A Robust and Lightweight Stable Leader Election Service for Dynamic Systems*

Nicolas Schiper
University of Lugano, Switzerland
nicolas.schiper@lu.unisi.ch

Sam Toueg
University of Toronto, Canada
sam@cs.toronto.edu

Abstract

We describe the implementation and experimental evaluation of a fault-tolerant leader election service for dynamic systems. Intuitively, distributed applications can use this service to elect and maintain an operational leader for any group of processes which may dynamically change. If the leader of a group crashes, is temporarily disconnected, or voluntarily leaves the group, the service automatically re-elects a new group leader. The current version of the service implements two recent leader election algorithms, and users can select the one that fits their system better. Both algorithms ensure leader stability, a desirable feature that lacks in some other algorithms, but one is more robust in the face of extreme network disruptions, while the other is more scalable.

The leader election service is flexible and easy to use. By using a stochastic failure detector [5] and a link quality estimator, it provides some degree of QoS control and it adapts to changing network conditions. Our experimental evaluation indicates that it is also highly robust and inexpensive to run in practice.

1. Introduction

In this paper we describe and experimentally evaluate a *fault-tolerant leader election service* for dynamic systems. Leader election plays an important role in the design of fault-tolerant applications. Intuitively, this is because a leader can be used as a central coordinator that enforces consistent behavior among processes. For example, the algorithms in [13, 16, 9] require a fault-tolerant leader election mechanism to manage data replication or to enforce process agreement despite failures. More generally, a leader election service can be used to solve *consensus* and *atomic broadcast* — two primitives at the core of Lamport's *state machine approach* for building fault-tolerant applications [12].

The service we propose can be used to elect and maintain a leader among any dynamically changing subset of application processes (called a *group*) in a system with random process crashes, process recoveries, message losses and message delays. The leader of a group must be operational (and a current member of that group): if it crashes, is temporarily disconnected, or voluntarily leaves the group, the service automatically re-elects a new leader and notifies the processes in the group of this change. There may be periods of time when a group has no operational leader or has several leaders (e.g., just after a crash of the current leader or when several processes compete to gain the group leadership), but the service ensures that these periods are very short in practice: roughly speaking, as long as a group has at least one reasonable candidate for leadership (an operational process with well-behaved communication links), the service provides the group with a unique and stable leader.

Groups are *dynamic*, i.e., each application process can join or leave any group at any time (each process can concurrently belong to several groups), and the service provides a leader among the operational processes of each group. If the current leader of a group fails or leaves the group, the service automatically elects a new leader.

When a process joins a group, it can indicate whether it is willing to be a leader for this group (i.e., whether it is a "candidate" for leadership). For each group, the service selects a leader only among the (currently operational) candidates in that group. This feature can be useful in practice for several reasons. First, a process may be unwilling to be the group leader because it cannot handle the associated workload. Second, the cost of a leader election (in terms of messages exchanged) is typically proportional to the number of candidates that concurrently compete for this position. So a large group may want to restrict the election to a small number of candidates (e.g., among $t + 1$ candidates, t of which may fail).

*This research was supported in part by NSERC Canada (grant number 250468-07) and by SNSF Switzerland (project number 200021-107824).

To select the "core" algorithms for our service, we implemented three different leader election algorithms and experimentally compared their performance under several different settings. Of these three algorithms, two performed extremely well even in systems with very high rates of workstation and communication failures. These two algorithms are enhanced versions of algorithms in [2, 4] that were modified to integrate the stochastic failure detector (FD) algorithm of Chen *et al.* [5] and a link quality estimator in order to provide some degree of QoS control and to adapt to changing network conditions. In the current version of the service, users can chose between these two leader election algorithms, and select the one that fits their system better: as our experimental evaluations show, one is more robust in the face of extreme network disruptions, while the other is more scalable. With either algorithm, the service have several desirable features, that we now describe.

The service ensures a high degree of *leader stability*. Roughly speaking, leader stability means that the current leader is not demoted and replaced if it is still operational [1]. This is in contrast to several leader election algorithms, notably those that select a leader using some fixed function on process identifiers. For example, consider the algorithm that selects the leader as the process with the smallest identifier among the processes that seem to be currently alive [17, 8, 14]. With this algorithm, a group leader ℓ is systematically demoted and replaced every time a process with a smaller identifier than ℓ newly joins the group (or rejoins the group after it recovers from a crash), and this occurs even if ℓ is fully functional and has been working well as the group leader. It is clear that demoting fully functional leaders for such spurious reasons can be costly and disruptive to leader-based applications, and therefore should be avoided.

With this service, applications have some control on the *Quality of Service (QoS) of the leader election.* More precisely, for each group g, each application process p in g can specify bounds on (1) the time p takes to detect the crash of the current leader of g, and (2) the "accuracy" of this detection (in terms of the frequency and duration of *mistakes*, i.e., of false crash detections). Bounding (1) is important because, as our experiments suggests, the time it takes to detect the crash of a leader is often the dominating component of what we call *leader recovery time* (i.e., the time that elapses from the crash of a leader to the installation of a new leader). Bounding (2) is also important because making a mistake on the leader (i.e., thinking that it has crashed while it is still functional) can lead to an unnecessary and expensive change of leader. To achieve the above, we implemented the stochastic failure detector (FD) algorithm of Chen *et al.* [5] and integrated it in (the leader election algorithms of) our service: Under some conditions, this FD provides QoS guarantees on the speed and accuracy of the failure detection.

The leader election service adapts to changing network conditions. In fact, applications do not specify static, low-level parameters such as timeout durations or the frequency of "I-am-alive" messages, these are automatically determined and continuously updated according to the "current" network conditions as follows. The underlying FD algorithm of Chen *et al.* periodically reevaluates these FD parameters as a function of two inputs: (a) the required QoS of the FD (as specified by an application), and (b) the current quality of the communication links (as given by a link quality evaluator module). This allows the underlying failure detector, and ultimately the service, to automatically adjust to changing network conditions.

Finally, our experimental evaluation described in Section 6 indicates that the leader election service is indeed quite stable, robust and inexpensive to run.

The source code of our leader election service, as well as information on how to install and use it, are available at: http://www.inf.unisi.ch/phd/schiper/LeaderElection/. Several experimental results that are omitted here are also posted on that site.

