Quality of Service of an Asynchronous Crash-Recovery Leader Election Algorithm

Vinícius A. Reis[1] **, Gustavo M. D. Vieira**[1]

[1]DComp – CCGT – UFSCar
Sorocaba, São Paulo, Brasil

angiolucci@gmail.com, gdvieira@ufscar.br

Abstract. *In asynchronous distributed systems it is very hard to assess if one of the processes taking part in a computation is operating correctly or has failed. To overcome this problem, distributed algorithms are created using unreliable failure detectors that capture in an abstract way timing assumptions necessary to assess the operating status of a process. One particular type of failure detector is a leader election, that indicates a single process that has not failed. The unreliability of these failure detectors means that they can make mistakes, however if they are to be used in practice there must be limits to the eventual behavior of these detectors. These limits are defined as the* quality of service (QoS) *provided by the detector. Many works have tackled the problem of creating failure detectors with predictable QoS, but only for crash-stop processes and synchronous systems. This paper presents and analyzes the behavior of a new leader election algorithm named NFD-L for the asynchronous crash-recovery failure model that is efficient in terms of its use of stable memory and message exchanges.*

1. Introduction

Fault-tolerant distributed systems are created by the aggregation of many non fault-tolerant computer systems in clusters, coordinated by fault-tolerant software. These commodity clusters are modeled as *asynchronous* systems where there are no bounds to the message transmission delays and processing time. As a consequence, it is very hard to assess if one of the processes taking part of a computation is operating correctly or has failed. To overcome this limitation, many distributed algorithms assume the stronger computational model of asynchronous processes augmented with unreliable failure detectors [Chandra and Toueg 1996]. More than just flagging failed processes, unreliable failures detectors capture in an abstract way timing assumptions necessary to the correct operation of many distributed algorithms [Lamport 1998, Chandra and Toueg 1996]. The unreliability of these detectors is key to this abstraction: errors can be made and failures are detected eventually, in a way that reflects the timing uncertainties intrinsic to asynchronous distributed systems.

Unreliable as they are, failure detectors are enough to design correct distributed algorithms [Chandra et al. 1996]. However, if they are to be used in practice there must be limits to the eventual behavior of these detectors. These limits are properties of the distributed system (processes and network links), of the failure detector algorithm used and of its operational parameters, defining the *quality of service (QoS)* provided by the detector [Chen et al. 2002]. The QoS of a failure detector can have a direct impact on

the performance of a distributed algorithm. For example, the Paxos algorithm [Lamport 1998] uses a single coordinator process as the sequencer that orders and distributes a set of totally-ordered messages among a group of processes. The selection of the coordinator is made by a failure detector and the progress of the algorithm is halted while the coordinator is failed. If this coordinator is *wrongly* assumed to be failed by the failure detector, Paxos throughput is affected, even if the mistake is eventually corrected. The more mistakes the failure detector makes, the worse Paxos real world performance will be. Thus, QoS of failure detectors is an important parameter to be considered.

Many works have tackled the problem of creating failure detectors with predictable QoS. The seminal work of Chen et al. [Chen et al. 2002] defined a set of QoS metrics, created a new failure detector with a precise model of its QoS and used this model to configure its parameters to match a desired QoS. Other works have experimentally studied the QoS of failure detectors [Falai and Bondavalli 2005, Nunes and Jansch-Porto 2004], created more robust QoS models [Sotoma et al. 2006] and proposed application-specific QoS for a system-wide failure detector service [Hayashibara et al. 2004]. All these algorithms and QoS models were created assuming the crash-stop process abstraction, where processes can only fail by crashing and once crashed they never return to the computation. The only exception is the work of Ma et. al. [Ma et al. 2010] that tackles the problem of analyzing the QoS of Chen's algorithm in the crash-recovery process abstraction, where processes fail by crashing but later recover preserving their stable memory.

The works regarding QoS of failure detectors also share another, rather ironical, property: many assume a *synchronous* system where there is a (possibly unknown) bound to the message transmission delays and processing times. In this model it is easier to create models of QoS, usually based on the existence of synchronized clocks. Despite being a reasonable assumption in current systems, take for example the use of NTP in clusters, these algorithms are burdening the system designer with a more stringent synchrony model than the one required by the algorithm itself. One exception is one of the algorithms proposed by Chen et. al. [Chen et al. 2002], which in its turn has weaker QoS model based on the accuracy of a *predictor*. For asynchronous systems, the ability of the failure detector to ascertain the state of a process based on the history of its past communications is very important for its QoS, as shown by [Nunes and Jansch-Porto 2004] and explored by [Hayashibara et al. 2004]. Thus, there is no robust QoS model for asynchronous systems beyond the model originally proposed by Chen et al..

