Ω Meets Paxos: Leader Election and Stability Without Eventual Timely Links

Dahlia Malkhi[1], Florin Oprea[2,⋆], and Lidong Zhou[3]

[1] Microsoft Research Silicon Valley and the Hebrew University of Jerusalem
[2] Department of Electrical and Computer Engineering, Carnegie Mellon University
[3] Microsoft Research Silicon Valley

Abstract. This paper provides a realization of distributed leader election without having any eventual timely links. Progress is guaranteed in the following weak setting: Eventually one process can send messages such that every message obtains f timely responses, where f is a resilience bound. A crucial facet of this property is that the f responders need **not** be fixed, and may change from one message to another. In particular, this means that no specific link needs to remain timely. In the (common) case where $f = 1$, this implies that the FLP impossibility result on consensus is circumvented if one process can at any time communicate in a timely manner with one other process in the system.

The protocol also bears significant practical importance to well-known coordination schemes such as Paxos, because our setting more precisely captures the conditions on the elected leader for reaching timely consensus. Additionally, an extension of our protocol provides leader *stability*, which guarantees against arbitrary demotion of a qualified leader and avoids performance penalties associated with leader changes in schemes such as Paxos.

1 Introduction

A fundamental design guideline pioneered in the Paxos protocol [1] and later employed in numerous coordination protocols is to separate *safety* properties from *liveness* properties. Safety must be preserved at all times, and hence, its implementation must not rely on synchrony assumptions. Liveness, on the other hand, may be hampered during periods of instability, but eventually, when the system resumes normal behavior, progress should be guaranteed. In various coordination protocols such as Paxos, liveness hinges on a separate leader election algorithm, with the problem of finding a good leader election algorithm left open.

It is well known in the theory of distributed computing that liveness of consensus cannot be guaranteed in a purely asynchronous system with no timing assumptions [2]. Ω is known to be the weakest failure detector [3,4] that is sufficient for consensus, hence provides the liveness properties of consensus. Ω

⋆ Work done during a summer internship at Microsoft Research Silicon Valley.

P. Fraigniaud (Ed.): DISC 2005, LNCS 3724, pp. 199–213, 2005.
© Springer-Verlag Berlin Heidelberg 2005

essentially implements an eventual leader election, where all non-faulty processes eventually trust the same non-faulty process as the leader.

While Ω captures the abstract properties needed to provide liveness, it does not say under which pragmatic system conditions is progress guaranteed. It leaves open the interesting questions of what synchrony conditions should be assumed when implementing Ω and what additional properties would yield an ideal leader election algorithm for practical coordination schemes such as Paxos.

A revisit of Paxos. In this paper, rather than cooking up arbitrary synchrony assumptions and additional properties, we derive the desired features of our protocols from Paxos, a cornerstone coordination scheme employed in various reliable storage systems such as Petal [5], Frangipani [6], Chain Replication [7], and Boxwood [8].

At a high level, Paxos is a protocol for a set of processes to reach consensus on a series of proposals. With a leader election algorithm, a process p that is elected leader first carries out the `prepare` phase of the protocol. In this phase, p sends a `prepare` message to all processes to declare the *ballot number* it uses for its proposals, learns about all the existing proposals, and requests promises that no smaller ballot numbers be accepted afterwards. The `prepare` phase is completed once p receives acknowledgments from $n - f$ processes. Once the `prepare` phase is completed, to have a proposal committed, leader p initiates the `accept` phase by sending an `accept` message to all processes with the proposal and the ballot number it declares in the `prepare` phase. The proposal is *committed* when p receives acknowledgments from $f + 1$ processes. Whenever a higher ballot number is encountered in the `prepare` phase or the `accept` phase, the leader has to initiate a new `prepare` phase with an even higher ballot number. This could happen if there are other processes acting as leaders, unavoidable in an asynchronous system.

To implement a replicated state machine, Paxos streamlines a series of consensus decisions. A new leader p carries out the `prepare` phase once for all its proposals. After the completion of the `prepare` phase, p carries out only the `accept` phase for each proposal until a new leader emerges by initiating a new `prepare` phase.