Roadmap. The rest of the paper is structured as follows. Section 2 summarises related work. Section 3 describes the failure detector of Chen *et al.*. Section 4 presents the architecture of the leader election service. Section 5 describes the QoS metrics that we use to evaluate and compare the different leader election algorithms that we implemented in our service. Section 6 describes our experimental settings and results. Some concluding remarks appear in Section 7.

2. Related Work

The literature on failure detectors and more specifically leader election is abundant. We now briefly review some of the relevant papers. In [8], the authors study the problem of leader election in partitionable networks. They define the notion of *stable partition* as a partition in which every pair of processes can communicate in a timely manner. In each one of these stable partitions, the process with the smallest identifier is elected as the partition leader. In [14], the authors give a communication-efficient algorithm, an algorithm in which eventually only the leader sends messages. However, their algorithm requires strong system assumptions, i.e., all links have to be eventually timely (a link is eventually timely if all messages sent after some time t take at most δ units of time to be received). In [10], the authors study the leader election problem in a probabilistic model, i.e., a model where process crashes and link failures are probabilistic. In this algorithm, a parameter controls the trade-off between the correctness probability guarantee and the message complexity.

In [1], several leader election algorithms are given for systems where at least one non-faulty process has input *and*

output communication links that are eventually timely (all the other links may lose all messages). These algorithms are *message-efficient*, i.e., after a leader is elected, only the leader sends messages (it must do so periodically to inform other processes that it is still alive). Furthermore, they ensure a strong form of *leader stability*, i.e., a leader is not demoted if its input and output links have been timely for some amount of time Δ (Δ depends on the algorithm). In [2] and [4], algorithms are given for systems where only the output links of some non-faulty process are eventually timely (the input links may be arbitrarily slow). One of these algorithms is communication-efficient but it assumes that message losses are only intermittent (a message repeatedly sent over a lossy link is eventually received). Another algorithm tolerates links that completely "crash" (i.e., lose all messages), but it requires quadratic communication even after a leader has been elected. In [3], the authors consider systems where at most f processes may crash and give a communication-efficient algorithm where only f of the output links of a non-faulty process need to be eventually timely. In [15], the authors give a strongly stable and communication-efficient algorithm that requires one non-faulty process to be eventually f-accessible, i.e., a process that has eventually timely input and output links to f processes. An interesting feature of this algorithm is that these eventually timely links need not be fixed and may change during the execution of the algorithm. In [7], a communication-efficient algorithm is presented that does not require a priori knowledge of the processes' identities.

3. Chen *et al.*'s Failure Detector with QoS

Failure detection is at the core of any leader election service: it is used to detect whether the current leader has failed and needs to be replaced, and to determine which of the candidates for replacing the failed leader are operational.

In our service, we implemented the stochastic failure detector algorithm due to Chen *et al.* [5] and integrated it with a link quality estimator module and a dynamic scheduler. Together, these modules provide some QoS control on failure detection under changing network conditions. We now briefly describe these modules (shown in Figure 1).

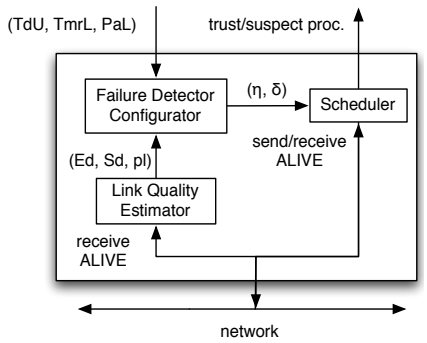

Figure 1. The failure detector module

When a process p monitors the status of another process q, it gives the QoS requirement of this monitoring in terms of 3 parameters, denoted T_D^U, T_{MR}^L, and P_A^L: (a) T_D^U is an upper bound on the time the FD takes to detect the crash of q, (b) T_{MR}^L is a lower bound on the expected time between two consecutive mistakes of the FD (the FD makes a mistake when it tells p that q crashed and this is not true), and (c) P_A^L is a lower bound on the probability that, at a random time, the FD is correct about the operational status of q.

It is clear that achieving such a QoS requirement depends on the frequency at which q sends *alive* messages to p, the timeout that p uses on these messages, and on quality of the communication link from q to p. In the following, we briefly explain the three modules that are related to this.

The *Link Quality Estimator* module continuously estimates the "quality" of the link from q to p in terms of three quantities: the probability of message loss p_L, the expected value of message delay E_d, and the standard deviation of message delay S_d. This estimation is done using the *alive* messages that p receives from q.

The *Failure Detector Configurator* computes the FD parameters that ensure the required QoS under some assumptions on the network behavior [5]. More precisely, this module (dynamically) computes (1) the frequency η at which q must send *alive* messages to p, and (2) the timeout δ that p must use to determine q's operational status. To compute η and δ, the module takes two inputs: (a) the required QoS of the monitoring of q, i.e., the values of T_D^U, T_{MR}^L, and P_A^L given by p, and (b) the estimated quality of the link from q to p, i.e., the latest values of p_L, E_d, and S_d computed by the Link Quality Estimator module.

The *Scheduler* uses the output (η, δ) of the Failure Detector Configurator as follows: it schedules the sending of *alive* messages by q at a frequency of η, and it uses the current timeout δ and the time that p received its last *alive* message from q to schedule the trust/suspect notifications at p.

4. The Leader Election Service Architecture

The architecture of the leader election service is based on the failure detector service proposed by Deianov *et al.* in [6] and implemented by Ivan *et al.* in [11]. This architecture reduces the overall network and CPU overhead by sharing some tasks (e.g., estimating the quality of the communication links or determining whether a workstation is operational): the cost of these tasks is shared by the various applications that concurrently use the service. The leader election service is written in C and is compatible with any Unix/Linux. The service's architecture is illustrated in Figure 2 and briefly explained below.

Application processes are linked to a shared library implementing the service's API. The main API functions al-

low processes to register/unregister with the service and to join/leave groups. This library communicates with the *Command Handler* module to serve the requests of application processes.

To use the leader election service, a process p must first register itself with the service using a unique process identifier. Once this is done, p can join and leave any group at anytime. To join a group g, p must specify the following four parameters: (1) g's identifier, (2) whether p is a candidate for g's leadership or not, (3) the way p wishes to find out who is the current leader of g (by an *interrupt* from the service, whenever the leader of g changes, or by *querying* the service, whenever p wants to do so), and (4) the QoS (T_D^U, T_{MR}^L, P_A^L) of the underlying FD used by the service to elect a leader in group g.

