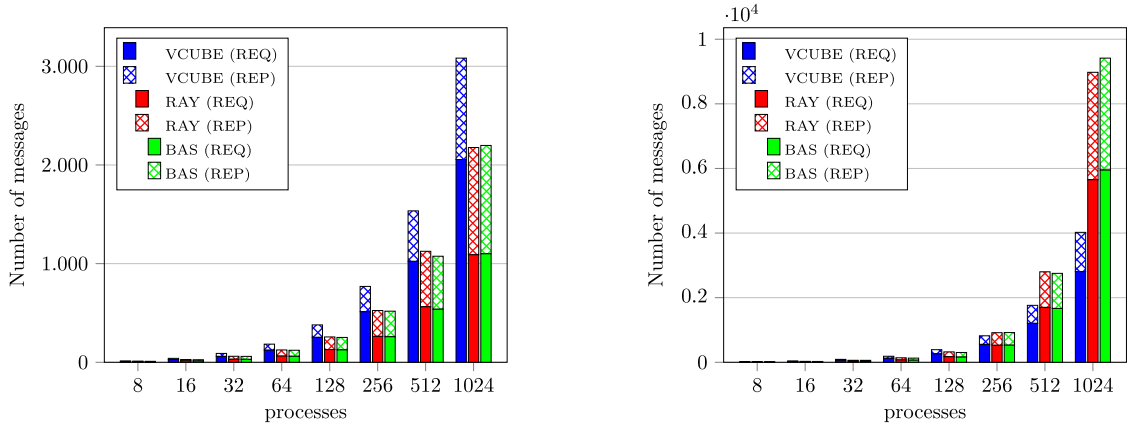
The number of messages per request under faulty scenarios is presented in Fig. 11. The results are similar to those obtained in fault-free scenarios. Under a LIGHT load (Fig. 11(a)), for example, as the system increases in size, VCube sends more messages than the others, due to the reverse spread of ACKs on the tree. However, it is possible to see that the number of messages sent by RAY and BAS increases considerably. VCube sends less messages when compared with the fault-free scenarios. This occurs because VCube stops sending messages to faulty processes after it detects they are faulty. RAY always sends REQUESTs to all processes, even if they are faulty (no failure detection) and BAS uses a non-hierarchical failure detector, in general slower than VCube failure detector to detect events (specially in larger systems).

In the second experiment with faults, scenarios with 128 processes were carefully investigated. Random faults were generated from 1 to 100 processes at random time instants. Under a LIGHT load (Fig. 12(a)), in most cases, VCube allocates more resources than RAY and BAS. RAY is always worst, since it has no failure detection. As soon as the number of faulty processes increases, BAS is able to allocate more resources than VCube, as we can see when there are 20 faulty processes. Moreover, with 50 and 100 faulty processes, for example, BAS was unable to allocate even a single resource. This situation occurs due to the time the fault event happened and the diagnostic latency of the detector employed. Under a HIGH load, RAY and BAS are limited by the one-to-all broadcast mechanism. VCube presents high performance, supported by the



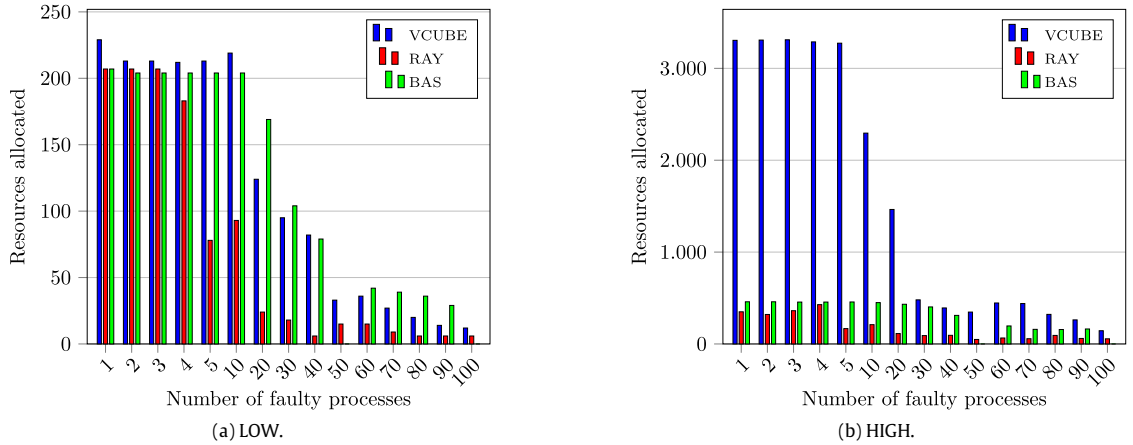
(a) LOW. (b) HIGH.

