Anonymous Stabilizing Leader Election using a Network Sequencer*

Matthias Wiesmann[†,‡] Xavier Défago

Japan Advanced Institute of Science and Technology
Asahidai 1-1, Nomi, Ishikawa 923-1292, Japan
E-mail: {wiesmann|defago}@jaist.ac.jp

Abstract

In this paper, we present an anonymous, stable, communication efficient, stabilizing leader election algorithm that works using anonymous communication primitives. The algorithm offers properties similar to that of the Ω failure detector, with the added property of totally ordering the sequence of proposed leaders. The algorithm does not need to know beforehand the identity or the number of processes in the system, and operates using a constant amount of memory. We present the algorithm, discuss performance issues and optimizations and present experimental results of a prototype implementation.

1. Introduction

An important issue in distributed systems is the leader election problem. Leader election is one of the basic building blocks of distributed programming. Many distributed protocols rely at some point on a leader election primitive.

The leader election problem is usually defined as selecting one node among a known set of nodes, but this set might not be known. This is the case when leader election is used during the initial setup of the system. At this time, the identity of the nodes and their number are unknown. When the identity of nodes is unknown, only anonymous communications primitives, like multicast, can be used. We call *anonymous leader election* a leader election protocol that only relies on anonymous communication primitives.

A leader election primitive called Ω is the basis of the Paxos consensus algorithm [10]. The relationship between Ω and the failure detector W is discussed in [4]. The implementation of Ω in a weak synchrony model is discussed in [2]. A time-free leader election primitive is presented in [11]. Aguilera et al. introduced several characteristics for leader election primitives, including stability [1] and communication efficiency [3]. Fernández et al. introduce an anonymous leader election protocol in [7].

In this paper we present an anonymous leader election protocol. This protocol does not need any information about the number, the identity, or the addresses of the nodes that participate in the protocol. Instead, the protocol relies on a anonymous multicast primitive (ip-multicast) and sequence numbers extracted from the networking infrastructure. The sequence numbers can be obtained without any special configuration or programming of said equipment, instead, our algorithm only relies on information available using standard interfaces.

Our leader election protocol offers the following properties: a) eventually all participant agree on one leader b) leaders are selected in the same order on all leaders c) the protocol is stable d) the protocol is communication efficient e) the protocol requires a constant amount of memory. The first property is equivalent to the Ω failure detector [10]. The second means that applications can use the leaders delivered by the leader elector without fears of deadlocks. This is very important when selecting a primary server. The third property means that if the leader does not crash, and the communication links between the leader and the other nodes are timely, no new leader is elected. The fourth property ensures that, eventually, only the leader will send messages on the network. The fifth property compares favourably to the protocol introduced by Fernández et al. [7] that requires a amount of memory proportional to the number of processes in the system. Additionally our protocol does not require a total ordering of process identifiers.

Our protocol is well suited for electing a leader in settings where a fixed networking infrastructure is present, but cannot be reconfigured or reprogrammed to suit the needs of the application. Example include selecting a leader among a set of mobile nodes connected to a single wireless base-station, electing a cluster-head in a computing farm, or configuring the primary server among replicas. The main advantage of our protocol is that it can exploit functionality that is available in existing network equipment without requiring any special configuration or running code on the networking equipment.

This paper is structured as follows: Section 2 describes

*Work supported by MEXT Grant-in-Aid for Scientific Research on Priority Areas (Nr. 18049032) and MEXT Grant-in-Aid for Young Scientists (A) (Nr. 18680007).

†Swiss National Science Foundation Fellowship PA002-104979.

‡Matthias Wiesmann is now affiliated with Google Switzerland, Freigustraße 12, 8002 Zürich.

21st International Conference on Advanced Networking and Applications(AINA'07)
0-7695-2846-5/07 $20.00 © 2007 IEEE

the model, Section 3 describes the algorithm, Section 4 presents the implementation of the reliable broadcast and the failure detector used by the algoritm. Section 5 explains how the network sequencer is implemented. Section 6 discusses optimizations and performance considerations of the algorithm. Section 7 describes the prototype implementation of the protocol and discusses performance. Finally, Section 8 concludes the paper.