Moreover, no previous work has analyzed the QoS of a leader election algorithm. A leader election is a type of failure detector that instead of indicating if a single process has failed, it indicates a single process that has *not* failed, with the identity of this process agreed by all correct processes. In some senses, this type of failure detector consumes less system resources because it monitors only a single process [Larrea et al. 2000] and it is the weaker form of failure detector required to solve consensus [Chandra et al. 1996]. Paxos, a consensus algorithm, requires a leader election for asynchronous systems composed by crash-recovery processes. There are suitable leader election algorithms that support crash-recovery processes [Martín et al. 2009], but unfortunately there are no known algorithms with these properties that have a suitable model for their QoS.

This paper presents a new leader election for the asynchronous crash-recovery failure model that is efficient in terms of its use of stable memory and message exchanges.

Our algorithm named NFD-L is an extension of Chen's algorithm and retains its QoS properties in the absence of failures. We analyze the QoS of this algorithm and present experimental data as a first step to providing a complete QoS model for this failure detector in the presence of failures and recoveries.

2. Preliminaries

In this section we will describe the basic concepts of distributed systems and the relationship between its abstractions. We then use those abstractions to present more sophisticated concepts, such as failure models and failure detectors.

2.1. Synchrony and Failure Models

Asynchronous distributed systems have no bounds for (i) how much time it takes for a message to be delivered and (ii) how much time it takes for a process to do some computation. *Synchronous* distributed systems can rely on hard bounds for message delay and computing time. *Partially synchronous* distributed system are systems where processes and links behaves most of the time asynchronously, but there is an unknow time in the future after which it behaves synchronously [Cachin et al. 2011].

Another property that defines a distributed system is process failure models. The *crash-stop* model considers *correct* a process that never crashes. Once crashed, a process is considered *faulty* and it never recovers. The *crash-recovery* failure model considers correct a process that never crashes or crashes and recovers a finite number of times. Thus, in the *crash-recovery* model a faulty process is a process that crashes and never recovers or a process that crashes and recovers infinitely [Cachin et al. 2011].

In this paper we consider asynchronous processes augmented with unreliable failure detectors. This is a type of partially synchronous distributed system where the timing assumptions are confined to the failure detector. We also assume crash-recovery processes that communicate with each other by exchanging messages through links. The links may drop or delay messages, but only deliver a message previously sent by some process.

2.2. Failure Detection and Leader Election

Given a distributed system composed by processes and links, a failure detector is a component that outputs not necessarily correct information about which processes are correct or faulty. It works as a local component, queried by processes in order to know about which processes are still correct. When a process q queries its local failure detector about the state of process p, the only two possible responses it can receive are either *trust*, if the failure detector trusts process p is correct or *suspect*, if the failure detector suspects p is faulty. Failure detectors are usually implemented by exchanging messages through its links [Chen et al. 2002]: a process p sends a heartbeat message to another process q every η time units, if q receives no messages from p after a timeout plus a safety margin α, the failure detector in q will start suspecting p may be crashed.

An important result is that it is possible to build reliable distributed systems on top of an unreliable failure detector [Chandra and Toueg 1996]. It means a failure detector is not supposed to be correct while the system behaves asynchronously [Fischer et al. 1985], it may make mistakes by suspecting correct processes or trusting faulty ones. However, assuming a partially synchronous system, eventually the processes and links will behave

synchronously and then the failure detector will stop making mistakes. Based on these assumptions, reliable distributed systems are designed to be *safe* when the system behaves asynchronously and only *progress* when it behaves synchronously [Guerraoui 2000]. Despite the mistakes it can make, a failure detector is a powerful abstraction as it encapsulates unpredictable system behavior.

A failure detector is specified in terms of two properties: *completeness* and *accuracy* [Chandra and Toueg 1996]. Completeness is the property that describes how well a failure detector will perceive real failures, while accuracy is the property that describes how well it will avoid mistakes [Reynal 2005], e.g., false detections. Failures detectors will behave differently and support other failures models by simply strengthening or loosening completeness and accuracy [Guerraoui 2000].

Of special interest for this work is the Ω failure detector, originally presented in [Chandra et al. 1996]. The Ω failure detector is a *leader election*, a failure detector that outputs a single trusted process. Formally, the Ω failure detector is specified by the following properties [Guerraoui 2000]:

- *Eventual Accuracy*: There is a time after which every correct process trusts some correct process.
- *Eventual Agreement*: There is a time after which no two correct processes trust different processes.

