A Hierarchical Adaptive Leader Election Algorithm for Crash-Recovery Distributed Systems

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

Allan Edgard Silva Freitas
Federal Institute of Bahia (IFBA)
Salvador, BA, Brazil
allan@ifba.edu.br

Vinicius Fulber-Garcia
Federal University of Parana (UFPR)
Curitiba, PR, Brazil
viniciusfulber@ufpr.br

Elias Procópio Duarte Jr.
Federal University of Parana (UFPR)
Curitiba, PR, Brazil
elias@inf.ufpr.br

ABSTRACT

Leader election is one of the basic building blocks of distributed systems. Multiple different distributed applications employ a leader for decision making or as a coordinator. Traditional leader election algorithms are usually based on all-to-all communications and scales poorly. This work presents a hierarchical adaptive leader election algorithm for distributed systems under the crash-recovery fault model, which allows processes to maintain secondary non-volatile memory. The proposed solution is based on the vCube virtual topology, which presents multiple logarithmic properties, being scalable by definition. One of the contributions of the work is that it is the first to adapt the vCube to the crash-recovery model. The leader is the correct process with the smallest identifier, among those that are most stable, i.e. have failed and recovered the least number of times. The algorithm is adaptive in the sense that processes that change from stable to unstable receive a penalty in order to avoid slowing down the election. Simulation results comparing with the traditional approach show that the proposed solution significantly reduces the number of messages required for leader election, as well as the time to execute a single testing round.

CCS CONCEPTS

• **Computing methodologies → Distributed algorithms**; • **Computer systems organization → Fault-tolerant network topologies**.

KEYWORDS

Dependability, Fault-Tolerance, Unreliable Failure Detectors, Autonomic Systems, Distributed Algorithms

ACM Reference Format:
Luiz Antonio Rodrigues, Allan Edgard Silva Freitas, Vinicius Fulber-Garcia, and Elias Procópio Duarte Jr.. 2024. A Hierarchical Adaptive Leader Election Algorithm for Crash-Recovery Distributed Systems. In *13th Latin-American Symposium on Dependable and Secure Computing (LADC 2024), November 26–29, 2024, Recife, Brazil.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3697090.3697102

LADC 2024, November 26–29, 2024, Recife, Brazil
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1740-6/24/11…$15.00
https://doi.org/10.1145/3697090.3697102

1 INTRODUCTION

Leader election is a fundamental problem in distributed systems, and is employed by multiple applications and protocols. The environments in which the leader is used vary from distributed operating systems to cloud computing systems. The leader's role range from specific tasks such as resource allocation to the orchestration of critical operations. The leader can be responsible for decision-making, act as a coordinator, or even just hold some reference data for other processes. The leader can also be responsible for the coordination of tasks and for ensuring consistency, reliable task execution, and adapting the system to maintain the performance under predefined limits [26]. Furthermore, the leader can simplify communication by providing a central point to disseminate specific messages and decisions.

Informally, the purpose of a distributed leader election algorithm is to ensure that all processes choose a correct process as leader [5]. Furthermore, all processes must choose the *same* leader. When the traditional crash model is adopted, leader election is equivalent to failure detection. The difference is that it returns, instead of the list of suspected processes, a single correct process which is the leader. In the crash-recovery model, processes fail and recover and can also present unstable behavior, toggling states continuously. An unstable process can be as harmful as a faulty process [9]. In the crash-recovery model, an election algorithm seeks not only a correct but also the most *stable* leader. Traditionally, leader election is implemented with a brute-force algorithm, in which every process monitors every other process. That all-monitor-all approach requires a quadratic number of messages per monitoring interval.

This work proposes a hierarchical solution to the leader election problem based on the vCube virtual topology [12]. The vCube is a virtual hypercube when all processes are correct, and the number of processes is a power of 2. Its main characteristic is, however, that as processes fail and recover, the vCube reorganizes itself autonomously, maintaining several logarithmic properties [8]. vCube is also a failure detector. Stein and others [27] have proposed the implementation of a $\diamond P$ failure detector based on the vCube considering the partially synchronous GST (Global Stabilization Time) model. In the GST model, the system is initially asynchronous, but after a certain time instant – called GST – it starts to behave synchronously.

The present work has two main contributions. It is the first to specify the vCube for the *crash-recovery* model. Although vCube always allowed process recovery in the original specifications, this recovery implicitly implies the total loss of the internal state of the failed process except the identifier. In this way, a process that recovers is like a new process, which must catch up by obtaining all the necessary information from other correct processes that have not failed. In the present work, each process that runs vCube has local non-volatile secondary memory and can maintain state information, which can be retrieved when the process recovers after a failure.

The hierarchical adaptive leader election algorithm is built on top of the vCube assuming the failure-recovery model. Each process keeps its number of *incarnations* in non-volatile secondary memory. When the process begins executing the algorithm, it is in its first incarnation. Each time the process fails and recovers, the number of incarnations is increased by one. This way, the number of incarnations reflects how often a process has failed and recovered. The leader election algorithm assumes the partially synchronous GST model so that false suspicions can occur until the system becomes synchronous. The objective of the algorithm is to elect the most "stable" leader, that is, the correct process with the smallest number of incarnations. Thus, the criterion to select the leader is the following: the leader is the process with the smallest identifier among the processes with the smallest number of incarnations.

Furthermore, the algorithm is adaptive in the sense that it applies a penalty to stable processes that start presenting unstable behavior. Consider for example that a given process has 3 incarnations, in a system where the remaining processes have 20 incarnations. That process is the elected leader. Now suppose that the process becomes unstable, failing and recovering continuously. If nothing is done, eventually the number of incarnations will keep growing and become equal to that of the other processes. However, this may take too long to happen. Thus, if the same leader fails and recovers a certain number of times (say, 3 times) then the penalty is applied, and the number of incarnations becomes equal to that of the process with the next smallest number of incarnations. In this way, a previously very stable process becoming unstable does not slow down the election.

We discuss the correctness of the algorithm in terms of the classical properties of leader election: eventual accuracy and eventual agreement. Informally, eventual accuracy determines that every correct process elects a correct process as the leader. Eventual agreement dictates that no two correct processes elect two different processes as leaders. The algorithm was also evaluated through simulation. Results show a significant reduction in the number of messages for the election compared to the brute-force (all-monitor-all) algorithm. It also reduces the execution time of each testing round, although vCube requires more rounds to detect an event and converge with the single elected leader.