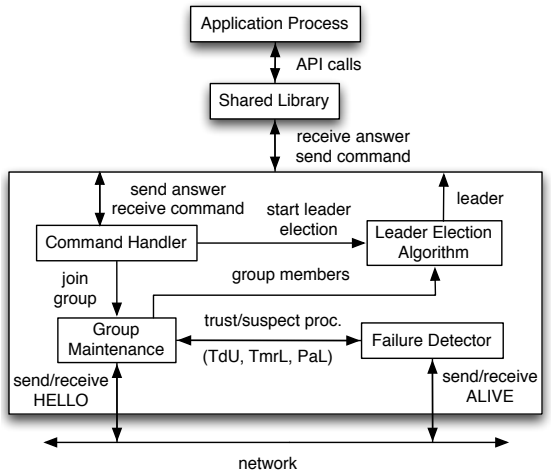

Figure 2. The leader election service

The core functionality of the service resides in the *Group Maintenance*, the *Failure Detector*, and *Leader Election Algorithm* modules.

For each group g, the *Group Maintenance* module builds and maintains (a) the set of processes that are currently in g, and (b) the subset of processes of g that are currently "active" in g (roughly speaking, a process in g is active if it is currently alive and competing for the leadership of g; as we will see in Section 6, depending on the leader election algorithm used, either all the alive processes in g are active [4] or eventually only the leader of g remains active [2]). To determine (a) and (b), the Group Maintenance module needs to determine the status of processes in each group. To do so, it uses the *Failure Detector* module, which was implemented using the failure detector of Chen *et al.* as decribed in Section 3.

The *Leader Election Algorithm* module maintains a leader in each group g by executing the algorithm in [4] or in [2] in g, depending on the version of the service used (these two algorithms are sketched in Section 6). Other leader election algorithms can be "plugged in" here in future versions of the service.

5. Leader Election QoS Metrics

We used three QoS metrics to compare the performance of leader election services in various settings. The first one captures the "speed" of the leader election service: it measures the time that the service takes to *recover from the crash of the current leader*, i.e., the time that elapses from the moment a group loses its current leader due to a crash to the time when the service completes the election of a new group leader. Roughly speaking, this measures the sum of two periods: (a) *detection time*, i.e., the time that the service takes to detect that the current leader of a group has crashed (and so an election must occur), and (b) *election time*, i.e, the time that the service takes to ensure that all the alive processes in the group agree on a new leader. Note that the group is effectively leaderless during all this time.

To define this metric more precisely, we say that *a group has a leader at time t* if, at time t, there is some alive process ℓ such that every alive process in this group has ℓ as its leader. The *leader recovery time*, denoted T_r, is a random variable that measures the time that elapses from the time when the leader of a group crashes to the time when the group has a leader again. The *average leader recovery time* metric, denoted $\overline{T_r}$, is the expected value of T_r.

The second metric that we use, called *average mistake rate*, captures the "accuracy" and "stability" of a leader election service: it is the rate at which the service makes a "mistake" by demoting a functional group leader. Intuitively, such a mistake may occur for one of two reasons. First, the service's failure detection mechanism may be *inaccurate*: it erroneously suspects that the current leader ℓ has crashed, and this triggers an unnecessary reelection to replace ℓ. Second, the service may be *unstable*: it may demote and replace a well-behaved leader ℓ for spurious reasons (e.g., it may demote and replace ℓ just because a process with a smaller id than ℓ joins the system). It is clear that demoting a functional leader should be avoided: it prevents the progress of all leader-based applications until a new leader is elected.

To capture the above metric, we say that the demotion of a proces ℓ from leadership is *unjustified* if ℓ loses the leadership of the system *even though ℓ has not crashed*. The *mistake rate*, denoted λ_u, is a random variable that measures the number of *unjustified* leader demotions that occur in an hour. The *average mistake rate*, denoted $\overline{\lambda_u}$, is the expected value of λ_u.

The last metric that we consider measures the availability of a group leader. Specifically, for any given group, the *leader availability* metric, denoted P_{leader}, is the probability that, *at a random time*, the group has a (commonly agreed and alive) leader. Intuitively, this metric is of inter-

est to applications for the following reason. Suppose that a leader-based application needs to use the services of a leader at random times. Then P_{leader} is the probability that when the application needs a leader, it finds that a commonly agreed and functional leader is available. Note that, for each group, this metric corresponds to the proportion of time when the group has a leader.

6. Evaluation of Leader Election Services

In this section, we experimentally evaluate and compare three different versions of the leader election service, denoted $\mathcal{S}_1$, $\mathcal{S}_2$, and $\mathcal{S}_3$, under various network behavior settings. All three versions share the same service architecture (described in Section 4) except that each one uses a different algorithm in the *Leader Election Algorithm* module (as we explain later). To evaluate and compare the performance of $\mathcal{S}_1$, $\mathcal{S}_2$, and $\mathcal{S}_3$, we use the leader election QoS metrics described in Section 5. We also evaluate their costs in terms of CPU and network bandwidth utilization.

In the following, we first describe the parameters of our experiments, and then describe the experimental results for each version of the service.

6.1. Experimental System Parameters

The experiments were conducted on a LAN consisting of a cluster of 12 workstations connected with a gigabit switch. Each workstation is a P4 3.2 Ghz, with 512 MB of RAM, running an instance of the leader election service on top of SuSe Linux 9.2. All our experiments, except those measuring scalability, consisted of a single group of 12 application processes, one in each workstation, using the service. The duration of each experiment varied between 1 and 5 days.

In our local area network, workstations rarely crash, there are practically no message losses and the average message delay is only about 0.025 ms. To evaluate our leader election service implementation in other (less favorable) network environments, we implemented a module that causes message losses or delays (by actually dropping or delaying the service messages) and a module that simulates the crashes and recoveries of workstations (by actually killing and later restarting individual instances of the service running on those workstations). Note that each workstation crash also kills one of the 12 application processes using the leader election service (in particular, it may kill the current leader of the group).

Our experiment parameters control (1) the behavior of the network that runs the leader election service and (2) the QoS of the underlying FD used by the service. We now describe these parameters in more detail.