This pair of properties ensures every correct process will eventually trust the same correct process. The eventual behavior means it is necessary a long enough period of synchrony in order for the properties to be achieved. The Ω failure detector is used as a building block to solve more complex problems such as consensus [Lamport 1998] and atomic broadcast [Chandra et al. 1996].

It is rather easy to create a leader election using regular failure detection. Let each process in a distributed system use a failure detector to monitor every other process and create a set C of processes it believes to be correct. Eventually the set C will be the same in all correct processes and the leader can be chosen as the process in C with the smallest pid. However, this reduction is very costly in terms of heartbeat messages, as it requires $N^2 - N$ heartbeat messages for each η, one for each unidirectional communication link. Moreover, this reduction doesn't provide a very useful property: *leader stability*. Leader stability is the ability of the failure detector to never remove the leadership from a correct process because another process (with a lower pid, for instance) begins to be trusted [Malkhi et al. 2005]. Leader stability is very useful for Paxos, for example, because a leadership change in this algorithm can be a costly operation [Vieira et al. 2014].

2.3. QoS of Failure Detection with Crash-Stop Failures

A set of failure detectors with a model for quality of service was first proposed by Chen et al. [Chen et al. 2002]. In that work, the authors presented the concept of QoS for failure detectors, alongside quantitative metrics to measure it. They also presented a set of failure detectors that were designed to have a precise model of the behavior of these metrics. Roughly speaking, quality of service means the failure detector is configured to meet strict application requirements and work accordingly to the network probabilistic behavior.

The failure detector proposed by Chen et al. has two main components: an *estimator* and a *configurator*. The estimator is responsible for analyzing the network probabilistic behavior in terms of message losses and message delays. The configurator is responsible for generating a suitable failure detector based on the behavior observed by the estimator and on the QoS requirements provided by the application developer. The application developer must specify the application requirements in terms of three metrics:

- Detection time (T_D): This measures how much time elapses from the occurrence of a crash and it being detected by the failure detector;
- Mistake recurrence time (T_{MR}): This measures the time between two consecutives mistakes made by the failure detector;
- Mistake duration (T_M): This measures how much time the failure detector takes to correct itself once it has made a mistake.

The result is a customized failure detector in which both heartbeat inter-sending interval η and the safety margin α are shaped to meet the given constraints, when it is feasible. Once configured, a failure detector is ready to work as usual: a monitored process p sends every η time units a heartbeat message to a monitor process q, which will wait for those messages for a specific time, plus a safety margin α.

In this paper we will discuss only one of the failure detectors proposed by Chen et al., namely the *New Failure Detector with Expected Arrival Time Calculation (NFD-E)*. The NFD-E algorithm is the most suitable for partially synchronous distributed systems as it assumes no clock synchrony among the processes and estimates the arrival time of future heartbeats. This algorithm is particularly interesting because its QoS model is based only on the variance of the message propagation delays and not on the unknown delays themselves. However there is some limitations about the required assumptions needed to this algorithm work properly. It assumes a crash-stop failure model and, although there is no assumption about synchrony among processes, it is necessary that each process has access to a local clock. Furthermore, it is assumed there is no clock drift between any two local clocks.

The NFD-E algorithm works as follows: for all $i \geq 1$, a monitored process p sends at time $i \cdot \eta$ a heartbeat message m_i to a monitor process q. Also for all $i \geq 1$, process q waits for message m_i until a *freshness point* $\tau_i = EA_i + \alpha$. If no message with a label equals or grater than i is received by q until τ_i expires, i.e., no fresh message arrives before its timeout, then q starts suspecting p. To estimate each arrival time EA_i, process q uses a *predictor* based on the set of n previously received messages from p. Let $m'_1, ..., m'_n$ be the messages received by q, $s_1, ..., s_n$ be the sequence numbers of such messages, $A'_1, ..., A'_n$ be their receipt time according to q's local clock and ℓ the largest sequence number among all $s_1, ..., s_n$. Then $EA_{\ell+1}$ is estimated by [Chen et al. 2002]:

$$EA_{\ell+1} \approx \frac{1}{n} \left(\sum_{i=1}^{n} A'_i - \eta s_i \right) + (\ell + 1)\eta \qquad (1)$$

The predictor that calculates EA_i actually estimates the average message delay for the last n messages, shifting backward in time by $\eta \cdot s_i$ each A'_i. By adding α to estimation made by the predictor, the failure detector is able to absorb some variation in message

delay. A study made by [Nunes and Jansch-Porto 2004] observed that the predictor used in the NFD-E algorithm is not the more accurate, but the use of a constant safety margin α allows the failure detector to achieve a good QoS. Hence the predictor used by the NFD-E algorithm plays a more important role in QoS achievement than the one originally observed by its authors, being responsible for accounting for the uncertainty in message propagation delays typical of partially synchronous system.