The rest of the paper is organized as follows. Section 2 describes the system model. Section 3 presents related work. The proposed algorithm is presented in Section 4. Simulation results are in Section 5, and Section 6 concludes the work.

2 SYSTEM MODEL

A distributed system is a set of processes $\Pi = \{p_0, p_1, ..., p_{N-1}\}$, also referred to by the process identifiers: $\Pi = \{0, 1, ..., N-1\}$. Processes communicate by exchanging messages. The system is fully connected, i.e. the topology can be represented by a complete graph. Thus any process can send a message directly to any other process without having to employ intermediaries. Processes fail and recover according to the crash-recovery model. The most important feature of this model is the fact that processes have local secondary, non-volatile memory to maintain state information. After a process fails and recovers it can retrieve that information.

The communication channels are perfect, thus whenever a correct process sends a message to another correct process, the destination eventually delivers the message without duplication. The primitives to send and receive a message are atomic. The system is defined under the partially synchronous GST model. Thus the system is initially asynchronous but after an unknown instant of time, it becomes and remains synchronous forever. The system is monitored with the vCube failure detector, described below. vCube classifies each process in two possible states: *correct* or *suspected*. A process that has crashed does not respond to any stimulus. Before the GST, a correct but slow process may be *suspected* of having failed.

Leader election can be seen as an abstraction that is in many ways equivalent to failure detectors. The main difference is that while a process invokes a failure detector to obtain the list of suspected processes [28], it invokes the leader election to obtain a single correct process which is the current leader.

Failure detectors were originally defined as an abstraction to allow the execution of consensus in synchronous distributed systems prone to crash faults [6]. Two properties were defined to allow the evaluation of which were the requirements for a fault detector to effectively make consensus possible in that setting: completeness and accuracy. Informally, completeness refers to the fact that eventually, the failure detector will suspect all crashed processes. Accuracy refers to the fact that correct processes are not (incorrectly) suspected by the detector. In practice, completeness is trivial to obtain in real distributed systems. If a process suffers a crash fault, it does not respond to any stimulus and will be detected as soon as the monitoring procedure executes the next round of process monitoring. Accuracy, on the other hand, may happen but is impossible to guarantee. Suppose the performance of some process reduces significantly to the point it slows down every task. That slow process can be easily mistaken as faulty. Thus, accuracy may or may not happen depending on the execution of the detector.

According on how they monitor a process, there are two types of failure detectors: *pull* or *push*. In the *push* model, each correct process periodically sends heartbeat messages to the other processes, informing them that it is fully functional. In the *pull* model, a process sends a response upon receipt of a stimulus, i.e. a test. Tests are run periodically on a testing interval. Processes neither have access to a global clock nor their clocks are synchronized, so testing intervals can differ between processes (despite being nominally identical, e.g. 30 seconds or 10 milliseconds). The concept of a testing round is defined to capture this difference in the testing intervals of the multiple processes. A testing round is defined as the time interval in which all correct processes execute their assigned tests. A testing round is thus as slow as the slowest tester. A test can be implemented as simply as a *heartbeat-request* that must be responded to by a *heartbeat-reply* or consist of multiple requests and replies, involving a set of procedures to be executed by the tested process to check different aspects of its internal state.

Three different strategies to implement pull-based failure detectors are presented in [12]. The first is the most traditional, the brute-force algorithm, in which each correct process tests all other processes at every testing interval. The second strategy is the vRing (virtual Ring) failure detector, in which the processes form a virtual ring. vRing uses the minimum number of tests: N, while brute-force requires $N^2 - N$ tests. On the other hand, the maximum latency for a process on a vRing to correctly identify that some other process has failed/recovered can reach up to N testing rounds. Brute-force is optimal in terms of this metric, always requiring a single testing round for all processes to detect any failure/recovery. Note that false suspicions can also occur, and in this case, the number of tests executed by vRing increases, becoming identical to that of brute-force in the worst case.

The third failure detector is the vCube, according to which processes form a virtual hierarchical topology. However, any process can (in principle) test any other process and the underlying topology must be fully-connected, represented by a complete graph. The virtual edges of a vCube correspond to the tests that the correct processes execute. When the number of processes is a power of 2, and there are no faulty processes or false suspicions, vCube is a hypercube. However, after processes fail, the vCube reorganizes itself, maintaining several logarithmic properties, such as the number of neighbors of each process and the maximum distance between two processes [16].

vCube organizes processes into progressively larger clusters with 2^{s-1} processes, $s = 1, .., \log_2 n$. A cluster is an ordered list of processes. Figure 1 shows the clusters of an 8-process vCube. Function $c_{i,s}$ (Equation 1) returns the ordered list of processes of each cluster, where $\oplus$ is the exclusive *bitwise* operator (*XOR*).

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

Table 1 shows function $c_{i,s}$ for 8 processes. To determine the edges of the virtual topology, for each process i, there is an edge (j, i), such that j is the first correct process in $c_{i,s}$, $s = 1... \log_2 n$. After a process detects that any other process has failed/recovered, its adjacent set of edges (tests) is recomputed. For example, in Figure 1, process p_0 originally tests process p_4 in *cluster* 3, but after p_4 fails, p_0 test p_5, which is the next correct process in the $c_{0,3}$ list.

Table 1: Function $c_{i,s}$ for 8 processes.

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

A process running vCube maintains a counter (*timestamp*) for the number of times the state of each monitored process changes.

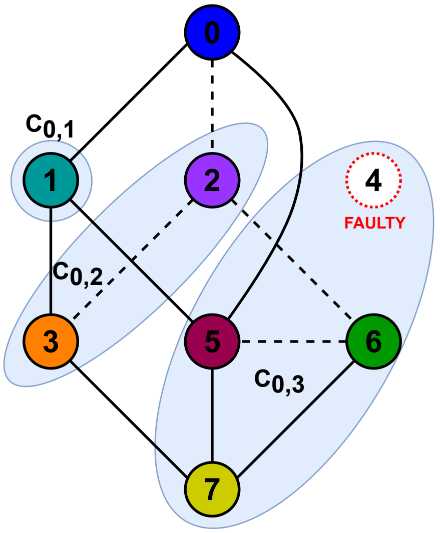


Figure 1: Clusters of a 3-vCube with $2^3 = 8$ processes; because p_4 is faulty, p_0 and p_6 are connected to p_5.