2. Model

We assume a set of processes $\Pi = \{p_1 \dots p_N\}$. Processes do not know the number of processes N or the identity of other processes. We do not require any ordering of process identifiers. Processes can crash and recover, new processes can join the system at any time. If a process is eventually forever up, we say it is *good*. Processes do not have access to a shared clock. We assume the existence of three basic communications facilities: a reliable broadcast primitive, a failure detector and a network-sequencer. The protocol does not use any point-to-point communication.

2.1. Reliable Broadcast

We assume the existence of a reliable broadcast primitive called r-broadcast. The call r-broadcast(m) broadcasts message m to all processes. This broadcast primitive is reliable, i.e if the sender is good, then all good processes eventually deliver the message by the way of the primitive r-deliver(m). The implementation of this primitive is discussed in Section 4.

2.2. Failure Detector

We assume the existence of an oracle for detecting the failure of the current leader. The failure detector is a predicate that returns *true* if the current leader is trusted and *false* otherwise. This failure detector can make mistakes: it can suspect a leader that is not crashed, or trust a leader that is crashed. We assume that for a given leader l a) at least fd_{good} good processes eventually suspect l if l is crashed, b) less than fd_{bad} processes suspect l if l is not crashed. The implementation of the failure detector is discussed in Section 4.

2.3. Network Sequencer

We assume a network sequencer is available in the system. The network sequencer is a shared facility that returns monotically increasing positive natural numbers. The sequencer is a facility that never fails (it is part of the network) and can be queried by any node. The sequencer has the following properties:

Increase If a query to the sequencer returns v_a, then any later query will return a value $v_b > v_a$.

K-loss If a query returns value v_a, and the query that immediately follows returns v_b, then $(v_b - v_a) \leq k$.
If $k = 1$, we say the sequencer is perfect.

The implementation of the sequencer is discussed in Section 5.

2.4. Constants

We assume the existence of a constant R. This constant can be any positive integer, i.e $R \in \mathbb{N}^+$. In order for the algorithm to be live, we need that $F < R \leq G$. G is the minimal number of good processes whose failure detector is correct and F is the upper bound on lost sequence numbers and false suspicions in one round of the algorithm. So $G \leq fd_{good}$, and $F \leq (k \cdot f) + (fd_{bad})$, where f is the maximum number of processes that can crash during one round of the protocol. Constant k is given by the network sequencer and constants fd_{good} and fd_{bad} are given by the failure detector.

3. Algorithm

The principle of the algorithm is the following: when there is no leader, or when the current leader is suspected, process p proposes itself to become a new leader. To do this, process p gets a sequence number and builds a *token*. A token contains the identity of the sender (p) and the sequence number acquired by p. In order to avoid situations where each suspicion or late message changes the leader, the space of sequence number is divided in *rounds* of R values. We say a round is *closed* if values from a subsequent round have been received. A round that is not closed is *open*. The current leader is the process that has broadcast the highest sequence number in the last closed round.

Basically, tokens acts as proposals for a new leader. If at least R such proposals are sent, a new leader is elected. Each node keeps track of the absolute highest sequence number and the highest sequence number in the last closed round. As the algorithm only has to manage those two elements, the memory requirements are constant and do not depend on the number of participating processes.

The pseudo code of the algorithm is described in Figure 1. The algorithm involves two concurrent tasks. The *listen* task is responsible for listening to incoming messages extract the token from the message and update the values associated with the last closed round v_{closed} and the highest seen value v_{open}. The *suspector* task is responsible for proposing the current process if there is no current leader, or the current leader is suspected. The run-loop of this task is to propose the current process, wait for a new leader, wait for the leader to be suspected and start again.