The full algorithm as presented by its authors in [Chen et al. 2002] is shown in Algorithm 1.

Algorithm 1 NFD-E Algorithm

 1: **procedure** SENDHEARTBEAT   ▷ Procedure exclusive for p, using p's local clock.
 2:     **for** all $i \geq 1$, at time $i \cdot \eta$ **do** send heartbeat m_i to q

 3:

 4: **procedure** INITIALIZATION   ▷ Procedures exclusives for q, using q's local clock
 5:     $\tau_0 \leftarrow 0$
 6:     $\ell \leftarrow -1$

 7:

 8: **upon event** $\tau_{\ell+1} = now()$ **do**
 9:     $output \leftarrow Suspect$

10: **upon event** receives message m_j at time t **do**
11:     **if** $j > \ell$ **then**
12:         $\ell \leftarrow j$
13:         $\tau_{\ell+1} \leftarrow EA_{\ell+1} + \alpha$
14:         **if** $t < \tau_{\ell+1}$ **then**
15:             $output \leftarrow Trust$

16:

2.4. QoS of Failure Detection with Crash-Recovery Failures

In [Ma et al. 2010] the authors analyze the QoS of the synchronous *New Failure Detector with Synchronized Clocks (NFD-S)* algorithm found in [Chen et al. 2002]. This algorithm assumes a synchronous system and Ma et al. assume some sort of time synchronization to be present, such as the NTP protocol.

The expanded model outlined in [Ma et al. 2010] presented additional QoS metrics that complement those in [Chen et al. 2002] listed in Section 2.3:

- Query Accuracy Probability (P_A): it measures the probability that, at any arbitrary time when queried, the failure detector correctly indicates the state of the monitored process.
- Recovery Detection Time (T_{DR}): it measures the time the failure detector perceives a recovery at the monitored process.
- Detected Failure Proportion (R_{DF}): it measures the ratio of the detected crashes over the actual crashes.

The configuration of the failure detector and parameter estimation for a desired QoS are shown in [Ma et al. 2010]. While an interesting expansion of previous work,

[Ma et al. 2010] conclude that the proposed algorithm and QoS model needs to be enhanced, specifically to deal with short failures that may not be detected. Additionally, the experimental results in some tests differs substantially from the expected behavior of the implemented failure detector (specially for the R_{DF} QoS metric), reinforcing the need for more research for a failure detector in this system model. Because of these limitations, in this paper we do not use the expanded set of metrics for a crash-recovery system, but instead we use the basic set proposed by [Chen et al. 2002].

3. Algorithm

In the previous sections, we provided the theoretical background needed to understand key concepts such as synchrony models, failure detectors with quality of service and leader election. In this section we present the *New Failure Detector as a Leader Election Algorithm (NFD-L)* leader election algorithm, which is a derivative of NFD-E algorithm [Chen et al. 2002], shown in Algorithm 2. NFD-L is an efficient leader election with stability that only requires a single message each η. More importantly, NFD-L is designed to work on the crash-recovery system model with no clock synchrony among processes and requiring only a single stable memory write during process initialization.

3.1. A Failure Detector with QoS as a Leader Election

A leader election is a failure detector that outputs a single trusted process. The output of NFD-L is the *pid* of the trusted *leader*. It is possible that each process of the distributed system sees a different process as a *leader* at the same time, but when the system behaves synchronously for a sufficient amount of time, eventually all correct processes will agree on the same *leader*. To achieve this we combine the general principle of the bully algorithm [Garcia-Molina 1982] with the failure detecting properties of the NFD-E algorithm, using process uptime as a priority measure to ensure a basic level of stability.

Each process starts (or recovers) knowing nothing about the current leader. If it receives no message from a leader process, it then assumes it is the leader and starts sending heartbeat messages. A process loses the leadership when it receives a message from a process with greater priority: greater *uptime* or greater *pid* in the event of a tie on *uptime*. When a process recognizes another process as a leader, it stops sending heartbeats and starts monitoring heartbeats sent by the leader, in a behavior similar to NFD-E. Thus, at this moment, only a process acting as a leader sends periodic heartbeat messages to all other processes that will behave as monitors. The leader election stabilizes when each correct process receives the message of a single process with high enough priority.