Using the timestamp of a tested process, it is possible to determine each new event that occurs (suspected failure or recovery), thus making it possible to differentiate recent events from older events. Initially, each process is considered correct, and the corresponding timestamp is zero. When there is suspicion, the timestamp is incremented by one unit. Hence, an even value of timestamp indicates that the process is perceived as correct, while an odd value indicates that it is suspected.

vCube is a scalable solution for failure detection, as the maximum number of tests executed is $Nlog_2N$, while the worst case latency is of log_2N testing rounds. vCube has been used to implement a local area network monitoring tool [11], and several distributed abstractions, such as blockchains [14], distributed mutual exclusion [23], dynamic quorums [22], publish-subscribe [7], causal broadcast [24], and atomic broadcast [25].

3 RELATED WORK

Leader election is a classical building block of distributed algorithms. For example, Paxos [19] is one of the best-known consensus algorithms and employs a built-in leader election algorithm. Paxos elects a leader after a series of message exchanges and voting rounds, and the elected leader coordinates the actions of the nodes. Raft [18] is another consensus algorithm that is heavily based on leader election. Raft classifies processes into three roles: leader, follower, and candidate. A leader is elected from among the candidates through a series of message exchanges, votes and timeouts.

Aguilera and others [1] note that using unstable processes as leaders to coordinate actions in a distributed system can degrade performance. The authors define process instability in the crash-recovery model as the number of incarnations (in the original paper, the term the authors use is *epochs*). The number of incarnations of a given process corresponds to the number of times that process has failed and recovered. They propose the use of stability information to elect the leader.

Different criteria have been used to elect the leader. Biswas and others [4] assume a multi-hop system. The criteria for selecting the leader consider CPU and memory capacities as well as connectivity, including the degree and eccentricity of the candidates. Based on the multiple criteria, an ordered list of potential leaders is computed. The NFD-L algorithm for leader election proposed by Reis and others [20] assumes an asynchronous system but also under the crash-recovery model. NFD-L employs a counter of how long a process has remained correct, instead of an incarnation counter. The authors argue that their approach is able to elect a stabler process as leader.

Fernandez and others [13] also use criteria for leader election that reduce the probability of a process becoming a leader at each new incarnation, with the purpose of electing a stabler leader. Biswas and others [3] assigns to each process a rank that is computed based on failure rate and load, electing the process with the lowest failure rate and lowest load. In another recent approach, [31] also uses the number of incarnations to define which nodes are "healthier" and chooses the leader among them. Finally, [15] is another solution that relies on stable storage to store the subsequent incarnations and uses the minimum number of incarnations as the election criterion.

Cachin and others [5] specify the traditional brute-force leader election algorithm for the crash-recovery model, in which all processes monitor all others. The algorithm uses heartbeats, that is, if a heartbeat from a process is not received within a monitoring interval, then it is suspected and is removed from the set of leader candidates. The algorithm also assumes the GST partially synchronous model, and each time a process recovers, the monitoring interval increases, so that when the system becomes synchronous the *timeout* is large enough to receive heartbeats from all correct processes. Although most leader election algorithms are based on the brute-force all-monitor-all strategy, there are others such as the one specified by Santoro [26] that is built on top of an overlay network based on a full hypercube that does not tolerate crashes.

This work proposes a leader election algorithm based on the vCube virtual topology for the crash-recovery model. In comparison with the related work, the use of the vCube is a distinctive feature, allowing scalable leader election with fewer messages and steps compared to brute force approaches. Similar to some of the related approaches (such as [1] and [15]), the vCube solution uses the number of incarnations to determine the stability of the processes. However a penalty is defined for previously stable processes that start to present unstable behavior, in order to speed up the election in those cases.

4 THE HIERARCHICAL LEADER ELECTION ALGORITHM

This section describes the proposed hierarchical algorithm for leader election. As mentioned before, the algorithm relies on the vCube virtual topology and assumes the *crash-recovery* model. According to that fault model, each process maintains non-volatile memory to keep information that it can use after recovering from a failure. The pseudo-code is presented as Algorithm 1. Each process keeps the following variables carrying state information:

- $Correct_i$: List of processes considered correct by process i.
- $timestamp_i[]$: Array of timestamps that each process running vCube maintains for every other process. The timestamp in this case is a state counter. Initially, every process is assumed to be correct and the timestamps are set to zero. Every time the failure or recovery of a process is detected the timestamp is incremented. Thus, an even value indicates that the corresponding process is considered to be correct and an odd value indicates that the process is suspected of having failed. As each process running vCube may receive information about the state of some specific process from multiple processes, timestamps allow vCube to determine whether the information received is new or old.
- $leader_i$: Identifier of the process considered to be the leader by process i.
- $incarnation_i[]$: Each process i keeps a counter of how many times it has failed and recovered, and this information is disseminated as processes are tested. Process i keeps array $incarnation_i[]$ with information about the incarnation of every individual process of the system. This array is employed to as the main criterion to elect a leader.

Initially, all processes elect process zero as the leader (line 7). At each testing round, a process running vCube executes its assigned tests - line 14 - and obtains information from processes tested as correct (line 15). The tester can then update the local arrays that keep the two counters: incarnations and timestamps. In both cases, local information is only modified if a greater value is received. After completing the testing round, the process executes procedure `CheckLeader()` that either maintains the current leader or elects a new leader. The leader is the correct process with the smallest identifier among the processes with the lowest number of incarnations.

After a process recovers from a failure, procedure `recover` is executed (line 46). The process retrieves the information about its last incarnation from secondary memory, increments the counter, and saves the updated value. It then restarts running vCube and the leader election algorithm.

Figure 2 shows an example execution. The vCube has three dimensions and all 8 processes are correct. Each process i tests the processes in clusters $c_{i,s}$, $s = 1, 2, 3$, i.e.

p_0 tests p_1, p_2, p_4;
p_2 tests p_3, p_0, p_6;
p_3 tests p_2, p_1, p_7;
p_4 tests p_5, p_6, p_0;
p_5 tests p_4, p_7, p_1;
p_6 tests p_7, p_4, p_2;
p_7 tests p_6, p_5, p_3.

Whenever p_i tests p_j as correct it obtains new information from p_j. If the number of incarnations of all processes is the same, process p_0 is elected as leader.

In a second example, we assume that the process p_0 has failed. In the first testing round, p_1, p_2 and p_4 test and suspect p_0. In the next round, among other tests, p_3 tests p_1, p_5 tests p_4, and p_6 tests p_4. All those testers receive the information that p_0 is suspected of having failed. Finally, in the third round, p_7 learns about the event from any of the tests it executes. Therefore, all processes receive the information about the failure of p_0 and select p_1 as the leader.