Each time a token is delivered, the sequence number is compared to v_{open} and v_{closed}. If the token's sequence number is larger than v_{open}, v_{open} and id_{open} are updated. If the token's sequence number is the largest in the last closed round, v_{closed} and id_{closed} are updated. The leader can

```
variables
    v_closed :=⊥ ;                 ◁ Highest value in closed round
    id_closed :=⊥ ;    ◁ Owner of highest value in last closed
    round, i.e the leader.
    id_open :=⊥ ; ◁ Owner of the highest value in open round.
    v_open :=⊥ ;                   ◁ Highest value in open round.
    s_newLeader ;        ◁ Semaphore signaling a leader update.

task listen begin
    loop
        r-deliver(token) ;          ◁ Blocking receive, wait for
        incoming proposals.
        if token.v > v_open then
            v := v_open ; id := id_open ;   ◁ Save old maximum
            in v
            v_open := token.v ; id_open := token.id ;   ◁ Update
            the current maximum.
            if ⌊v_open/R⌋ > ⌊v/R⌋ then
                v_closed := v ;   ◁ Previous maximum value is
                in closed round.
                id_closed := id ;        ◁ Previous maximum is
                leader.
                signal (s_newLeader) ;   ◁ We have updated the
                leader
        if ⌊token.v/R⌋ = (⌊v_open/R⌋ − 1) then
            if token.v > v_closed then
                v_closed := token.v ;   ◁ We update the largest
                in closed round.
                id_closed := token.id signal (s_newLeader) ;
                ◁ We have updated the leader
end
task suspector begin
    loop
        when ¬ trust-leader() do
            token.id := self ;                  ◁ Sender is self.
            token.v := get-sequence() ;      ◁ Get sequence
            number.
            r-broadcast(token) ;   ◁ Broadcast of the values.
            wait until (s_newLeader) ;
end
```

Figure 1. leader election algorithm

change as a result of two situations: a) a new round starts, and therefore the nodes with the largest value in the previous round becomes the leader b) the leader was not the largest sequence of the last closed round and a token within the same round but a larger sequence number is delivered. This only happens if the node holding the last value of a round is slow.

3.1. Properties

Lemma 1 *A selected leader is an existing process.*
PROOF. Processes only propose themselves. Any selected process therefore exists. □Lemma 1

Lemma 2 *For each closed round r, there is a least one proposed sequence number v_r.*
PROOF. A round r without proposed value v_r implies a gap in sequence numbers that is larger or equal to R, but by according to the model definition, the maximum number of lost sequence number is $F < R$. □Lemma 2

Lemma 3 *For each closed round r, there is a least one leader.*
PROOF. By Lemma 2 there was a least one proposed sequence number for round r. Let $V_r = \{v_1^r \dots v_n^r\}$ be the set of proposed values for round r. The leader is given by $\max(V_r)$, which always returns a value. □Lemma 3

Lemma 4 *If $G \geq R$ good processes suspect the current leader a new leader is elected.*
PROOF. If G processes suspect the current leader, they will propose themselves as leader. As they are good, their tokens will eventually be received. This means at least G sequence numbers will be proposed. As $G \geq R$ at least one new round will be opened and the previous closed. Thus a new leader will be selected. □Lemma 4

Lemma 5 *If all good processes are suspected by less than $f < \frac{R}{k}$ processes, new leaders eventually stop being elected.*
PROOF. Let v the current value of the sequencer, the number of sequences needed to close round r is $\delta_r = R - mod(v,R)$. The suspecting processes can broadcast s sequences, such as $s \leq k \cdot f < R$ values. If $s < \delta$ no new round can be started, and therefore no new leader is elected. If $s \geq \delta$ a new round $r+1$ will be started. In this new round $\delta_{r+1} = R - s + \delta_r \geq 1 + \delta_r$. As $\delta_{r+1} \geq 1 + \delta_r$, we see that δ is strictly monotically increasing. Thus eventually, $\delta_{r+n} > s$ and new leaders are not elected anymore. □Lemma 5

Lemma 6 *The protocol is stable.*
PROOF. If the leader does not crash, the link from the leader to all processes is timely and no message loss occurs, the leader is not suspected. So no new round is started and no new leader is selected. □Lemma 6