In steady state operation and assuming a broadcast communication medium the cost of NFD-L will be equivalent to single instance of the NFD-E algorithm, compared to the $N^2 - N$ instances of the naive reduction. Furthermore, in the fail-free case after stabilization the QoS of NFD-L will be equivalent to the QoS of NFD-E as it is a generalization of this algorithm.

3.2. Dealing with Crashes and Recoveries

Each heartbeat message m sent by a monitored process p carries an incremental sequence number i used by a monitor process q to determine if an incoming heartbeat is still fresh,

Algorithm 2 NFD-L Algorithm
___
1: **procedure** INITIALIZATION
2:     $leader \leftarrow \bot$
3:     $self.uptime \leftarrow 0$
4:     $retrieve(self.zerotime)$
5:     **if** $self.zerotime = \bot$ **then**
6:         $self.zerotime \leftarrow now()$
7:         $store(self.zerotime)$
8:     $i \leftarrow \frac{now()-self.zerotime}{\eta}$

9: **procedure** SENDHEARTBEAT
10:     **if** $self.pid = leader$ **then**
11:         **for** all $i \geq 1$, at time $i \cdot \eta$ **do**
12:             send heartbeat m_i to all processes
13:             $self.uptime \leftarrow self.uptime + 1$

14:

15: **upon event** receives message m_j at time t **do**
16:     **if** $sender(m_j) = leader$ **then**
17:         **if** $j > \ell$ **then**
18:             $\ell \leftarrow j$
19:             $\tau_{\ell+1} \leftarrow EA_{\ell+1} + \alpha$
20:     **else**
21:         **if** $uptime(sender(m_j)) > uptime(leader)$ **then**
22:             $\ell \leftarrow j$
23:             $leader \leftarrow sender(m_j)$
24:             $output \leftarrow leader$
25:         **else**
26:             **if** $uptime(sender(m_j)) = uptime(leader)$ **then**   ▷ Tiebreaker
27:                 **if** $pid(sender(m_j)) > pid(leader)$ **then**
28:                     $\ell \leftarrow j$
29:                     $leader \leftarrow sender(m_j)$
30:                     $output \leftarrow leader$

31:

32: **upon event** $\tau_{\ell+1} = now()$ **do**
33:     $leader \leftarrow self.pid$
34:     $output \leftarrow leader$
___

thus trusting or suspecting p to remain as leader. After recovering from a crash, it is important that process p starts sending correct sequence numbers to q, i.e., process p should remember which one was the exact sequence number i it sent before it crashed. If process p "forgets" its sequence number and starts sending i from the initial value, it will not be trusted by process q until i reaches the value it held before process p crashed.

The naive solution to remember the correct sequence number i to be used after a crash, is to write on persistent storage i every time process p sends a heartbeat message m_i. However, while correct, this solution isn't efficient regarding the number of writes to costly stable memory. We propose a cheaper solution that uses a single write operation once a process is initialized and to a read operation per process recovery.

Our solution takes advantage of the assumption that every process has access to a local clock and that the i-th heartbeat message should be sent at time $i \cdot \eta$. It works as follows: when a process starts for the very first time or it recovers from a previous crash, it checks if it wrote on persistent storage a timestamp of a previous initialization. If there is not a previous timestamp on persistent storage, the process then writes the current timestamp on persistent storage. This way, this value will be read and constant at every single recovery of that process. Using this value a process calculates what the current message label i is supposed to be, based on a function of the elapsed time since the first startup and η (Line 8). It ensures that every time a process starts or recovers, the message label i will not violate the properties defined in [Chen et al. 2002], i.e., every message label i sent by a process will be greater than the previous label $i - 1$. The only requirement is that the local clock keeps increasing with no drift during crashes.

4. QoS Analysis

In this section we present the QoS analysis of the NFD-L algorithm. The QoS analysis is made using the three metrics proposed in [Chen et al. 2002], T_D, T_{MR} and T_M (Section 2.3), with the addition of the T_{DR} metric proposed in [Ma et al. 2010] for the crash-recovery failure model (Section 2.4). These metrics will be grouped in failure detector accuracy metrics (T_{MR}, T_M) and failure detector speed metrics (T_D, T_{DR}). The experiments have the objective of assessing if the QoS of the proposed failure detector is within the bounds achieved by Chen's algorithm.