Algorithm 1 Hierarchical Leader Election

```
/* Executed by process $i$ of system $\Pi = \{0, .., n-1\}$ */

procedure INITIALIZATION( )   ▷ Executed only in the first run
  $timestamp_i[p] \leftarrow 0, \forall p \in \Pi$
  $incarnation_i[p] \leftarrow 0, \forall p \in \Pi$
  STORE($incarnation_i[i]$)
  $count\_lead \leftarrow 0$
  STORE($count\_lead$)
  $leader_i \leftarrow 0$
  $recovering \leftarrow false$
  STARTMONITOR( )

procedure STARTMONITOR( )
  $Correct_i \leftarrow \Pi$
  repeat
    for $s \leftarrow 1$ to $\log_2 N$ do
      for all $j \in c_{i,s}$ | $i$ is the first correct process $\in c_{j,s}$ do
        TEST($j$)
        if $j$ is $correct$ then
          Get information from $j$:
              $timestamp_j[]$ and $incarnation_j[]$
          if $timestamp_i[j] \mod 2 = 1$ then
            $Correct_i \leftarrow Correct_i \cup \{j\}$
            $timestamp_i[j]++$
        else
          if $timestamp_i[j] \mod 2 = 0$ then
            $Correct_i \leftarrow Correct_i \setminus \{j\}$
            $timestamp_i[j]++$
    CHECKLEADER( )
    Wait until the next test interval
  forever

procedure CHECKLEADER( )
  $leader_i \leftarrow p \in Correct_i \mid p = min(incarnation_i[p], p)$
  STORE($leader_i$)
  if $recovering = true$ then
    CHECKPENALTY( )

procedure CHECKPENALTY( )
  if $leader_i = i$ then
    if $count\_lead \geq 3$ then
      $candidate \leftarrow p \in Correct_i, p \neq i$
                  $\mid p = min(incarnation_i[p], p)$
      $incarnation_i[i] \leftarrow incarnation_i[candidate] + 1$
      STORE($incarnation_i[i]$)
  else                                   ▷ $i$ is not the leader
    if $count\_lead > 0$ then
      $count\_lead \leftarrow 0$
      STORE($count\_lead$)
  $recovering \leftarrow false$

procedure RECOVER( )   ▷ Executed on process recovery
  $timestamp_i[p] \leftarrow 0, \forall p \in \Pi$
  $incarnation_i[p] \leftarrow 0, \forall p \in \{\Pi \setminus i\}$
  RETRIEVE($incarnation_i[i]$)
  $incarnation_i[i]++$
  STORE($incarnation_i[i]$)
  RETRIEVE($count\_lead$)
  RETRIVE($leader_i$)
  if $leader_i = i$ then
    $count\_lead++$
    STORE($count\_lead$)
  $recovering \leftarrow true$
  STARTMONITOR( )
```

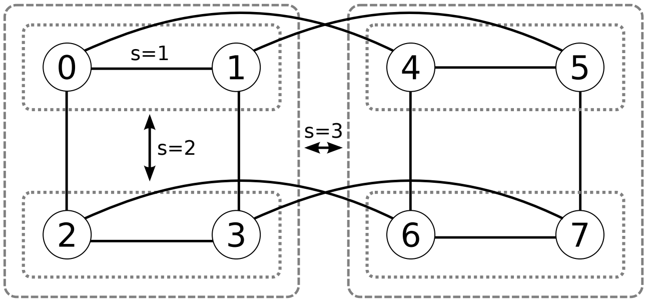


Figure 2: Clusters of a vCube with $2^3 = 8$ processes.

In a final example scenario, we assume that p_0 has recovered from the failure. In this case, its incarnation counter will be incremented to 2. Although p_0 is considered correct by all other processes, p_1 will remain as the leader because its incarnation counter is 1.

Next, we discuss the correctness of the hierarchical adaptive leader election algorithm as well as its time complexity, and the number of messages required.

4.1 Correctness

The classical properties of a leader election algorithm are eventual accuracy and eventual agreement, which we prove next for the proposed algorithm.

Eventual Accuracy According to eventual accuracy, every correct process elects a correct process as the leader. First note that vCube always satisfies the strong completeness property, i.e., every faulty process is eventually suspected. A faulty process has crashed, and thus does not send any message in response to a test request. Thus every crashed process is eventually suspected by every correct process. Therefore, as the election outcome will select a process among those considered to be correct, every correct process always elects a correct process as its leader.

Now consider an unstable process that repeatedly fails and recovers, even after the GST. That process will not be elected, as its incarnation number keeps growing. If there is at least one stable process that does not fail after the GST, its incarnation number will be eventually smaller than that of the unstable processes that fail and recover and it will be elected.

Note that the penalty defined by the algorithm for stable nodes that become consistently unstable reduces the time it takes for the incarnation number of that process to be greater than of another one that might have been unstable, but is now stable.

Eventual agreement. According to eventual agreement, no two correct processes elect two different processes as leaders. The eventual agreement property can only be guaranteed after the GST, i.e., several leaders can be elected before the system exhibits the properties of the synchronous model. The reason is that the lists of correct processes may vary across the different correct processes of the system, as some may raise suspicions that are not raised by others. However, after the GST, the set of correct processes converges to the same set throughout the vCube. Since the leader is chosen based on the process with the lowest identifier among the correct processes with the lowest incarnation numbers, this deterministic criterion ensures that all correct processes will eventually agree on the same leader.

5 SIMULATION RESULTS

This section presents an evaluation of the proposed hierarchical adaptive leader election algorithm conducted with simulation. First, we describe the environment, parameters, and metrics. Then, we present and discuss the results obtained. Our experiments compare the vCube to the classical all-monitor-all solution (ALL) [5], where each correct process executes tests on all other processes in all rounds.

5.1 Simulation Environment

The proposed distributed and adaptive leader election algorithm was implemented with the Neko [30] simulation tool, a Java framework[1], developed to enable the simulation of distributed algorithms. The Neko architecture is organized in two main levels: application and network. An application is built in the form of micro-protocols. The micro-protocols are employed by processes, which are instances of the class NekoProcess. At the application level, processes communicate by exchanging messages. Messages are sent and received using methods send and deliver. The second component of the Neko architecture is the network, which can be either a simulated network or a real network. In this work, the experiments were obtained with a simulated network RandomNetwork, which employs a `lambda` parameter to generate variable transmission delays following an exponential distribution. Using a particular seed ensures reproducibility.