Our goal is to distill the conditions under which Paxos can have new proposals committed in a timely fashion and to provide a leader election protocol exactly under those conditions. Therefore, we make the following observations:

- After an initial `prepare` phase, in order for a leader p to make timely progress, it suffices for p to obtain timely responses for its `accept` message from any set of $f + 1$ processes (or f processes besides itself). The set could change for different `accept` messages.
- Any leader change incurs the cost of an extra round of communication for the `prepare` phase.

Contribution. Complying with the conditions under which we wish to enable progress in Paxos, our leader algorithm features the following two desired properties[1]:

[1] Formal definitions of these properties are provided in the body of the paper.

First, the algorithm guarantees to elect a leader without having any eventual timely links. Progress is guaranteed in the following surprisingly weak setting: Eventually one process can send messages such that every message obtains f timely responses, where f is a resilience bound. We name such a process $\diamond f$-*accessible*. A crucial facet of this property is that the f responders need *not* be fixed, and may change from one message to another. We emphasize that this condition stems from the workings of Paxos, whose safety does not necessitate that the f processes with which a leader interacts be fixed.

Our solution bears the following ramification on the foundations of distributed computing. It implies that the FLP [2] impossibility result on consensus with one failure ($f = 1$) is circumvented if one process can at any time interact in a timely manner with one other process in the system.

No previous leader election protocol provides any guarantee in these settings. In fact, the approach taken in most previous protocols is fundamentally incompatible with this condition, because previous protocols gossip about *suspicions* until the system converges. This does not allow for a leader to communicate at different times with different subsets of the system, as the leader will constantly be under suspicion of some part of the system. Thus, no easy "engineering" of previous protocols can provide progress under the $\diamond f$-accessibility condition.

The second contribution provided by our algorithm is leader *stability*. This is based on the observation that a leader change necessitates an execution of a `prepare` phase by the new leader, an often costly operation. We therefore embrace the notion of stability that a *qualified* leader not be demoted, where a leader is considered qualified if it remains capable of having proposals committed in a timely fashion. For Paxos, when $n = 2f + 1$ holds, a leader is qualified if it is non-faulty and maintains timely communication with a set of f other processes at all times, with the set possibly changing over time.

An important practical measure for a leader election protocol is its communication complexity [9]. It is desirable that under a *steady state*, where a qualified leader operates without being suspected by any other process, only the leader incurs periodic communication with the rest of the system (hence achieving $O(n)$ steady-state link-utilization communication complexity.) In Section 6, we sketch an extension of our protocol that achieves this ideal. This extension works only with the basic version of our protocol for Ω, which does not uphold stability. It is left as an open question whether a stable leader election algorithm with $O(n)$ steady-state message complexity exists under the condition of $\diamond f$ accessibility.

2 Related Work

Our review of previous work concentrates on the two properties of interest to us: synchrony conditions and leader stability.

On synchrony conditions. A simple solution for the leader election problem is as follows [1,10]. Periodically send alive messages from all to all, and let each process collect data on all the processes it heard from within the last broadcast

period. Each process elects as leader the process with the lowest process *id* from its view. This implementation requires that eventually all n^2 communication links become timely with a known communication bound.

A number of papers [11,9] relax this by assuming an *unknown* communication bound. The reduction to the known bound model involves gradually increasing timeout periods until no false alarms incur on the current leader. This "trick" may be used in almost all leader-election protocols, as is done, e.g., in [11,9,12,13,14,15]. Nevertheless, all communication links are required to be eventually synchronous.

Aguilera et al. further relaxes the model to one that has a process maintaining eventually timely links with the rest of system [12] and to one that has a process whose outgoing links to the rest of the system are eventually timely [13]. In [13], a single correct process called $\diamond$-source is assumed to have outgoing non-lossy and timely links eventually . Their protocol works by processes sending accusation messages to one another when they timeout. Intuitively, every process converges on the suspicions of the $\diamond$-source process, since its accusations are guaranteed to arrive timely at their destinations.

More recently, and most relevant to our work, there are several pieces of work that require surprisingly weak synchrony conditions for implementing Ω and consensus. This line of work limits the scope of timely links from the correct pivot process to only a subset of the system. There are two main flavors, one deals with failure-detection abstractions without explicit mentioning of synchrony conditions, and the second builds directly over partial synchrony conditions. We start with the first approach, which historically precedes the second.