Lemma 7 *The protocol is communication efficient.*
PROOF. Processes only send messages if either they are the leader, or suspect the current leader. If the current leader is not suspected, only the leader sends messages. □Lemma 7

Lemma 8 *If the algorithm returns two leaders L_a and L_b on two processes p_1 and p_2, they are returned in the same order.*
PROOF. Every selected leader has an unique associated sequence value. Sequences have a strict ordering. A leader can only be replaced by a leader with a larger sequence number. □Lemma 8

4. Implementing the reliable-broadcast and the failure detector

The reliable broadcast primitive and the failure detector required by the algorithm described in Section 3 are implemented together using the same unreliable ip-multicast

```
input : η Heartbeat period, ε Suspicion timeout
begin
    t_last := now ;                    ◁ Delivery of last heartbeat
    V := ∅ ;                           ◁ Set of delivered values
end
procedure r-broadcast(token) begin
    while v_closed = token.v do
        broadcast(token) ;             ◁ Send token
        sleep(η) ;                     ◁ Sleep for heartbeat period.
end
procedure deliver(token) begin
    if token.v ∉ V then
        r-deliver(token) ;             ◁ First time we see token, so we
        deliver.
        V := V ⋃ {token.v} ;           ◁ Update V
    t_last := now ;                    ◁ Update time of last heartbeat.
end
function trust-leader begin
    return now − t_last < ε ;
end
```

Figure 2. broadcast and failure detection algorithms

primitive. The reliable broadcast is implemented by periodically sending the message. This ensures eventual delivery and acts as a heartbeat for the failure detector. This is similar to the implementation of Ω described in [9, 7].

Figure 2 illustrates the algorithm of both the reliable multicast and the heartbeat functionality. Variables η and ε represent the heartbeat interval and the time-out period respectively. This algorithm uses a very simple fixed time-out scheme, more complex failure detection schemes [12, 8, 6, 5] could also be used.

5. Implementing the Sequencer

The sequencer is implemented using functionality that is present in many pieces of network equipment: the Simple Network Management Protocol (SNMP) interface. Routers, smart switches, wireless access-points and other network equipments very often contain an SNMP agent, a small server that responds to SNMP queries. This agent is used to implement the sequencer described in Section 2. As the sequencer is implemented inside the network infrastructure, the assumption that it does not fail is valid – or more precisely, if the sequencer fails, the network fails and leader election is impossible anyways. We assume that the sequencer runs on a critical part of the network infrastructure. If this critical infrastructure is hardened or replicated, so is the sequencer.

The sequencer is implemented by sending a special query to the SNMP agent. The query reads the value SNMPv2-MIB::snmpInGetRequests.0 (1.3.6.1.2.1.11.15.0) of the Management Information Base (MIB). This value reflects the number of the SNMP requests the agent has received, and is increased each time a request reaches the agent. As requesting this value is also an SNMP request, the counter is increased with every request. There are two issues with sequence number implemented this way: sequence gaps, and counter resets.

Sequence number normally only increase by one for each request, but sequence gaps can appear if external programs query the SNMP agent or if the response packet gets lost. The problem of lost response packets arises because SNMP generally runs on top of the UDP protocol, the request to the agent is transmitted in one UDP packet and the response another one. Both the request and the response can get lost, but only the loss of the response causes a sequence gap.

A counter reset can occur for two reasons. First, the counter is implemented as a 32 bit integer, if the counter reaches 2^{32}, it simply resets to 0, this is a *wrap-around*. Second, the counter is reset when there is an equipment restart. A restart results in all the data-structures been cleared, including the counter we use, we call this a *restart-reset*. Concretely, if a process receives from the sequencer a value smaller that R, it broadcasts a special message that restarts the whole algorithm.

A more advanced approach would be to discriminate between wrap-around or a restart-reset. In the case of a wrap-around, the algorithm does not need to be reset, instead we can use a sliding window technique and interpret a value v smaller than 2^{31} as being $v+2^{32}$. In the case of a restart-reset, the algorithm is effectively reset. The difference between wrap-around and restart-reset can be detected by also querying the value SNMPv2-MIB::sysUpTime.0 of the MIB. This value contains the uptime of the equipment and is reset in case of restart-reset, but not in the case of a wrap-around. We have decided not to implement this technique, given the fact that both wrap-arounds and restart-resets are rare, the performance improvement does not justify the added complexity.