Workstations behavior. In all our experiments, workstations crash and recover at random times. More precisely, for each workstation, the *time between two consecutive workstation crashes* is exponentially distributed with an expected value of 600 seconds (i.e., on the average each process crashes every 10 minutes). The *time that a workstation takes to recover from a crash* is also exponentially distributed, with an expected value of 5 seconds (i.e., on the average a workstation takes 5 seconds to recover from a crash). Even though crashing every workstation every 10 minutes on the average is extremely pessimistic for realistic environments, we chose this setting for two reasons: (1) to stress-test and evaluate the leader election service under adverse conditions, and (2) to collect enough samples (e.g., to estimate the average leader recovery time) in reasonable time. A recovery time of about 5 seconds is long enough to force the algorithms to notice and react to every crash, and it is short enough to prevent frequent periods during which no process is alive.

Communication links behavior. Every group of n processes has $n(n-1)$ directed communication links. In our experiments, we simulated two types of links — lossy links and links that are prone to crashes — as described below.

LOSSY LINKS. In most of our experiments (Figure 3 to Figure 5) we simulate communication links with random message losses and delays. More precisely, every message sent through a lossy link has a probability p_L of being dropped by the link. If a message is not dropped by the link, its delay is exponentially distributed with an expected value $\overline{D}$.

We run experiments where the *probability of message loss* p_L is 1/10, 1/100, or 1/1000, and the *expected message delay* $\overline{D}$ is 1ms, 10ms, or 100ms. Thus, we evaluated each version of the leader election service with 9 different values of the tuple $(\overline{D}, p_L)$ that characterizes lossy links behavior. For brevity, in this paper we only show the experimental results for the "worst" 4 of these 9 pairs, namely, those where $\overline{D}$ is 10ms or 100ms, and p_L is 1/10 or 1/100, i.e., for the pairs $(10\text{ms}, 0.01)$, $(100\text{ms}, 0.01)$, $(10\text{ms}, 0.1)$, and $(100\text{ms}, 0.1)$.

To see how the various leader election algorithms perform in systems with well-behaved communication links, we also run experiments where the message losses and delays were *only* those that really occurred in our local area network. During these experiments, we measured an average message delay of only 0.025 ms and there were practically no message losses, i.e., these experiments were conducted over links characterized by the pair $(0.025\text{ms}, 0)$.

In summary, we evaluated each leader election algorithm with ten different behaviors of lossy communication links: the nine simulated ones and the behavior of our local area network, which corresponds to $(0.025\text{ms}, 0)$. These behaviors cover a large spectrum of network types such as local-area, metropolitan-area, and wide-area networks.

LINKS PRONE TO CRASHES. In some experiments (Figure 7) we simulate links that are subject to random crashes and recoveries. When a link crashes, it completely *dis-*

connects the receiver from the sender (by dropping all the sender's messages) until the link recovers. When a link recovers, it becomes fully operational again.[1] In our experiments with link crashes, for each link, *the time between two consecutive link crashes* is exponentially distributed with an expected value of 60 seconds, 300 seconds, or 600 seconds. So in these experiments *each* link crashes every 1, 5, or 10 minutes, on the average. The *time that a link takes to recover from a crash* is also exponentially distributed with an expected value of 3 seconds — a period that is long enough to trigger disruptive "false suspicions" (as it will be clear later, with our QoS settings, any link crash that lasts more than one second can force some process to falsely suspect another one to have crashed).

QoS of the underlying FD. As we mentioned earlier, our leader election service uses the stochastic FD by Chen *et al.* [5] as the underlying failure detector. Under some probabilistic assumptions on the system, this FD allows any process p to monitor the status of another process q with some QoS guarantees in terms of the *speed and accuracy* with which it detects q's crash. Every application that uses our leader election service can set the QoS of the FD to gain some control on the QoS of the leader election service.[2] In almost all our experiments, we set the QoS of the underlying FD (of Chen *et al.*) as follows. For every process p that monitors a process q: (1) Process p takes at most 1 second to detect q's crash; (2) on average p makes at most 1 mistake about the status of q every 100 days; and (3) the probability that p correctly estimates q's operational status, at a random time, is at least 0.99999988. Formally, $T_D^U = 1$ second, $T_{MR}^L = 100$ days, and $P_A^L = 0.99999988$. We selected this strong QoS setting to evaluate the performance of the service under demanding requirements.

It is clear that the QoS of the leader election service depends on the QoS of the underlying failure detector that the service uses. For example, a failure detector that takes a long time to detect the crash of the current leader will delay its replacement by at least the same amount of time, and this increases the leader recovery time and decreases the overall leader availability (in Section 6.6 we describe some experiments that explore the relation between crash detection time and leader recovery time). On the other hand, a failure detector that makes frequent mistakes (e.g., by erononeously declaring that the current leader crashed) can increase the number of unjustified leader demotions, i.e, increase the average mistake rate of the leader election service.

Unless we explicitly state otherwise, all the experiments described henceforth were done with the above settings.

[1] When a link is operational, its message losses and delays are those of our (real) network: losses are practically nil, and the average delay is 0.025ms.

[2] In fact, each group of processes can chose a different QoS for the underlying FD.

6.2. The Service $\mathcal{S}_1$

Description. The leader election algorithm of $\mathcal{S}_1$ denoted Ω_{id} is quite simple: the leader of a group is just the process with the smallest identifier among the processes that are currently deemed to be alive in this group. To estimate who is currently alive in a group, processes can periodically send an *alive* message to every other process in the group, say once every η seconds, and use some timeout δ on these messages (so that p declares that q has crashed if more than δ time elapses since p received the last *alive* from q).

An *ad-hoc* implementation of the above failure detector uses *fixed* values for η and δ. Such an implementation, however, has two drawbacks. First, the delicate question of how to fix η and δ is (usually) left to the user. But determining the "right" values for η and δ is far from trivial: it depends on the network characteristics (e.g., the distribution of message delays and the probability of message loss) and also on speed and accuracy of the failure detection that one wants to achieve. Second, using fixed values for η and δ does not adapt well to changing network conditions.

To avoid these problems, $\mathcal{S}_1$ uses the FD algorithm described in [5] (in conjunction with a link quality estimator module that continuously evaluates the network behavior).

Experimental Evaluation. Figure 3 shows the average leader recovery time $\overline{T_r}$ and the average mistake rate $\overline{\lambda_u}$ of $\mathcal{S}_1$ as measured in our experiments.[3] More precisely, it gives the values of $\overline{T_r}$ and $\overline{\lambda_u}$ that we measured when we run $\mathcal{S}_1$ in each of 5 networks with different lossy link charateristics (recall that $\overline{D}$ is the average message delay and p_L is the probability of message loss). In Figure 4, we show the leader availability of $\mathcal{S}_1$ (and compare it to the one of $\mathcal{S}_2$).