Fig. 10. Obtaining time in faulty scenarios with $k = 3$ during 1000 time units and $f = 3$ random faults.



(a) LOW load. (b) HIGH load.

Fig. 11. Number of messages in faulty scenarios during 1000 time units.



(a) LOW. (b) HIGH.

Fig. 12. Resources allocated in faulty scenarios with 128 processes during 1000 time units.

spanning-tree-based broadcast and the hierarchical failure detection mechanisms.

9. Conclusions

In this work we presented an autonomic distributed permission-based algorithm for k-mutual exclusion. The algorithm is autonomic in the sense that it adapts itself dynamically and transparently to changes in the system composition. The algorithm is built on top of a VCube, a virtual hypercube-like topology, which has several logarithmic properties. VCube monitors processes and after a crash is detected, it heals itself by reconnecting correct processes. An autonomic distributed spanning tree algorithm is proposed. It is built exploring the hierarchical cluster organization of VCube. A best-effort broadcast algorithm which is based on the autonomic spanning trees is then proposed to propagate messages reliably through the VCube. Finally, the fault tolerant k-mutual exclusion algorithm uses the best-effort broadcast algorithm to disseminate information among processes.

The k mutual-exclusion algorithm was described, specified and implemented. We present comparison results with two other algorithms: (1) the algorithm proposed by Raymond that tolerates up to $k - 1$ fault processes, but failures degrade its performance and (2) the algorithm by Bouillaguet, Arantes, and Sens that use a failure detector $\mathcal{T}$ to detect crashes; up to $n - 1$ crashes are tolerated. Experimental results show that, although the overall number of messages of our solution is greater than of the other two algorithms, our solution guarantees that the efficiency to obtain resources is maintained throughout the execution regardless of the number of faulty processes. In addition, the proposed algorithm presented the highest total number of resource units allocated during system execution in comparison with the other two algorithms in all scenarios. Finally, when the number of processes is greater than 64 our algorithm presents the lowest time to obtain a resource unit. These results confirm the efficiency and scalability of the proposed autonomic solutions.

Future research directions include extending the proposed algorithm to allow process recovery. Another direction is to defined autonomic mutual exclusion for systems with different resource types. Also left as future work is the implementation of the autonomic mutual exclusion strategy using quorums.

References

[1] D. Agrawal, A. El Abbadi, Efficient solution to the distributed mutual exclusion problem, in: Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing, in: PODC '89, ACM, New York, NY, USA, 1989, pp. 193–200. http://dx.doi.org/10.1145/72981.72994. URL http://doi.acm.org/10.1145/72981.72994.

[2] D. Agrawal, A. El Abbadi, An efficient and fault-tolerant solution for distributed mutual exclusion, ACM Trans. Comput. Syst. 9 (1) (1991) 1–20. http://dx.doi.org/10.1145/103727.103728. URL http://doi.acm.org/10.1145/103727.103728.

[3] S. Ahuja, N. Carriero, D. Gelernter, Linda and friends, Computer 19 (8) (1986) 26–34. http://dx.doi.org/10.1109/MC.1986.1663305.

[4] D.R. Avresky, Embedding and reconfiguration of spanning trees in faulty hypercubes, IEEE Trans. Parallel Distrib. Syst. 10 (3) (1999) 211–222. http://dx.doi.org/10.1109/71.755820.

[5] B. Awerbuch, I. Cidon, S. Kutten, Optimal maintenance of a spanning tree, J. ACM 55 (2008) 1–45. doi:http://doi.acm.org/10.1145/1391289.1391292.

[6] B.N. Bershad, M.J. Zekauskas, W.A. Sawdon, The midway distributed shared memory system, in: Digest of Papers. Compcon Spring, 1993, pp. 528–537, http://doi.org/10.1109/CMPCON.1993.289730.

[7] M. Bertier, L. Arantes, P. Sens, Hierarchical token based mutual exclusion algorithms, in: Proc. of the IEEE Int'L Symp. on Cluster Computing and the Grid, ACM, Washington, USA, 2004, pp. 539–546. URL http://dl.acm.org/citation.cfm?id=1111683.1111834.

[8] M. Bouillaguet, L. Arantes, P. Sens, Fault tolerant k-mutual exclusion algorithm using failure detector, in: Int'L Symp. on Parallel and Distr. Comp., IEEE, 2008, pp. 343–350. http://dx.doi.org/10.1109/ISPDC.2008.57.

[9] M. Bouillaguet, L. Arantes, P. Sens, A timer-free fault tolerant k-mutual exclusion algorithm, in: Fourth Latin-American Symposium on Dependable Computing, in: LADC'09, IEEE, 2009, pp. 41–48. http://dx.doi.org/10.1109/LADC.2009.10.