The work of [16,17] introduces the notion of *limited scope failure detectors*, where the scope of the accuracy property of an unreliable failure detector is defined with respect to a parameter (x) as the minimum number of processes that must not erroneously suspect a correct process to have crashed. This yields failure detector classes S_x (respectively, $\diamond S_x$), whose accuracy properties are required to hold only on a subset of the processes whose size is x. The usual failure detectors S (respectively, $\diamond S$) implicitly consider a scope equal to the total number of processes. A limited-scope detector in the classes S_k or $\diamond S_k$ is straight-forward to implement using periodic alive messages and timeouts, given a system in which one correct process (eventually) has x outgoing timely links. Therefore, under these conditions, a possible construction of Ω is to as follows: first implement $\diamond S_x$; then transform $\diamond S_x$ to $\diamond S$ [14]; finally transform $\diamond S$ to Ω [18].

Aguilera et al. [15] adopts a more direct approach. Define a process p to be a $\diamond f$-source if eventually it has f outgoing links that are timely. Any of the f recipient endpoints of these links may be faulty. Assuming a bound f on the number of crashed processes, Aguilera et al. [15] presents an Ω construction with the existence of one correct $\diamond f$-source. The protocol counts suspicions of processes about all other processes and exchanges vectors of suspicion-counters. Each process elects as leader the process with the lowest suspicion counter, breaking ties by process *id*s. Intuitively, the suspicion counters of crashed processes

grow indefinitely, whereas the $\diamond f$-source has a guaranteed bounded suspicion-counter. This guarantees that eventually a correct process is elected as leader (among all the ones whose counters are bounded), and furthermore, it remains so permanently because all counters are non-decreasing.

Both the S_f condition and the $\diamond f$-source condition are neither weaker nor stronger than ours: Let p denote, respectively, the pivot correct process that upholds any of these models. The $\diamond f$-source assumption and the $\diamond S_f$ accuracy assumption require timeliness only on f *outgoing* links from p, and no correctness of the f recipients. Our $\diamond f$-accessible assumption requires f bi-directional timely links from p, as well as correctness from the f recipients, which are stronger assumptions. However, in $\diamond f$-source, the set of f links is *fixed* throughout the execution, as is the limited-scope subset of $\diamond S_f$, whereas $\diamond f$-accessible allows the f links to vary in time, which is a weaker assumption.

Although formally these models are incomparable, we note that our assumptions are strongly motivated by practical needs, particularly those of the Paxos protocol. In Paxos, if there is a single leader, the leader can carry out the `accept` phase and make progress so long as it is able to communicate with f processes. This is exactly the condition under which our Ω implementation is guaranteed to operate. In particular, the leader may in realistic settings have a "moving set" of f timely links. But so long as at any moment, some set of f links are timely, our protocol can guarantee progress. Under these conditions, the $\diamond f$-source assumption does not hold, nor does $\diamond S_f$, and the protocols of [14,15] may fail.

Leader stability. The only previous work we are aware of that considers some form of leader stability is the protocol of Aguilera et al. [12]. Their notion of stability relates to a leader that is recognized by all non-faulty processes as leader. For practical consensus protocols such as Paxos, this condition might have limited value, because no process inside the system can know when a leader is known to all others. In Paxos, a process must know whether it is a leader in order to decide whether to initiate the `prepare` phase. Therefore, our stability condition uses the leader's own perspective as the determining time to when its leadership stabilizes. This is what Paxos needs to avoid having leaders being arbitrarily de-crowned due to unnecessary `prepare` messages.

3 Informal Model

The system consists of a set P of n processes, each pair of which can directly communicate by sending and receiving messages over a bi-directional link. Each process is equipped with a drift-free local clock. Clocks of different processes need not be synchronized. When we reason about the system, we often use a global wall-clock t, which is not known or used by the processes within the system.

Each process executes a sequence of steps triggered either by message reception or timer expiration. In a step, a process may perform any number of local computations, send messages, and set timers. For simplicity, we denote the time it takes to perform a step as zero.