4.1. Enviroment and configuration of the leader election

These experiments ran at the Maritaca computational cluster, located at Universidade Federal de São Carlos, Sorocaba, São Paulo. We used 5 identical nodes, each one equipped with an 8 cores/16 threads processor and 16 GB of volatile memory. Each node used a hard disk as its persistent storage and all nodes were interconnected by a Gigabit Ethernet link.

The NFD-L algorithm by design selects a node as the leader and the other four nodes behave as monitor processes, receiving heartbeats sent by the leader. We monitored the leader election output of each monitor process to assess the QoS metrics. To generate a constant load on the network and on the system, we used the leader election as a service provider for the Treplica replication framework [Vieira and Buzato 2008]. The replication rate was set to the saturation point of the application with this setup, which was 2700 operations per second on average.

To generate the parameters η and α for the leader election, we used the configurator described by [Chen et al. 2002] during a period when the network was under high load due to the test application. We then measured the network probabilistic behavior for 1 hour, repeating it 3 times in total in different hours of the day. The observed message loss probability $p_L = 0.0175917$ and the observed message delay variance $V(D) = 25.3356$ were then used with the following QoS metrics $T_D = 1000$ ms, $T_{MR} = 3600000$ ms and $T_M = 1000$ ms to obtain $\eta = 330$ ms and $\alpha = 670$ ms.

We made two set of experiments in order to respectively observe the accuracy and the speed of the failure detector. The accuracy was measured by counting the average mistake rate (T_{MR}) and the mistake duration (T_M) for each monitor process during 6 runs of 1 hour each. During the experiment the leader process did not fail, thus all suspicions were mistakes that were eventually corrected. Let $t_{m0}, ..., t_{mn}$ be the timestamps of the mistakes made by a process and $t_{r0}, ..., t_{rn}$ be the timestamps of the correction of those mistakes, then we computed the mistake rate of that process as:

$$\frac{1}{T_{MR}} = \frac{1}{\frac{1}{n}\left(\sum_{i=1}^{n}\left(t_{mi} - t_{mi-1}\right)\right)}$$

The average mistake duration of a process was computed as the average time taken to correct a mistake:

$$T_M = \frac{1}{n}\left(\sum_{i=0}^{n}\left(t_{ri} - t_{mi}\right)\right)$$

The speed of the leader election algorithm was measured by the crash detection time (T_D) and by the recovery detection time (T_{DR}). To be able to measure T_{DR}, we have randomly selected a node as a high priority leader, breaking the specification of NFD-L in such way that this leader would be immediately reelected after a recovery. We then injected a single crash in the leader process and then we recovered it, measuring the time elapsed for each process to detect the crash and, afterwards, the time elapsed for each process detect the leader recovery. This experiment consisted of 10 repetitions of a crash-recovery cycle, with 60 seconds between a crash and the following recovery. Let t_c be the time of a leader crash and t_d the time of detection of that crash by a monitor process, the detection time T_D was obtained by calculating the difference of times:

$$T_D = t_d - t_c$$

In a very similar way, the recovery detection time was calculated considering the recovery time t_r and the detection time of that recovery t_{dr} by a process:

$$T_{DR} = t_{dr} - t_r$$

For this experiment we used the NTP protocol at the local network to synchronize the clocks of the nodes of the cluster. This procedure is not necessary for the algorithm to work properly, but it made easier to calculate time differences between the nodes. By using the NTP protocol we also observed that the average message delay between the nodes were a thousand of times smaller than the message intersending interval η intended to be used in our experiments, so we considered the message delay negligible.

4.2. Analysis

We plotted the QoS metrics observed in the experimental runs in boxplots. For each run the data points represent the metric as measured by each one of the four monitor process with respect to the current state of the actual leader. This amounts to a total of 24 points for the experiments assessing the accuracy of the failure detector and 40 points for the experiments assessing the speed of the failure detector. The expected value of the QoS metric used to configure the NFD-L algorithm is shown by the traced line in all plots. Numeric values for the data points represented in the figures are shown in Table 1, listing the values of the 1st, 2nd and 3rd quartiles.

Figure 1(a) shows the observed mistake rate. The majority of the processes (the darker line at the bottom of the image) made no mistakes, successfully achieving the required QoS of $T_{MR} = 3600000$ ms. However, there were outlier processes that did not achieve the required QoS. Among these, there is a single outlier that is not shown on figure, with a mistake rate of 33.34 mistakes/s. These outliers consist of a couple of mistakes in an one hour experiment, negatively affecting the average mistake recurrence time. In fact, the way the metric is computed (as proposed in [Chen et al. 2002]) increases the impact of each of the low number of mistakes. For example, the 33.34 mistakes/s outlier was created by one pair of mistakes 30 ms apart. The Figure 1(b) shows the observed mistake duration. The processes that made no mistakes were disconsidered. All the observed processes did meet the required QoS $T_M = 1000$ ms.