[10] S. Bulgannawar, N. Vaidya, A distributed k-mutual exclusion algorithm, in: Proc. of the 15th Int'L Conf. on Distr. Comp. Systems, IEEE Computer Society, Los Alamitos, CA, USA, 1995, pp. 153–160. doi:http://doi.ieeecomputersociety.org/10.1109/ICDCS.1995.500014.

[11] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, P. Kouznetsov, Mutual exclusion in asynchronous systems with failure detectors, J. Parallel Distrib. Comput. 65 (2005) 492–505. http://dx.doi.org/10.1016/j.jpdc.2004.11.008.

[12] E.P. Duarte Jr., L.C. Bona, V. Ruoso, Vcube: A provably scalable virtual topology for dynamic systems, in: Proc. of the 44th Annual IEEE/IFIP Int'L Conference on Dependable Systems and Networks, in: DSN, 2014.

[13] E.P. Duarte Jr., T. Nanya, A hierarchical adaptive distributed system-level diagnosis algorithm, IEEE Trans. Comput. 47 (1998) 34–45. http://dx.doi.org/10.1109/12.656078. URL http://portal.acm.org/citation.cfm?id=275418.275421.

[14] R.G. Gallager, P.A. Humblet, P.M. Spira, A distributed algorithm for minimum-weight spanning trees, ACM Trans. Program. Lang. Syst. 5 (1983) 66–77. doi:http://doi.acm.org/10.1145/357195.357200.

[15] H. Garcia-Molina, D. Barbara, How to assign votes in a distributed system, J. ACM 32 (1985) 841–860. doi:http://doi.acm.org/10.1145/4221.4223.

[16] F.C. Gärtner, A survey of self-stabilizing spanning-tree Construction Algorithms, Tech. rep., Swiss Federal Institute of Technology (EPFL), 2003.

[17] R. Guerraoui, L. Rodrigues, Introduction to Reliable Distributed Programming, Springer-Verlag, Berlin, Germany, 2006.

[18] V. Hadzilacos, S. Toueg, Fault-tolerant Broadcasts and Related Pproblems, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993, pp. 97–145. URL http://portal.acm.org/citation.cfm?id=302430.302435.

[19] H. Kakugawa, A Study on the Distributed k-Mutual Exclusion Problem (Master's thesis), Faculty of the Information Engineering, Graduate School of Engineering of Hiroshima University, 1992. URL http://www-masu.ist.osaka-u.ac.jp/~kakugawa/Reprints/master-thesis.pdf.

[20] A.D. Kshemkalyani, M. Singhal, Distributed Computing: Principles, Algorithms, and Systems, first ed., Cambridge University Press, New York, NY, USA, 2008.

[21] L. Lamport, Time, clocks and the ordering of events in a distributed system, Commun. ACM 21 (7) (1978) 558–565. http://dx.doi.org/10.1145/359545.359563. URL http://doi.acm.org/10.1145/359545.359563.

[22] J. Leitão, J. Pereira, L. Rodrigues, HyParView: A membership protocol for reliable gossip-based broadcast, in: DSN, 2007, pp. 419–429. http://dx.doi.org/10.1109/DSN.2007.56.

[23] G. Masson, D. Blough, G. Sullivan, in: D.K. Pradhan (Ed.), Fault-tolerant Computer System Design, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996 (Chapter) System Diagnosis.

[24] A. Mostefaoui, M. Raynal, Intrusion-tolerant broadcast and agreement abstractions in the presence of byzantine processes, IEEE Trans. Parallel Distrib. Syst. 27 (4) (2016) 1085–1098. http://dx.doi.org/10.1109/TPDS.2015.2427797.

[25] M. Naimi, Distributed mutual exclusion on hypercubes, SIGOPS Oper. Syst. Rev. 30 (3) (1996) 46–51. http://dx.doi.org/10.1145/230908.230917. URL http://doi.acm.org/10.1145/230908.230917.

[26] M. Naimi, M. Trehel, A. Arnold, A log n distributed mutual exclusion algorithm based on path reversal, J. Parallel Distrib. Comput. 34 (1996) 1–13.

[27] S.-H. Park, S.-H. Lee, Quorum based mutual exclusion in asynchronous systems with unreliable failure detectors, in: Grid and Distributed Computing, in: Communications in Computer and Information Science, vol. 261, Springer, 2011, pp. 25–34. http://dx.doi.org/10.1007/978-3-642-27180-9_4.

[28] K. Raymond, A distributed algorithm for multiple entries to a critical section, Inform. Process. Lett. 30 (1989) 189–193. http://dx.doi.org/10.1016/0020-0190(89)90211-1.

[29] K. Raymond, A tree-based algorithm for distributed mutual exclusion, ACM Trans. Comput. Syst. 7 (1) (1989) 61–77. http://dx.doi.org/10.1145/58564.59295. URL http://doi.acm.org/10.1145/58564.59295.