An approach was proposed by Rodrigues [21] to simulate process faults in Neko. It is possible to define an interval in which some process will crash using the regular Neko configuration file. Consider that an application sends messages to the crashed process. In that case, the failure simulation support class checks whether the process has crashed and, if this is the case, the message is discarded. The same applies to messages received from the network.

5.2 Performance Metrics

The performance of distributed algorithms is usually measured by two metrics: execution time and number of messages generated [29]. In this section, we first evaluate the latency of the proposed distributed adaptive leader election algorithm considering both systems without and with faults. In a scenario with no faults, the latency is the time interval in which the monitoring information including the incarnation number is propagated throughout the vCube to all correct processes. In the scenario with a crash fault, the latency is the time it takes for all correct processes to detect the occurrence of a fault and elect the leader (which may result in keeping the same leader).

In the second scenario, we also evaluated for how long the system remained inconsistent, i.e. after the failure of the leader, we measured the time it took until the new leader was elected by all correct processes. A similar case happens when a process with the smallest number of incarnations recovers and until but has not yet been elected by the other correct processes. This case was not explicitly evaluated in the simulation, as the length of inconsistent states (which was measured) gives the same results.

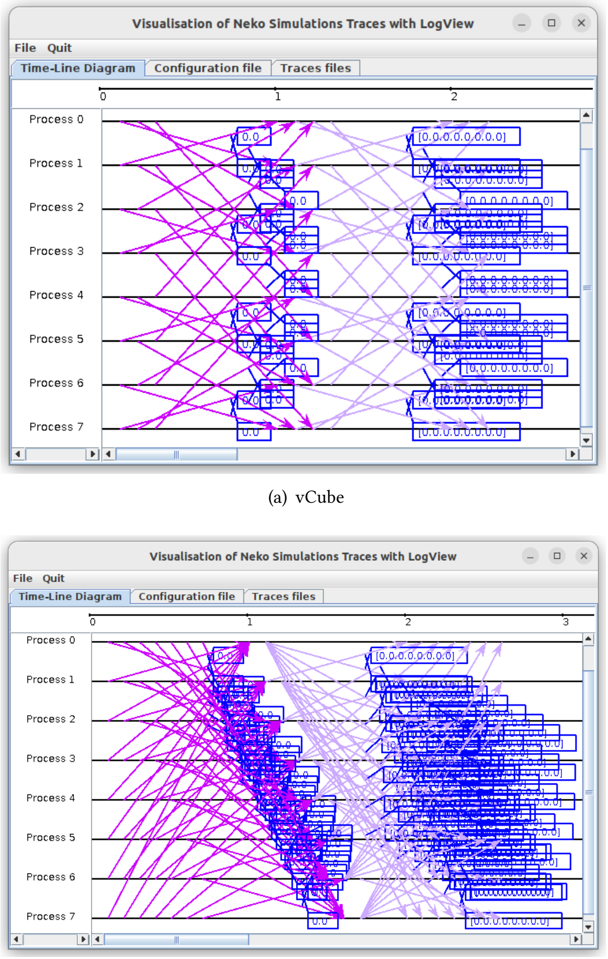
For each scenario, systems with $n = 2^d$ processes were used, for $d = 3, 4, .., 9$, i.e. $n = 8, 16, .., 512$. The failure parameters are described in the following scenarios, where applicable.

As mentioned before, the Neko network model used was RandomNetwork. For each message sent, a transmission time of 0.1 time intervals plus a random transmission time component generated with an exponential distribution with parameter lambda=0.2. This means that the total time interval between sending the message and its delivery by the receiver is at least 0.1 time intervals, plus a random value. However, for successive messages sent, the transmission time is shifted by 0.1 (as shown in Figure 3). Each scenario was executed 31 times and the results show the mean and confidence interval of 95%.

5.3 Fault-free Scenario

In the first scenario, none of the processes fail. Thus it suffices to execute a single round of tests was performed by each algorithm for comparison purposes. In Figure 4 it is possible to observe (a) the execution time and (b) the number of messages required by vCube and ALL. Each test performed by a process on another consists of a test request (REQUEST) and a response (REPLY). The messages are presented on a logarithmic scale and are also shown in Table 2. The execution time varies for both algorithms, but it is clear that the increase is higher for ALL as the number of processes increases. The number of messages is significantly higher for ALL (N^2), while vCube sends $N \log_2 N$ messages per testing round.

[1] Neko is available at https://github.com/arluiz/neko



(a) vCube

(b) ALL (All-to-all)

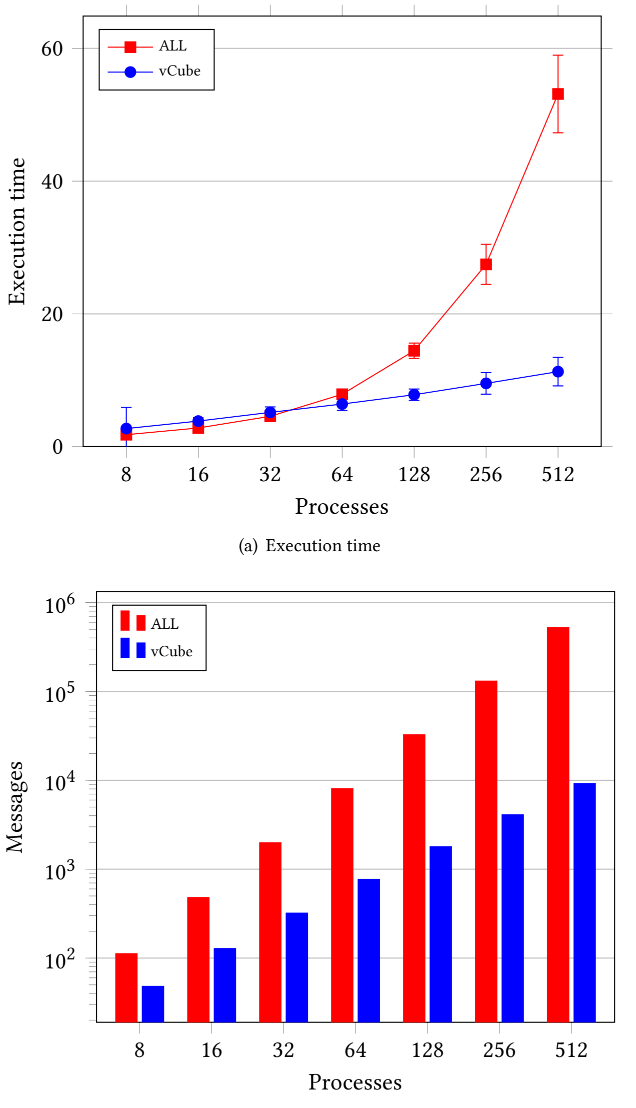
Figure 3: Messages exchanged as tests are executed by both vCube and ALL with 8 processes.