(a)   (b)
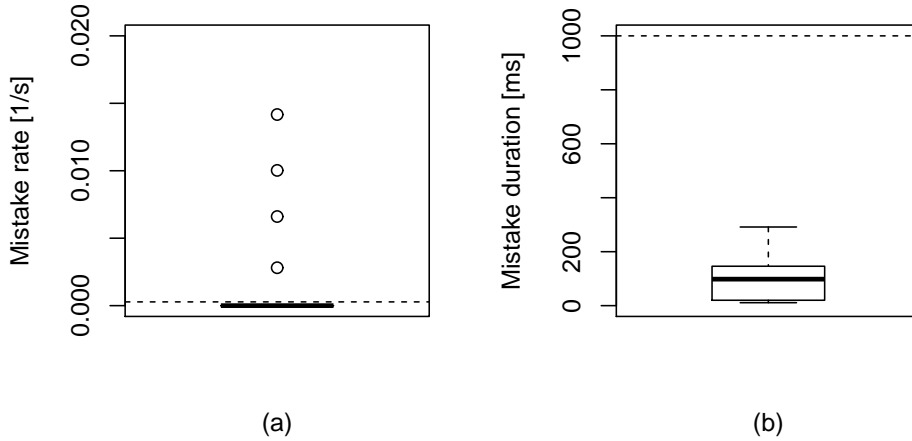
Figure 1. (a) The mistake rate $1/T_{MR}$ and (b) the mistake duration T_M. The traced line in the plots represents the upper bound QoS requirements for the leader election.

The crash detection time is shown in Figure 2(a). All the crashes were detected within the required QoS as the crash detection is bounded to $T_D = \eta + \alpha$. The Figure 1(b) shows the recovery detection time T_{DR} observed by the monitor processes. The speed metrics had less variation than the accuracy ones. It suggests that accuracy is more sensitive to process dependability than to network probabilistic behavior. As we assessed the QoS metrics of the failure detector integrated in a loaded application, pauses in the processing of the application created loss and delay of messages more intense than the ones

created by network probabilistic behavior alone. In a sense, this amounts to periods of omission faults where the processes stopped processing. We conjecture that a QoS model for the crash-recovery process model should be able to absorb this process behavior.

(a)   (b)
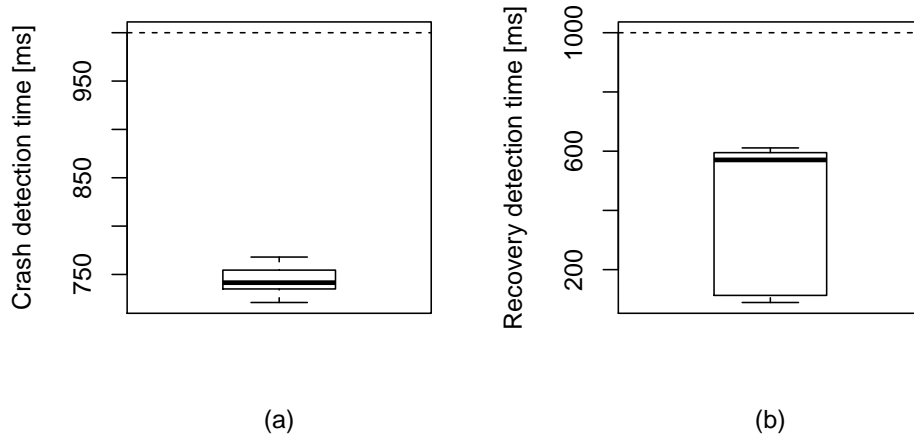
Figure 2. (a) The leader crash detection time T_D and (b) the leader recovery detection time T_{DR} observed by each monitor process. The traced line represents the upper bound QoS requirements for the leader election.