We first note that across 5 networks with widely different behaviors, the values of $\overline{T_r}$ and $\overline{\lambda_u}$ are remarkably stable. For example, in a network with an average message delay of only 0.025 ms and (practically) no message losses, $\overline{T_r}$ is 0.81 seconds; while in a network with an average message delay of 100 ms and where 1 message every 10 is lost (on average), $\overline{T_r}$ is 0.94 seconds — a very small increase for a network whose message delays and message losses are *orders of magnitude* worse than the first one. Similarly, $\overline{\lambda_u}$ remains close to 6 mistakes per hour in all 5 networks, despite large differences in network behavior.

While this may be surprising at first, it is due to the underlying FD that adapts to changing network behavior in order to meet a given QoS: in all our experiments this QoS remains the same. So, intuitively, the FD layer automatically compensates for the differences in network behavior, and shields the above layers from these differences.

Note that the leader recovery time of $\mathcal{S}_1$ is fairly close to 1 second (across all the 5 networks), and that 1 second is also the maximum failure detection time that we chose for

[3] We also show the 95% confidence intervals of $\overline{T_r}$ and $\overline{\lambda_u}$.

the underlying FD. This is not a coincidence: as we will see in Section 6.6, the time taken to detect the crash of a leader is a dominating component of the leader recovery time.

Finally, we note that $\mathcal{S}_1$ is *unstable*: it makes about 6 mistakes every hour. Recall that a mistake occurs when the service demotes and replaces a functional leader (we also call this an unjustified demotion). In general such a mistake occurs if (a) the underlying failure detector erroneously suspects that the current leader has crashed, or (b) the leader election algorithm itself is unstable. In our experiments, the underlying FD never made a mistake[4], and all the unjustified demotions were due to the leader election algorithm Ω_{id} of $\mathcal{S}_1$: about 6 times every hour, a process with a smaller id than the current leader re-joined the group (after recovering from a crash) and demoted this leader.

In the next two sections, we describe $\mathcal{S}_2$ and $\mathcal{S}_3$, two leader election services that are much more stable than $\mathcal{S}_1$.

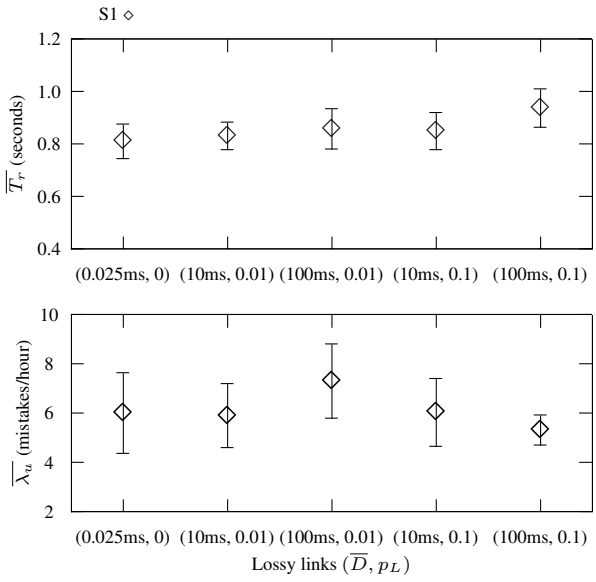

Figure 3. $\mathcal{S}_1$ in lossy networks

6.3. The Service $\mathcal{S}_2$

Description. Roughly speaking, the leader election algorithm of $\mathcal{S}_2$, denoted Ω_{lc}, works as follows (a detailed description appears in [4]). Each process p keeps track of the last time it was suspected of having crashed, called p's *accusation time*, and p selects its leader among a set of processes that is constructed in two stages. In the first stage, p selects its *local leader* as the process with the earliest accusation time among the processes that p believes to be alive (i.e., among the processes from which p recently received an *alive* message). In the second stage, p selects its (global) leader as the local leader with the earliest accusation time among the local leaders of the processes that p believes to be alive. This *(local) leader forwarding* mechanism makes the algorithm robust in the face of link failures. In fact, [4] shows that Ω_{lc} works in a system where every link can be lossy or permanently crash, except for the outgoing links of some non-faulty process (these links must be timely).

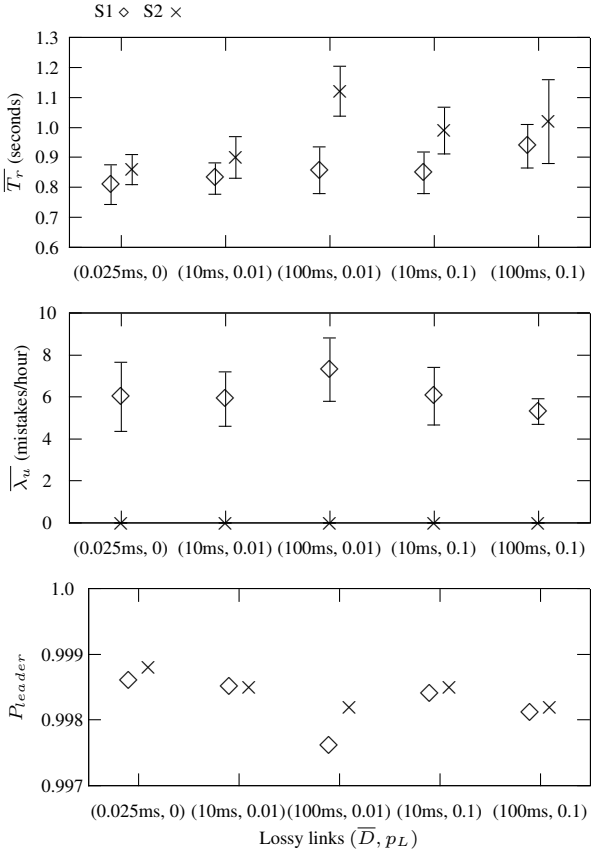

Figure 4. $\mathcal{S}_1$ and $\mathcal{S}_2$ in lossy networks

Experimental Evaluation. We compared $\mathcal{S}_2$ to $\mathcal{S}_1$ with the same settings as in the previous experiments (see Figure 4). First note that, in contrast to $\mathcal{S}_1$, $\mathcal{S}_2$ was perfectly *stable*: in each of the 5 networks considered, $\overline{\lambda_u} = 0$, i.e., no unjustified demotions occurred. This is a remarkable result given the high rate of failures and recoveries (recall that each workstation crashes and recovers every 10 minutes on average) and the very poor quality of the links in some of the networks considered.