Process and Communication Faults. Processes may fail by crashing permanently, and otherwise are non-faulty. A failure pattern F_p is a function from wall clock time to sets of processes that have crashed by that time. We say that p is non-faulty at time t if $p \notin F_p(t)$. We say that p is non-faulty if it is always non-faulty. There is a known resilience bound $f \leq \lfloor \frac{n-1}{2} \rfloor$ on the number of crashed processes.[2]

Communication links are reliable, in the sense that no message from a non-faulty process can be dropped, duplicated, or changed, and no messages are generated by the links.

Communication Synchrony. The conditions regarding timeliness of links are at the heart of our investigation. There is a known upper bound δ on the round-trip delay of messages, but it does not hold on all pairs of processes at all times. What is known is that eventually there is one process that is able to exchange messages within the δ delay with f other processes. We will now make this notion precise.

Definition 1. *Let $\overline{(p,q)}$ denote the communication link between p and q. We say that $\overline{(p,q)}$ is* timely *at time t if any message sent by p to q at time t receives a response within δ time. Note that if q becomes faulty before handling p's message, or q is slow to respond, then by definition the link is not timely.*

Definition 2. *A process $p \in P$ is said to be* f-accessible at time t *if there exist f other processes q such that the links $\overline{(p,q)}$ are timely at t.*

Our synchrony requirement is the following.

Definition 3. *($\Diamond f$-accessibility) There is a time t and a process p such that for all $t' \geq t$, p is f-accessible at t'.*

Note that the definition of f-accessibility allows a process p to be considered f-accessible even if the sets of f processes accessed by p at different times change. This property is fundamentally more practical than fixing a subset with which p must interact forever. This definition is derived from the way consensus protocols like Paxos [1] and revolving-coordinator consensus [3] operate.

We also note that there are several known ways to weaken our model with variations that bear practical importance. First, it is easy to extend the model to account for a non-zero bound on local processing time and clock drifts, but this would just be a syntactic burden. Second, it is possible to relax the assumption that the communication round-trip bound δ is a priori known. The trick for overcoming this uncertainty is to start with an aggressively-low guess of δ and gradually increase it when premature expirations are encountered. Most of the

[2] It is easy to generalize the discussion to use *quorum systems* instead of counting processes. A *read/write quorum system* for P, denoted $\mathcal{R}(P), \mathcal{W}(P) \subseteq 2^P$, is a pair of sets of subsets of P, such that every pair $Q_1 \in \mathcal{W}(P), Q_2 \in \mathcal{W}(P) \cup \mathcal{R}(P)$ has a non-empty intersection, $Q_1 \cap Q_2 \neq \emptyset$. Each subset is called a *quorum*. Quorums generalize thresholds as follows. Operations on $(f+1)$-subsets are replaced with operations on write quorums; operations on $(n-f)$-subsets are replaced with operations on read quorums.

claims in this paper can be adapted to reflect this technique of learning δ. For simplicity, we omit this from the discussion. Finally, our non-timely reliable links may be easily replaced with fair lossy-links as in [15], which are links that deliver infinitely many times any message-type that has been sent infinitely often. This requires repeatedly sending messages until acknowledged, and once again, is omitted from the discussion.

Problem statement. Our goal is to construct in our model a weak leader Ω, defined as follows [3]: Ω provides every process q at any time t with a local hint $\Omega_q(t)$, such that the following holds:

Definition 4 (Ω). *There exist a time t and a non-faulty process p, such that for any $t' \geq t$, every process q that is not faulty at time t' has $\Omega_q(t') = p$.*

4 Ω with $\diamond f$-Accessibility

Our first protocol implements Ω under the $\diamond f$-accessibility condition. The protocol for process p appears in Figure 1. It works as follows.

Each process maintains for itself a non-decreasing *epoch number*, as well as an *epoch freshness counter*. Epochs are implemented using the following data types and variables. An epoch number is a pair that consists of an integer field named *serialNum* and another field named *processId*, which stores either a process *id* or `null`. We assume a total ordering on process *id*s with `null` smaller than any process *id*. Epoch numbers are ordered lexicographically, first by *serialNum* and then by *processId*.

We define a state to be a pair consisting of an epoch-number field named *epochNum* and an integer field named *freshness*. States are ordered lexicographically, first by *epochNum* and then by *freshness*.