QoS Metric	1st. Quartile	Median	3rd. Quartile	Upper bound QoS
$1/T_{MR}$ (Figure 1(a))	0 1/s	0 1/s	0 1/s	$0.278 \cdot 10^{-3}$ 1/s
T_M (Figure 1(b))	20 ms	98.223 ms	146.03 ms	1000 ms
T_D (Figure 2(a))	735 ms	741.5 ms	754.25 ms	1000 ms
T_{DR} (Figure 2(b))	113 ms	570.5 ms	595 ms	1000 ms

Table 1. Summary of plot values

5. Conclusion

In this paper we presented the NFD-L leader election for the asynchronous crash-recovery failure model. This algorithm is an extension of the NFD-E [Chen et al. 2002] algorithm adapted to work as a leader election that is efficient in terms of its use of stable memory and message exchanges. The NFD-L algorithm used two novel techniques to achieve its efficiency: (i) instead of using a counter of crashes as usual [Martín et al. 2009], it uses a counter of uptime to prioritize stable processes and (ii) it uses a single write in stable memory to create a sequence of heartbeats that isn't interrupted by crashes, but only appears to be "paused", reducing crash failures to message omission failures.

We analyzed the performance of the NFD-L algorithm on a real system to verify how well it met the given QoS, configured as proposed in [Chen et al. 2002]. We were able to achieve the desired QoS, but the accuracy QoS metrics suffers for repetitive short period mistakes caused by the application. Our results suggest that accuracy of a failure detector is dependent on process dependability besides the effects of network probabilistic behavior. In particular, application overload spikes, lock contention and other performance

problems can severely interfere in the generation and processing of heartbeats. Thus, further enhancements of the QoS model are needed to make the algorithm suitable to an environment where the processes dependability is a main concern. We believe these changes must take in consideration crash-recovery parameters such as the ones found in [Ma et al. 2010] or be made more application specific as proposed in [Hayashibara et al. 2004].

Acknowledgments

References

Cachin, C., Guerraoui, R., and Rodrigues, L. (2011). *Introduction to Reliable and Secure Distributed Programming*. Springer Berlin Heidelberg.

Chandra, T. D., Hadzilacos, V., and Toueg, S. (1996). The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722.

Chandra, T. D. and Toueg, S. (1996). Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267.

Chen, W., Toueg, S., and Aguilera, M. (2002). On the quality of service of failure detectors. *Computers, IEEE Transactions on*, 51(5):561–580.

Falai, L. and Bondavalli, A. (2005). Experimental evaluation of the qos of failure detectors on wide area network. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, DSN '05, pages 624–633, Washington, DC, USA. IEEE Computer Society.

Fischer, M. J., Lynch, N. A., and Paterson, M. S. (1985). Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382.

Garcia-Molina, H. (1982). Elections in a distributed computing system. *IEEE Trans. Comput.*, 31(1):48–59.

Guerraoui, R. (2000). Indulgent algorithms (preliminary version). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, pages 289–297, New York, NY, USA. ACM.

Hayashibara, N., Defago, X., Yared, R., and Katayama, T. (2004). The φ accrual failure detector. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems*, SRDS '04, pages 66–78, Washington, DC, USA. IEEE Computer Society.

Lamport, L. (1998). The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169.

Larrea, M., Fernández, A., and Arévalo, S. (2000). Optimal implementation of the weakest failure detector for solving consensus (brief announcement). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, pages 334–, New York, NY, USA. ACM.

Ma, T., Hillston, J., and Anderson, S. (2010). On the quality of service of crash-recovery failure detectors. *IEEE Trans. Dependable Secur. Comput.*, 7(3):271–283.

Malkhi, D., Oprea, F., and Zhou, L. (2005). Ω meets Paxos: Leader election and stability without eventual timely links. In *DISC '05: Proceedings of the 19th International Conference on Distributed Computing*, volume 3724 of *Lecture Notes in Computer Science*, pages 199–213. Springer.

Martín, C., Larrea, M., and Jiménez, E. (2009). Implementing the omega failure detector in the crash-recovery failure model. *Journal of Computer and System Sciences*, 75(3):178–189.

Nunes, R. C. and Jansch-Porto, I. (2004). QoS of timeout-based self-tuned failure detectors: The effects of the communication delay predictor and the safety margin. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, DSN '04, pages 753–, Washington, DC, USA. IEEE Computer Society.

Reynal, M. (2005). A short introduction to failure detectors for asynchronous distributed systems. *ACM SIGACT News*, 36(1):53–70.

Sotoma, I., Roberto, E., and Madeira, M. (2006). A markov model for providing quality of service for failure detectors under message loss bursts. Technical Report IC-06-013, Institute of Computing, University of Campinas.

Vieira, G. M., Garcia, I. C., and Buzato, L. E. (2014). Seamless Paxos coordinators. *Cluster Computing*, 17(2):463–473.

Vieira, G. M. D. and Buzato, L. E. (2008). Treplica: Ubiquitous replication. In *SBRC '08: Proc. of the 26th Brazilian Symposium on Computer Networks and Distributed Systems*, Rio de Janeiro, Brasil.