The *average leader recovery time* of $\mathcal{S}_2$, however, is slightly larger than the one of $\mathcal{S}_1$ (this is due to the leader forwarding mechanism of $\mathcal{S}_2$ which slightly delays the demotion of a crashed leader). Despite this fact, thanks to its excellent stability, $\mathcal{S}_2$ has a better *leader availability* than $\mathcal{S}_1$ in all 5 networks. In fact, the availability of $\mathcal{S}_2$ is surprisingly high: even when every link drops one message out of 10 (on average), the average message delay is 100ms, and every workstation crashes about every 10 minutes, $\mathcal{S}_2$ provides a leader 99.82% of the time.

[4]This is not surprising since we set the QoS of the FD to make at most one mistake every 100 days (per monitored process).

With both $\mathcal{S}_1$ and $\mathcal{S}_2$, however, the number of *alive* messages that are periodically exchanged is quadradic with the number of processes in the group. We now consider a leader election service with a smaller "message-overhead".

6.4. The Service $\mathcal{S}_3$

Description. The service $\mathcal{S}_3$ is based on a leader election algorithm, denoted Ω_l, that is *communication-efficient*: eventually only the elected leader transmits *alive* messages [2]. As with Ω_{lc}, processes select their leader as the process with the smallest accusation time among a set of processes that compete for leadership. Communication-efficiency is achieved by reducing the set of competing processes, as follows. First, a process p considers that a process q is competing for leadership only if p receives an *alive* message *directly* from q. Second, if p finds that a competing process q has a smaller accusation time (and hence q is a better candidate for leadership than p), p voluntarely drops from the competition for leadership by stopping to send *alive* messages. Note that if p stops sending *alive* messages, other processes may think that p crashed, even though this is not the case. The algorithm includes a mechanism to ensure that such false suspicions do not increase p's accusation time.

Experimental Evaluation. We compared $\mathcal{S}_3$ to $\mathcal{S}_2$ with the same settings as before. Overall, our experiments showed that the message-efficient $\mathcal{S}_3$ is essentially as good as $\mathcal{S}_2$ in networks with lossy links (see Figure 5).

First, $\mathcal{S}_3$ was exceptionally stable: as with $\mathcal{S}_2$, in all our experiments $\mathcal{S}_3$ never demoted an operational leader (we do not show the graph for $\overline{\lambda_u}$ here, since, for all 5 network settings, $\overline{\lambda_u} = 0$ for both $\mathcal{S}_3$ and $\mathcal{S}_2$). Second, the average leader recovery times of $\mathcal{S}_3$ and $\mathcal{S}_2$ were very similar: they were both close to the 1 second upper-bound on the time to detect a crash that we chose for the underlying FD. Third, even in the worst of all 5 network settings (where each of the 12 processors crashes every 10 minutes, each of the 132 links loses 1 every 10 messages, and the average message delay and its standard deviation is 100ms) both services provide an operational leader at least 99.82 percent of the time.

6.5. Trade-off between $\mathcal{S}_2$ and $\mathcal{S}_3$

The previous results indicate that in networks with lossy links, both $\mathcal{S}_2$ and $\mathcal{S}_3$ behave well. In contrast to $\mathcal{S}_3$, however, the leader election algorithm of $\mathcal{S}_2$ was originally designed to tolerate some permanent *link crashes* in addition to lossy links, and so $\mathcal{S}_2$ is potentially more robust than $\mathcal{S}_3$ in extreme network conditions. As we shall see in this section, there is indeed a tradeoff between overhead and robustness: our experiments here indicate that while $\mathcal{S}_3$ is more scalable than $\mathcal{S}_2$, $\mathcal{S}_2$ performs better than $\mathcal{S}_3$ in networks with frequent link crashes.

CPU and bandwidth overhead. The graphs of Figure 6 show the overhead that $\mathcal{S}_2$ and $\mathcal{S}_3$ impose on the system

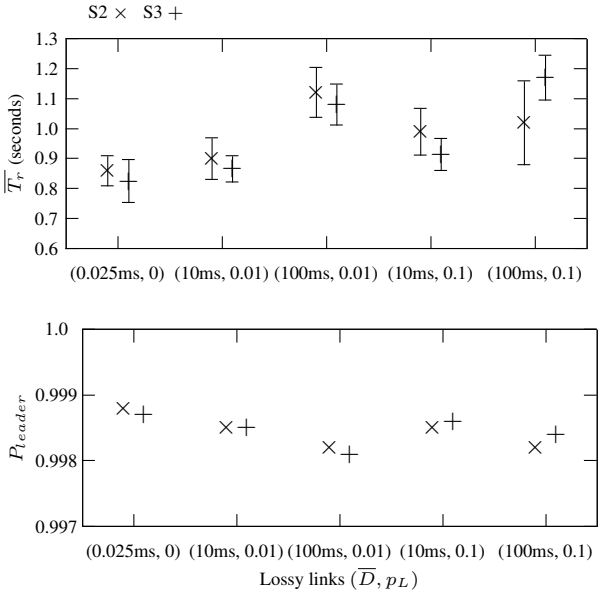

Figure 5. $\mathcal{S}_2$ and $\mathcal{S}_3$ in lossy networks

in terms of CPU and network bandwidth usage. Specifically, they show the overhead that we measured (per workstation) when we run $\mathcal{S}_2$ and $\mathcal{S}_3$ on 4, 8 or 12 workstations, in two very different networks: our (real) local area network, where the average message delay is 0.025 millisecond and there is practically no message loss, and a (simulated) network with lossy links, where the average message delay is 100ms and the probability of message loss is 1/10. In all cases, the QoS of the underlying FD is set to the usual default values explained in Section 6.1.