[30] M. Raynal, A simple taxonomy for distributed mutual exclusion algorithms, SIGOPS Oper. Syst. Rev. 25 (2) (1991) 47–50. http://dx.doi.org/10.1145/122120.122123. URL http://doi.acm.org/10.1145/122120.122123.

[31] M. Raynal, D. Beeson, Algorithms for Mutual Eexclusion, MIT Press, Cambridge, MA, USA, 1986.

[32] V.A. Reddy, P. Mittal, I. Gupta, Fair k mutual exclusion algorithm for peer to peer systems, in: 2008 the 28th International Conference on Distributed Computing Systems, 2008, pp. 655–662, http://doi.org/10.1109/ICDCS.2008.76.

[33] G. Ricart, A.K. Agrawala, An optimal algorithm for mutual exclusion in computer networks, Commun. ACM 24 (1) (1981) 9–17. http://dx.doi.org/10.1145/358527.358537. URL http://doi.acm.org/10.1145/358527.358537.

[34] L.A. Rodrigues, L. Arantes, E.P. Duarte Jr., An autonomic implementation of reliable broadcast based on dynamic spanning trees, in: Dependable Computing Conference (EDCC), 2014 Tenth European, 2014, pp. 1–12, http://dx.doi.org/10.1109/EDCC.2014.31.

[35] L.A. Rodrigues, J. Cohen, L. Arantes, E.P. Duarte Jr., A robust permission-based hierarchical distributed k-mutual exclusion algorithm, in: ISPDC, 2013, pp. 1–8.

[36] P. Romano, L. Rodrigues, An efficient weak mutual exclusion algorithm, in: 8th Int'L Symp. on Parallel and Distributed Computing, 2009, pp. 205–212, http://dx.doi.org/10.1109/ISPDC.2009.32.

[37] B.A. Sanders, The information structure of distributed mutual exclusion algorithms, ACM Trans. Comput. Syst. 5 (3) (1987) 284–299. http://dx.doi.org/10.1145/24068.28052. URL http://doi.acm.org/10.1145/24068.28052.

[38] P.C. Saxena, J. Rai, A survey of permission-based distributed mutual exclusion algorithms, Comput. Stand. Interfaces 25 (2) (2003) 159–181. http://dx.doi.org/10.1016/S0920-5489(02)00105-8.

[39] I. Suzuki, T. Kasami, A distributed mutual exclusion algorithm, ACM Trans. Comput. Syst. 3 (4) (1985) 344–349. http://dx.doi.org/10.1145/6110.214406. URL http://doi.acm.org/10.1145/6110.214406.

[40] A.S. Tanenbaum, M. Van Steen, Distributed Systems: Principles and Paradigms (2nd Edition), second ed., Prentice Hall, Upper Saddle River, NJ, USA, 2006.

[41] P. Urbán, X. Défago, A. Schiper, Neko: A single environment to simulate and prototype distributed algorithms, J. Inf. Sci. Eng. 18 (6) (2002) 981–997.

Luiz A. Rodrigues is assistant professor at Western Paraná State University (Unioeste), Brazil since 2007 and member of GIA (Research Group in Applied Intelligence). Luiz received B.Sc. degree in Computer Science from Unioeste (2003), M.Sc. Degree in Computer Science from the Federal University of Rio Grande do Sul (UFRGS, 2006) and Ph.D. in Computer Science from the Federal University of Paraná (UFPR, 2014). His main interests surround the Computer Science with emphasis on Computer Networks, Fault Tolerance, Distributed Systems and Algorithms, and Programming systems.

Elias P. Duarte Jr. is a Full Professor at Federal University of Parana, Brazil, where he is the leader of the Computer Networks and Distributed Systems Lab (LaRSis). He received a Ph.D. degree in computer science from Tokyo Institute of Technology, Japan (1997), the M.Sc. degree in telecommunications from the Polytechnical University of Madrid, Spain (1991), and the B.Sc. and M.Sc. degrees in computer science from Federal University of Minas Gerais, Brazil (1987/1991). His research interests include Computer Networks and Distributed Systems, their Dependability Management, and Algorithms. He has published more than 170 peer-reviewer papers and has chaired several conferences and workshops. He is a member of the Brazilian Computer Society and a Senior Member of the IEEE.

Luciana Arantes received her Ph.D. in Computer Science in 2000 from Paris 6 University (UPMC), France. She is currently an associate processor at UPMC and research member of INRIA/LIP6 Regal project-team. Her research focuses on distributed algorithms for large-scale, heterogeneous, dynamic, or self-organizing environments, such as Grid, peer-to-peer systems, Cloud computing or mobile networks. She is interested in scalability, fault-tolerance, self-organization, load balancing, and latency tolerance issues of distributed algorithms and systems.