6. Performance Considerations

While implementing the protocol, we improved performance with a few optimizations. A first optimization enables joining processes to accept the current leader with a single message. Basically, as the state of the protocol is very small (two integer values) we send it fully with every message. The leader adds the maximum known value v_{open} to every token it broadcasts. A recovering process parses the two sequences number contained in the token sequentially: first in handles v_{open} and then the value of the leader. This way, when a recovering process receives a token from a leader it select that leader immediately. This optimization does not change the way processes choose the current leader (Lemma 9) and can be seen as a case of piggy-backing messages.

This optimization makes sense if joining processes wait for some time to see if there is an existing leader in the system. Starting and recovering processes start with a special leader $\perp$, and can only propose themselves if they do not hear from a existing leader before some initial time ε_0. To avoid initial burst situations, ε_0 is chosen uniformly in the interval $]0 \ldots \varepsilon]$. We assume that $\frac{\varepsilon}{2} > \eta$, that is the failure detector

	Clients	Resp. Time	Throughput	Link Type
SF-0224FS	1	14.783 ms	67.646 seq/s	100 Mb/s
SF-0224FS	2	27.068 ms	73.889 seq/s	100 Mb/s
PCI FMG-24K	1	10.181 ms	98.165 seq/s	1000 Mb/s
PCI FMG-24K	2	20.17 ms	49.57 seq/s	1000 Mb/s

Table 1. Performance of network sequencer on switches

time-out is larger than the interval of two heartbeats. This is justified by the fact that heartbeats are transported using an unreliable mechanism and we want to avoid a suspicion caused by a single lost heartbeat.

If a starting process receives the message from a leader l during this time, it adopts l as their leader. This optimization ensures that starting or recovering processes do not force new leaders (Lemma 10). The main drawback of this optimization is that it adds some latency to the protocol. If we assume that R processes start at $t = 0$, then the decision time will be $t_{prot} + \frac{\varepsilon}{2}$, where t_{prot} is the time needed to run the protocol.

Lemma 9 *Leaders selected using the recovery optimization are the same as those of the algorithm in Figure 1.*

PROOF. If process l is a leader, then $\lfloor \frac{v_{closed}}{R} \rfloor + 1 = \lfloor \frac{v_{open}}{R} \rfloor$. The algorithm behaves as if two messages were delivered, first the proposal of the current leader with value v_{closed} is delivered, then the value v_{open} is delivered. Value v_{closed} is the last value of a closed round, so l is selected as leader. $\square_{\text{Lemma 9}}$

Lemma 10 *Unstable processes (processes that crash and recover) behave like good processes.*

PROOF. Recovering processes can only propose themselves if they suspect the leader $\perp$ and have received no proposal from an actual leader. This can only happen if the failure detector make a wrong suspicion. This behaviour is the same as for good processes. $\square_{\text{Lemma 10}}$

One important aspect of the performance of the protocol is the performance of the network sequencer. For a new leader to be elected, assuming no crashes and a perfect sequencer we need R sequences. If all processes start at time $t = 0$, then the leader will be elected at time $t_{election}$, as defined by the following formula:

$$t_{election} = \frac{\varepsilon}{2} + t^{par}_{oracle} + R \cdot t^{seq}_{oracle} + t_{net} \qquad (1)$$

Where $t_{net}$is the time needed to broadcast a message. The time needed to get a sequence from the network oracle t_{oracle} is divided in two parts: $t^{par}_{oracle}$and t^{seq}_{oracle}($t_{oracle} = t^{par}_{oracle} + t^{seq}_{oracle}$). Value t^{par}_{oracle}represents the time that is independent between processes, t^{seq}_{oracle}represents the processing time of the sequencer that is in contention between all processes. We assume that network contention is negligible.