Note that when the number of workstations increases, the CPU and network utilization of $\mathcal{S}_2$ grow more or less quadratically, but with $\mathcal{S}_3$ they grow only linearly. When the network quality degrades, the overhead of both services also increases. Overall, we can say that both services are lightweight in terms of CPU overhead: in the worst-case here $\mathcal{S}_3$ takes less than 0.04% of the CPU, and $\mathcal{S}_2$ takes at most 0.3% of the CPU. In terms of network bandwidth, in the worst-case $\mathcal{S}_3$ generates at most 6.48 KB/second of message traffic per workstation, while $\mathcal{S}_2$ generates 62.38 KB/s per workstation (which is considerably higher but still reasonable for many networks).

Robustness. We now compare the robustness of $\mathcal{S}_2$ and $\mathcal{S}_3$ in particularly chaotic environments. In our experiments, we run each service on a network of 12 workstations where (on average): (1) every workstation crashes (and later recovers) every 10 minutes, and (2) every link *crashes* every 10 minutes, every 5 minutes, or every minute, for a duration of 3 seconds on average after each crash.[5] The results are shown in Figure 7.

[5]When a link crashes, it completely *disconnects* the receiver from the sender (by dropping all the sender's messages) until the link recovers.

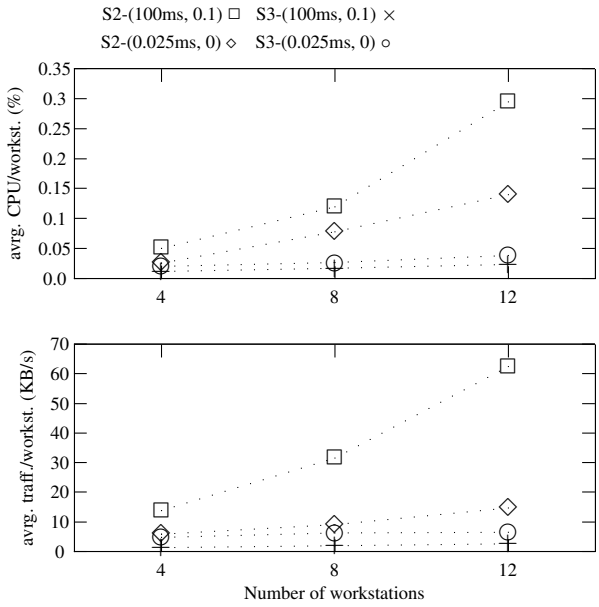

Figure 6. CPU and bandwidth overhead

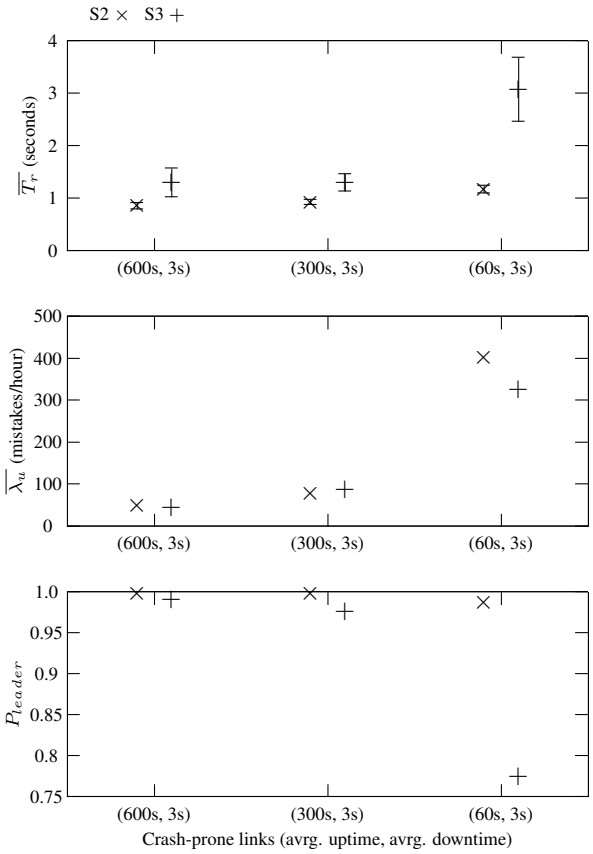

Figure 7. $\mathcal{S}_2$ and $\mathcal{S}_3$ with crash-prone links

First note that in these extreme environments, $\mathcal{S}_2$ has a better *leader availability* than $\mathcal{S}_3$ (see bottom graph). In fact, $\mathcal{S}_2$ is exceptionally robust: even in a network where every link crashes and loses all messages for 3 seconds every 60 seconds, and each workstation crashes every 10 minutes, $\mathcal{S}_2$ is able to provide the group with an operational leader 98.78 percent of the time. In this hostile environment, the leader availability of $\mathcal{S}_3$ is "only" 77.42%. In slightly less demanding settings, namely, when each links crashes about every 5 minutes, the leader availability of $\mathcal{S}_3$ goes up to 97.66% (compared to 99.80% for $\mathcal{S}_2$). Note also that $\mathcal{S}_2$ has a smaller average *leader recovery time* than $\mathcal{S}_3$ (see the upper graph): This is especially clear in the extreme scenario where, on average, every link crashes for 3 seconds every minute: in this case the leader recovery time of $\mathcal{S}_2$ is still close to 1 second, while it grows to about 3 seconds with $\mathcal{S}_3$. Finally, note that both $\mathcal{S}_2$ and $\mathcal{S}_3$ now have unjustified demotions (middle graph). This, however, is unavoidable: in an environment where links lose all messages for about 3 seconds, no FD can detect the crash of processes within 1 second (as we required here) without making mistakes.

6.6. QoS of the FD vs. QoS of $\mathcal{S}_2$ and $\mathcal{S}_3$

We also run some experiments to investigate how the QoS of the underlying FD affects the QoS of the leader election services $\mathcal{S}_2$ and $\mathcal{S}_3$. In these experiments, shown in Figure 8, we run $\mathcal{S}_2$ and $\mathcal{S}_3$ on 12 workstations, each of which crashes and later recovers once every 10 minutes on average, and the message delays and losses are those of our local area network (0.025ms and $p_L \approx 0$). We vary the upper bound T_D^U on the crash detection time of the underlying FD from 0.1 second to 1 second (the other QoS parameters of the FD are set to the default values given in Section 6.1) and see how this affects the QoS of $\mathcal{S}_2$ and $\mathcal{S}_3$.