Table 2: Number of messages exehanged in fault-free scenarios.

Processes	ALL	vCube	Balance
8	112	48	57.14%
16	480	128	73.33%
32	1,984	320	83.87%
64	8,064	768	90.48%
128	32,512	1,792	94.59%
256	130,560	4,096	96.86%
512	523,264	9,216	98.28%

5.4 Scenarios with Crashes

To simulate a crash fault, the Neko failure signal is sent for process 0 at time 0.0 of the simulation. In this case, due to the number of rounds it takes for both strategies (vCube and ALL) to allow all correct processes to detect the crash, the simulation was of $\log_2(n)$ rounds for vCube and two rounds for ALL, corresponding to the worst-case latency of each solution. The testing interval between was set to 100.0 time units for both algorithms.



(a) Execution time

(b) Total Messages

Figure 4: Fault-free execution based on the Neko RandomNetwork network model.

Since a single crash was simulated, the execution time is very similar to that of the scenario without failures. The same is true for the number of messages, although it is slightly lower as a crashed process does not execute any test. Figure 5 illustrates the behavior of vCube and ALL when detecting the failure of process p_0 in a system with $n = 8$ processes. Initially, processes p_6, p_7, p_3 and p_5 do not detect the failure because they do not test process p_0 directly and therefore they keep process 0 as the leader. On the other hand, processes p_1, p_2 and p_4 directly test process p_0 (Figure 2) and detect its failure in the first round. At this point, there is an inconsistency in the system. The inconsistency remains until the second round, which starts at time 100, in which p_6, p_7, p_3 and p_5 detect the failure of p_0 based on the tests they execute on their neighbors, and thus elect p_1 as the new leader.

Thus, only after all correct processes either test the crashed process or obtain information about the crash from other processes tested correct p_1 is elected leader by all correct processes. Since ALL requires all processes to test all other processes, the the latency is always the lowest (optimal). Depending on whether the crash happens in the begining or the middle of a round, the latency of ALL ranges from one to two testing rounds.

```
#vCube
0,000 p0 messages 0.0 p0 crash started!
1,390 p6 messages Leader remains: 0
1,392 p7 messages Leader remains: 0
1,676 p3 messages Leader remains: 0
2,547 p5 messages Leader remains: 0
4,000 p1 messages suspect 0
4,202 p2 messages suspect 0
4,933 p4 messages suspect 0
4,933 p4 messages New Leader: 1
5,241 p2 messages New Leader: 1
7,359 p1 messages New Leader: 1
...
100,360 p5 messages suspect 0
100,783 p7 messages suspect 0
100,871 p6 messages suspect 0
101,060 p3 messages suspect 0
101,088 p5 messages New Leader: 1
101,386 p6 messages New Leader: 1
101,748 p7 messages New Leader: 1
102,420 p3 messages New Leader: 1
```

```
#ALL
0,000 p0 messages 0.0 p0 crash started!
4,100 p1 messages suspect 0
4,100 p1 messages New Leader: 1
4,100 p2 messages suspect 0
4,100 p2 messages New Leader: 1
4,100 p3 messages suspect 0
4,100 p3 messages New Leader: 1
4,100 p4 messages suspect 0
4,100 p4 messages New Leader: 1
4,100 p5 messages suspect 0
4,100 p5 messages New Leader: 1
4,100 p6 messages suspect 0
4,100 p6 messages New Leader: 1
4,100 p7 messages suspect 0
4,100 p7 messages New Leader: 1
```

Figure 5: Execution log of the first testing rounds after the crash of process p_0 ($n = 8$).

Figure 6 shows the latency for the detection of a crash fault that occurred at time 0.0. The latency is the time it takes from the failure to its detection by all correct processes. Since process 0 is the first to be tested by all processes in the ALL, the time is proportional to the number of processes and also depends on the timeout (4 time units). The latency would be higher had the faulty process been the last to be tested. In any case, the ALL strategy guarantees that the detection in all cases (All-Latency) would occur in one, or at most two testing rounds. A second round is only required if the fault occurs halfway through the testing round (All-Total).

The latency of vCube is directly related to the hierarchical testing mechanism. Thus, in the first round, only those processes that are virtually connected to the faulty process detect the crash. In the second round, the neighbors that are two hops away learn about the crash, and so on. Table 3 shows the number of messages in crash scenarios. As with the non-fault scenarios, the number of test messages increases with the number of processes, with a considerably steeper increase for ALL.

Table 3: Number of messages in scenarios where p_0 crashes.

Processes	ALL	vCube	Balance
8	182	129	29.12%
16	870	488	43.91%
32	3.782	1.565	58.62%
64	15.750	4.560	71.05%
128	64.262	12.481	80.58%
256	259.590	32.688	87.41%
512	1.043.462	82.845	92.06%

Thus, although the number of messages required by the ALL strategy is much higher than vCube's (similar to the fault-free scenario), vCube's latency is always slightly higher than ALL's – vCube requires $log_2 N$ rounds in the worst case. Note that the latency to detect the recovery of process is the same, as the event information propagation follows the same strategy.

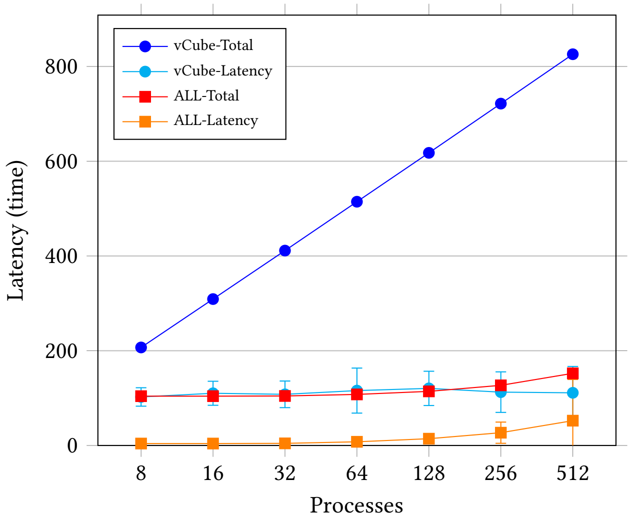


Figure 6: Latency to detect a failure of process p_0 at time 0.0.