We see that if R is large, the term $R \cdot t^{seq}_{oracle}$ will dominate. As the network oracle is implemented on network equipment with weak processing power, the throughput rate is limited, this leads to a large value for t^{seq}_{oracle}. Table 1 gives the performance of network oracle implemented on two different switches, with different number of clients. The average response time measured in milliseconds and the average throughput measured in sequences per second. Measures were done by querying for 10000 sequence numbers, we could not observe any lost sequence.

As the performance of the network oracle will have a large impact on the performance of the system, it makes sense to adapt the other parameters to the performance of the network oracle. In particular, as leader election is rate limited, suspicions from the failure detection should be limited to a similar rate. Therefore it makes no sense to have $\varepsilon \leq t_{oracle}$.

7. Implementation

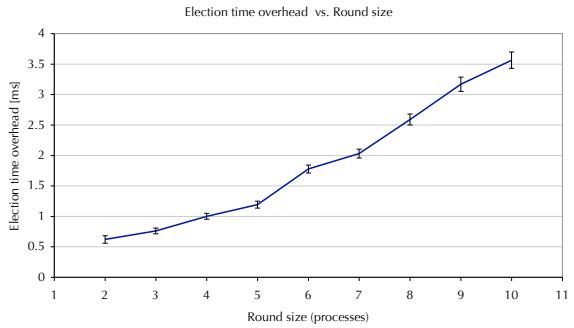
In order to validate this protocol, we implemented it in Java. The network oracle was implemented using the SNMP4J framework. Messaging was done using ip-multicast messages. The implemented code includes the optimizations discussed in Section 6. In order to measure the performance of the algorithm, we ran in on a computing cluster composed of nodes linked by a smart switch.

The nodes are x-series 305 machines with each an Intel Pentium 4 processor at 2.8 GHz and 2 GB of memory. The operating system is Linux Fedora Core 4 with kernel 2.6.15. The Java virtual machine is Sun's JRE 1.5.0-06. The SNMP4J framework is version 1.6c. The switch is a PCI FMG-24K switch with 24 gigabit ethernet ports, and was used both for linking the nodes and implementing the network sequencer. The network links are all full-duplex 1000 Mb/s. The ping time between hosts was around 0.15 ms.

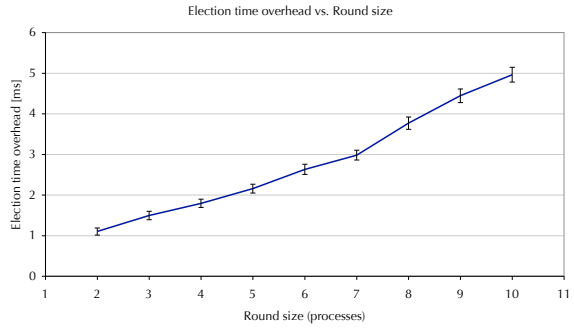
The main metric for evaluating the algorithm was the election overhead. The election overhead is the elapsed time between the moment a node proposes itself as leader and the moment a new leader is elected. The experiment consisted in running the protocol on ten nodes, every five seconds, the protocol is reset, this triggers a new leader election. The experiment was run for ten minutes.

Figure 3a shows the election overhead as a function of the round size with a system with 10 nodes and the following parameters $\eta = 10\ ms, \varepsilon = 30\ ms$. Those values are much too agressive for normal use, and were intended to observe the behaviour of the protocol in load situation. Higher loads tend to overload the sequencer and gave: election overhead does not diminish but becomes less stable. Every point in the graph is the average of at least 3000 measures, the vertical bars represent a 99% confidence interval. The actual election time will be $overhead + \frac{\varepsilon}{2}$.