As Figure 8 shows, T_D^U has a direct influence on both the average *leader recovery time* $\overline{T_r}$ and the *leader availability* P_{leader} of both $\mathcal{S}_2$ and $\mathcal{S}_3$: Roughly speaking, (a) $\overline{T_r}$ remains just a bit smaller than T_D^U, and (b) decreasing T_D^U by some amount improves both $\overline{T_r}$ and P_{leader} by a proportional amount. This indicates that detection time is a large component of the leader recovery time, and that one can effectively control both $\overline{T_r}$ and P_{leader} by setting T_D^U.[6]

7. Summary and concluding remarks

We evaluated our leader election service under a wide variety of settings to encompass a large range of possible applications and networks. In particular, we considered networks where the probability of message loss ranges from (practically) 0 to 1/10 and the average message delay ranges from 0.025ms to 100ms, as well as networks where communication links may completely disconnect for some periods of time.

[6] It should be noted that while decreasing T_D^U improves both $\overline{T_r}$ and P_{leader}, this also increases the cost of running the service. But even if we decrease the failure detection time to a very small value the cost of running $\mathcal{S}_3$ remains low: with $T_D^U = 0.1$ second, $\mathcal{S}_3$ took only 0.1% of the CPU and generated 12.6 KB/s of traffic per workstation; $\mathcal{S}_2$ took 1.23 % of the CPU and generated 135.17 KB/s of traffic per workstation.

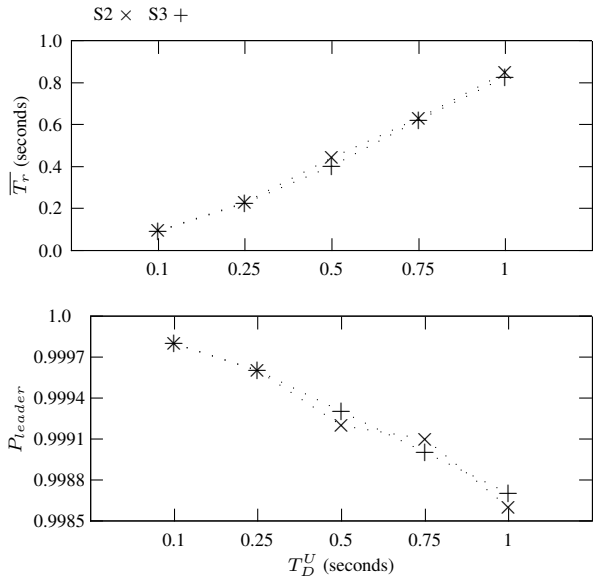

Figure 8. Effect of T_D^U on the QoS of $\mathcal{S}_2$ and $\mathcal{S}_3$

Overall, we found that two versions of the service, namely, $\mathcal{S}_2$ and $\mathcal{S}_3$, behave remarkably well in extremely unfavorable conditions, i.e., in networks with very high processor failures and very poor communication links. We believe that this robustness is due to the combination of leader election algorithms that were proven to work under weak systems assumptions [2, 4], with an underlying failure detector algorithm that provides some QoS control [5].

In future work, we will explore the expansion of the leader election service to very large networks. We first note that one of our services, namely $\mathcal{S}_3$, seems to be inherently scalable: it is based on a message-efficient leader election algorithm, and, as Figure 6 indicates, its CPU and network bandwidth overhead grows quite slowly with the size of the network. But it may still be too expensive to use $\mathcal{S}_3$ directly to elect a leader among a very large number of processes. There are at least two possible (orthogonal) approaches to do this while keeping the costs down. One is to run the election only among a relatively small number of candidates (the election results are then propagated to the rest of the application processes, who remain passive listeners during the election). The other is to arrange for hierarchical elections.

We believe that the current versions of the service can support both approaches. First, the service already allows each application process in every group g to declare whether it is a "candidate for leadership" in g, and the service elects a leader only among the current candidates in g. Second, the groups semantics can be used to elect a leader at each level of the election hierarchy by mapping "groups" to levels (group of local leaders, group of regional leaders, etc...).

Finally, it is worth noting that because the architecture of the leader election service is modular, the service can be easily "upgraded" by replacing the current failure detector or leader election algorithms with future state-of-the-art ones.

References

[1] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Stable leader election. In *proceedings of DISC'01*, pages 108–122. Springer-Verlag, 2001.

[2] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. On implementing Omega with weak reliability and synchrony assumptions. In *proceedings of PODC'03*, pages 306–314. ACM Press, 2003.

[3] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Communication-efficient leader election and consensus with limited link synchrony. In *proceedings of PODC'04*, pages 328–337. ACM Press, 2004.

[4] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. On implementing Omega with weak reliability and synchrony assumptions. Technical Report HAL-00259018, CNRS - France, November 2007.

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

[6] B. Deianov and S. Toueg. Failure detector service for dependable computing. In *proceedings of FTCS'00*, pages B14–B15. IEEE computer society press, 2000.

[7] A. Fernandez, E. Jimenez, and M. Raynal. Eventual leader election with weak assumptions on initial knowledge, communication reliability, and synchrony. In *proceedings of DSN'06*, pages 166–178. IEEE Computer Society, 2006.

[8] C. Fetzer and F. Cristian. A highly available local leader election service. *IEEE Transactions on Software Engineering*, 25(5):603–618, 1999.

[9] R. Guerraoui and P. Dutta. Fast indulgent consensus with zero degradation. In *proceedings of EDCC'02*, pages 191–208. Springer-Verlag, 2002.

[10] I. Gupta, R. van Renesse, and K. P. Birman. A probabilistically correct leader election protocol for large groups. In *proceedings of DISC'00*, pages 89–103. Springer-Verlag, 2000.

[11] D. Ivan and S. Toueg. An implementation of a shared failure detector service with QoS. 2001. Private Communication.

[12] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[13] L. Lamport. The Part-Time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[14] M. Larrea, A. Fernandez, and S. Arevalo. Optimal implementation of the weakest failure detector for solving consensus (brief announcement). In *proceedings of PODC'00*, page 334. ACM Press, 2000.

[15] D. Malkhi, F. Oprea, and L. Zhou. Omega meets Paxos: Leader election and stability without eventual timely links. In *proceedings of DISC '05*, pages 199–213. Springer, 2005.

[16] A. Mostéfaoui and M. Raynal. Leader-based consensus. *Parallel Processing Letters*, 11(1):95–107, 2001.

[17] R. D. Prisco, B. Lampson, and N. Lynch. Revisiting the Paxos algorithm. *Theoretical Computer Science*, 243(1–2):35–91, 2000.