5.5 Evaluating the Penalty Impact

To evaluate the impact of the penalty on newly unstable processes, consider the scenario in which process zero has an initial number of incarnations equal to zero, and the other processes have random incarnation values between 10 and 20 (Figure 7). Thus process zero had been very stable, while the others were unstable. The situation changes, and process zero starts a sequence of multiple failures/recoveries. In a scenario without the proposed penalty, process zero will continue to be elected leader until its number of incarnations exceeds that of another process with fewer incarnations, that is 10 incarnations. With the penalty, after three incarnations in sequence (lines 4, 7 and 10) process zero will adjust its number of incarnations (line 15). So, after another incarnation (line 18), the process (process 1) is finally elected (line 21). Note that the number of subsequent incarnations proposed by the algorithm to apply the penalty (3 in this case) is a parameter that can be changed as necessary.

```
1  #vCube - penalty impact
2  1,900 p0 messages e r p1 p0 I_AM_ALIVE [0, 10, 18, 19,
       17, 15, 13, 11]
3  ...
4  1,000 p0 messages 1.0 p0 crash started!!!!!
5  10,000 p0 messages crash stoped at crash-VCubeFD
6  ..
7  104,000 p0 messages 104.0 p0 crash started!!!!!
8  111,000 p0 messages crash stoped at crash-VCubeFD
9  ..
10 207,000 p0 messages 207.0 p0 crash started!!!!!
11 215,000 p0 messages crash stoped at crash-VCubeFD
12 ..
13 221,100 p0 messages e r p4 p0 I_AM_ALIVE [2, 10, 18, 19,
       17, 15, 13, 11]
14 221,100 p0 messages Leader remains: 0
15 221,100 p0 messages Penalty applied: new incarnation 11
16 ...
17 302,000 p0 messages 302.0 p0 crash started!!!!!
18 307,000 p0 messages crash stoped at crash-VCubeFD
19 ..
20 344,300 p0 messages e r p4 p0 I_AM_ALIVE [11, 10, 18, 19,
        17, 15, 13, 11]
21 344,300 p0 messages New Leader: 1
```

Figure 7: Execution log with the penalty approach ($N = 8$).

Figure 8 shows the impact of latency in the scenario in which the penalty is applied, considering the time required for process zero to stop being elected the leader after it becomes unstable. In the figure, process zero fails and recovers at random time instants defined along 100 simulation-time steps.

6 CONCLUSION

This work presented a hierarchical algorithm for leader election in distributed systems. The algorithm is defined on the vCube virtual topology, which adaptively reorganizes itself after process failures maintaining several logarithmic properties. The proposed algorithm assumes the *crash-recovery* model, i.e. a process can keep a counter in secondary memory indicating how many times it has failed and recovered. The algorithm employs that counter to elect one of the most stable processes as leader. A penalty is applied after stable processes fail and recover a pre-configured number of times, i.e. present unstable behavior. The purpose of the penalty is to avoid slowing down the election. The algorithm was implemented with simulation and compared with the traditional all-monitor-all approach. Results show that the proposed algorithm is efficient and scalable, and requires a significantly lower number of tests (messages).

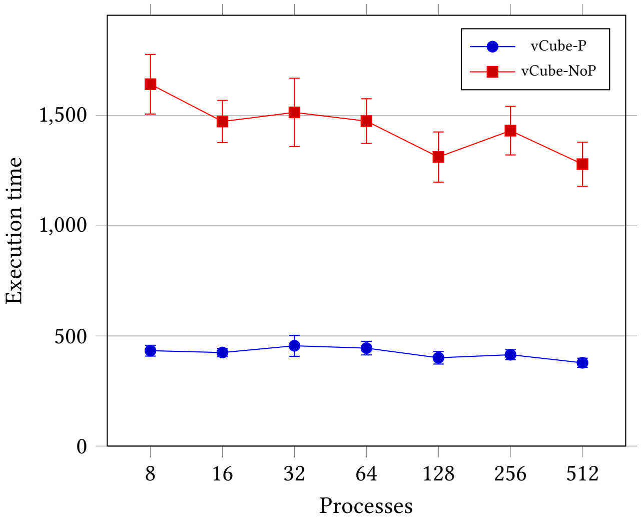


Figure 8: Penalty comparison (vCube-P) with the regular approach (vCube-NoP).

Future work includes adapting the algorithm to dynamic distributed systems where the composition of the system changes over time. Another future work is to define the algorithm for partitionable networks with general topologies [10], and the Byzantine fault model [2]. Furthermore, the investigation of intelligent strategies to improve leader election should also be considered [17].

ACKNOWLEDGMENTS

This work was partially supported by the Brazilian Research Council (CNPq - Conselho Nacional de Desenvolvimento Científico e Tecnológico) grant 308959/2020-5; FAPESB (Fundação de Amparo a Pesquisa do Estado da Bahia) grant TIC0004/2015; and CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil) - Finance Code 001.

REFERENCES

[1] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. 2000. Failure detection and consensus in the crash-recovery model. *Distributed computing* 13, 2 (2000), 99–125.

[2] Luiz Carlos Pessoa Albini, Elias Procópio Duarte Jr., and Roverli Pereira Ziwich. 2005. A generalized model for distributed comparison-based system-level diagnosis. *Journal of the Brazilian Computer Society* 10 (2005), 44–56. https://doi.org/10.1007/BF03192365

[3] Amit Biswas, Ashish Kumar Maurya, Anil Kumar Tripathi, and Samir Aknine. 2021. Frlle: a failure rate and load-based leader election algorithm for a bidirectional ring in distributed systems. *The Journal of Supercomputing* 77 (2021), 751–779.

[4] Amit Biswas and Anil Kumar Tripathi. 2021. Preselection based leader election in distributed systems. In *International Symposium on Intelligent and Distributed Computing*. Springer, 261–271.

[5] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. 2011. *Introduction to reliable and secure distributed programming.* Springer Science & Business Media.

[6] Tushar Deepak Chandra and Sam Toueg. 1996. Unreliable Failure Detectors for Reliable Distributed Systems. *J. ACM* 43, 2 (March 1996), 225–267. https://doi.org/10.1145/226643.226647

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

[8] Elias P Duarte, Luiz CP Albini, Alessandro Brawerman, and Andre LP Guedes. 2009. A Hierarquical Distributed Fault Diagnosis Algorithm Based on Clusters with Detours. In *The 6th IEEE Latin American Network Operations and Management Symposium.* IEEE, 1–6.