We see that the election time increases roughly as a linear function of the size of the round. If we compare the detection overhead to the response time of the sequencer given in Table 1, we see that it is smaller. This is caused by the way the protocol works. On average $\frac{R}{2}$ proposals are needed to elect a new leader. So when a process decides to propose



(a) $N = 10, \eta = 10\ ms, \varepsilon = 30\ ms$



(b) $N = 10, \eta = 50\ ms, \varepsilon = 150\ ms$

Figure 3. Election overhead vs. round size

itself, there is a good chance that the protocol has already selected a new leader. In fact in many cases, especially with small values of R, processes don't have the time to suspect the leader before one is selected. In some rare cases, the time needed to select a leader is around 15 ms, that is the time needed to get a token from the sequencer. So roughly, the larger then number of processes relatively to the round size R, the faster the protocol. This is interesting for large systems where R will be relatively small compared to the number of processes.

We repeated the same experiment with larger values for heartbeat and timeout. Those values are more representative of the time-outs of a cold-start or a background book-keeping situation. Figure 3b shows the results for $\eta = 50\ ms$, $\varepsilon = 150\ ms$, the curve is largely similar, with overall longer response times.

8. Conclusion

We have presented a stable, communication efficient, anonymous, stabilizing leader election algorithm which requires a constant amount of memory. We have shown how to implement the different abstractions required by the algorithm. The algorithm is well suited for selecting a primary server during configuration phases, as the algorithm does not put any requirement on the set of processes and their addressing schemes. Experimental results show that with today's switching equipment, the algorithm can select a leader in roughly 30 ms. While this is unsuitable for high-performance computing, this is is a reasonable overhead for configuration phases.

Acknowledgments

We would like to thank Péter Urbán and Yàn Yáng, for insightful discussions about this protocol and giving feedback for the paper.

References

[1] M. K. Aguilera, C. Delporte-Gallet, and H. Fauconnier. Stable leader election. In *Proc. of the 15th Int. Conf. on Distributed Computing (DISC'01)*, volume 2180, pages 108 – 122. Springer-Verlag, 2001.

[2] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. On implementing Ω with weak reliability and synchrony assumptions. In *Proc. of the 22nd annual Symp. on Principles of Distributed Computing (PODC'03)*, pages 306–314, 2003.

[3] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Communication-efficient leader election and consensus with limited link synchrony. In *Proc. of the 23rd annual symp. on Principles of distr. computing (PODC '04)*, pages 328–337. ACM, 2004.

[4] F. Chu. Reducing Ω to W. *Information Processing Letters*, 67(6):289–293, September 1998.

[5] X. Défago, P. Urbán, N. Hayashibara, and T. Katayama. Definition and specification of accrual failure detectors. In *Proc. of the Int. Conf. on Dependable Systems and Networks (DSN'05)*, pages 206–215, 2005.

[6] L. Falai and A. Bondavalli. Experimental evaluation of the QoS of failure detectors on wide area network. In *Proc. of the Int. Conf. on Dependable Systems and Networks (DSN'05)*, pages 624–633, June 2005.

[7] A. Fernández, E. Jiménez, and M. Raynal. Eventual leader election with weak assumptions on initial knowledge, communication reliability and synchrony. In *Proceedings of the Int. Conf. on Dependable Systems and networks (DSN'06)*, pages 166–189. IEEE, June 2006.

[8] N. Hayashibara, X. Défago, R. Yared, and T. Katayama. The ϕ accrual failure detector. In *Proc. of the 23rd Int. Symp. on Reliable Distributed Systems (SRDS'04)*, pages 66–78, October 2004.

[9] E. Jiménez, S. Arévalo, and A. Fernández. Implementing the Ω failure detector with unknown membership and weak synchrony. Technical Report RoSaC-2005-2, Universidad Rey Juan Carlos, Madrid, Spain, 2005.

[10] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[11] A. Mostefaoui, M. Raynal, and C. Travers. Crash-resilient time-free eventual leadership. In *Proc. of the 23rd Int. Symp. on Reliable Distributed Systems (SRDS'04)*, pages 208–217. IEEE, October 2004.

[12] R. C. Nunes and I. Jansch-Pôrto. QoS of timeout-based self-tuned failure detectors: The effects of the communication delay predictor and the safety margin. In *Proc. of the Int. Conf. on Dependable Systems and Networks (DSN'04)*, 2004.