[9] Elias P Duarte, Thiago Garrett, Luis CE Bona, Renato Carmo, and Alexandre P Züge. 2010. Finding stable cliques of PlanetLab nodes. In *DSN 2010*. IEEE, 317–322. https://doi.org/10.1109/DSN.2010.5544300

[10] Elias Procopio Duarte, Andrea Weber, and Keiko VO Fonseca. 2011. Distributed diagnosis of dynamic events in partitionable arbitrary topology networks. *IEEE Transactions on Parallel and Distributed Systems* 23, 8 (2011), 1415–1426. https://doi.org/10.1109/TPDS.2011.284

[11] Elias Procópio Duarte Jr. and LC Erpen De Bona. 2002. A dependable SNMP-based tool for distributed network management. In *Proc. International Conference on Dependable Systems and Networks*. IEEE, 279–284. https://doi.org/10.1109/DSN.2002.1028911

[12] Elias P Duarte Jr, Luiz A Rodrigues, Edson T Camargo, and Rogério C Turchetti. 2023. The missing piece: a distributed system-level diagnosis model for the implementation of unreliable failure detectors. *Computing* 105, 12 (2023), 2821–2845. https://doi.org/10.1007/s00607-023-01211-8

[13] Christian Fernández-Campusano, Mikel Larrea, Roberto Cortiñas, and Michel Raynal. 2017. A distributed leader election algorithm in crash-recovery and omissive systems. *Inform. Process. Lett.* 118 (2017), 100–104.

[14] Allan Edgard Silva Freitas, Luiz Antonio Rodrigues, and Elias Procópio Duarte Jr. 2024. vCubeChain: A scalable permissioned blockchain. *Ad Hoc Networks* 158 (2024), 103461. https://doi.org/10.1016/j.adhoc.2024.103461

[15] Carlos Gómez-Calzado, Mikel Larrea, Iratxe Soraluze, Alberto Lafuente, and Roberto Cortiñas. 2013. An Evaluation of Efficient Leader Election Algorithms for Crash-Recovery Systems. In *2013 21st Euromicro*. IEEE, 180–188. https://doi.org/10.1109/PDP.2013.33

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

[17] Bogdan Tomoyuki Nassu, Elias Procópio Duarte, and Aurora T. Ramirez Pozo. 2005. A comparison of evolutionary algorithms for system-level diagnosis. In *Proc. 7th Annual Conference on Genetic and Evolutionary Computation* (Washington DC, USA) *(GECCO '05)*. Association for Computing Machinery, New York, NY, USA, 2053–2060. https://doi.org/10.1145/1068009.1068350

[18] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (Philadelphia, PA) *(USENIX ATC'14)*. USENIX Association, USA, 305–320.

[19] M. Pease, R. Shostak, and L. Lamport. 1980. Reaching Agreement in the Presence of Faults. *J. ACM* 27, 2 (apr 1980), 228–234. https://doi.org/10.1145/322186.322188

[20] Vinícius A. Reis and Gustavo M. D. Vieira. 2017. Quality of Service of an Asynchronous Crash-Recovery Leader Election Algorithm. In *Anais do XXXV Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos* (Belém). SBC, Porto Alegre, RS, Brasil. https://sol.sbc.org.br/index.php/sbrc/article/view/2685

[21] Luiz Antonio Rodrigues. 2006. *Extensão do suporte para simulação de defeitos em algoritmos distribuídos utilizando o Neko.* Master's thesis. UFRGS.

[22] Luiz A. Rodrigues, Luciana Arantes, and Elias P. Duarte. 2016. An Autonomic Majority Quorum System. In *2016 IEEE 30th International Conference on Advanced Information Networking and Applications (AINA)*. IEEE, 524–531. https://doi.org/10.1109/AINA.2016.73

[23] Luiz A. Rodrigues, Elias P. Duarte, and Luciana Arantes. 2018. A distributed k-mutual exclusion algorithm based on autonomic spanning trees. *JPDC* 115 (2018), 41–55. https://doi.org/10.1016/j.jpdc.2018.01.008

[24] Luiz A. Rodrigues, Elias P. Duarte, João Paulo de Araujo, Luciana Arantes, and Pierre Sens. 2018. Bundling Messages to Reduce the Cost of Tree-Based Broadcast Algorithms. In *2018 Eighth Latin-American Symposium on Dependable Computing (LADC)*. ACM, 115–124. https://doi.org/10.1109/LADC.2018.00022

[25] Lucas V Ruchel, Edson Tavares de Camargo, Luiz Antonio Rodrigues, Rogério C Turchetti, Luciana Arantes, and Elias Procópio Duarte Jr. 2024. Scalable atomic broadcast: A leaderless hierarchical algorithm. *JPDC* 184 (2024), 104789. https://doi.org/10.1016/j.jpdc.2023.104789

[26] Nicola Santoro. 2006. *Design and analysis of distributed algorithms.* John Wiley & Sons.

[27] Gabriela Stein, Luiz Antonio Rodrigues, Elias ProcóPio Duarte Jr., and Luciana Arantes. 2023. Diamond-P-vCube: An Eventually Perfect Hierarchical Failure Detector for Asynchronous Distributed Systems. In *Proceedings of the 12th Latin-American Symposium on Dependable and Secure Computing* (La Paz, Bolivia) *(LADC '23)*. Association for Computing Machinery, New York, NY, USA, 40–49. https://doi.org/10.1145/3615366.3615420

[28] Rogerio C Turchetti and Elias Procopio Duarte. 2015. Implementation of failure detector based on network function virtualization. In *2015 IEEE International Conference on Dependable Systems and Networks Workshops*. IEEE, 19–25. https://doi.org/10.1109/DSN-W.2015.30

[29] P. Urban, X. Defago, and A. Schiper. 2000. Contention-aware metrics for distributed algorithms: comparison of atomic broadcast algorithms. In *9th Int'l Conf.on Computer Communications and Networks (ICCCN)*. 582–589. https://doi.org/10.1109/ICCCN.2000.885548

[30] P. Urban, X. Defago, and A. Schiper. 2001. Neko: a single environment to simulate and prototype distributed algorithms. In *15th Int'l Conf. Info. Networking*. IEEE, 503–511. https://doi.org/10.1109/ICOIN.2001.905471

[31] Jiangran Wang and Indranil Gupta. 2023. Churn-Tolerant Leader Election Protocols. In *43rd IEEE ICDCS*. IEEE, 96–107.