A process refreshes its epoch number in fixed periodicity of length Δ, by incrementing the epoch freshness counter and writing it to its *registry*, which is replicated on all processes in the system. If the refresh fails to complete updating the registry at $f+1$ processes within the known δ round-trip bound, the process increases its own epoch number. The vector *registry*[] records locally at each process the latest state it received from others: *registry*[q] is updated upon receipt of a refresh message from q.

Process p records the states it reads of all other processes in a vector named *views*[]. A process updates its view by periodically reading the entire registry vector from $n - f$ processes. Each entry *views*[q] has two fields. One is a *state* field, and the other is a bit called *expired* indicating whether q's state has been continuously refreshed or not. Initially, all *serialNum* and *freshness* fields are zeroed, and *expired* field set to `true`.

The idea is to select as a leader the process with the lowest non-expired epoch number (breaking ties using process *id*s). To assess whether an epoch number has expired or not, every process reads the registry of all processes from $n - f$ processes periodically. The exact period between the completion of a previous read and the start of the next must be at least $\Delta + \delta$ to guarantee that every

process has had a chance to refresh its registry at least once between reads. If a process p detects no change in another process q's counter, p expires q's epoch number and no longer considers q a contender for leadership until a new epoch is detected for q.

The intuition behind the success of the protocol is as follows. First, unless a process always manages to write its registry to f other processes within δ time units after some point, its epoch number will increase indefinitely or will be considered expired (e.g., when it fails).

Second, consider a process p that after a certain time t always manages to write its registry to f other processes within δ. It follows that eventually p stops increasing its epoch number. Note that this is true for any $\diamond f$-accessible process. Let p be the process whose epoch number stops increasing at the lowest value in the system. Denote that lowest epoch number as e_p. The timely refreshing of e_p makes it eventually known as p's epoch by all non-faulty processes. Observe that e_p never expires at any other process, because p succeeds in refreshing e_p's freshness counter every Δ time period. Furthermore, eventually all higher epoch numbers either become known to all non-faulty processes, or belong to processes whose (lower) epoch numbers expire. Hence, eventually all other processes will consider p leader.

The protocol also makes use of monotonically increasing counters, such as *refreshNum* and *readNum*, to associate responses with requests. These counters are initialized to 0. Variables *epochStartTime* and *lastCompletedReadStartTime* are introduced for later use, when the protocol is extended for stability in Section 5.

Process p also has a variable $leader : P \cup \texttt{null}$, that captures p's view of the current leader. *leader* is initially set to `null`. $\Omega_p(t)$ is thus defined to be the value of $leader_p$ on process p at time t. The correctness proof showing that the protocol in Figure 1 implements Ω appears in the full version of this paper [19].

5 Stability

Driven by our need to employ Ω within repeated consensus instances of the Paxos protocol, we now introduce a crucial addition to Ω.

The definition of Ω mandates that eventually a single leader stabilizes and is never replaced. However, it allows many leaders to be replaced many times until that time arrives. This is undesirable in many respects. In Paxos, replacing a leader is a costly operation. The new leader needs to perform an extra round of communication in order to collect information about the latest actions of the previous leader. In many other settings, electing a new leader involves heavy re-configuration procedures, which should be avoided if possible.

We therefore would like to require that a qualified leader (e.g., a $\diamond f$-accessible leader) never be demoted. To this end, we first need to define precisely what it means for a process to be a leader. Our definition is simple and is grounded in practice: A process p is a leader at time t if it considers itself a leader at time t. More precisely, we have the following simple definition:

Start `refreshTimer` with Δ time units; Start `readTimer` with $\Delta + \delta$ time units;

REFRESH:

Upon `refreshTimer` timeout: /* time to refresh the registry */
start `refreshTimer` with Δ time units;
$ackMsgCount := 0$; $refreshNum$ ++;
send $\langle \mathsf{refresh}, p, registry[p], refreshNum \rangle$ to every $q \in P$;
start `roundTripTimer` with δ time units;

Upon receiving $\langle \mathsf{refresh}, q, rg, rn \rangle$:
if $(registry[q] < rg)$ $registry[q] := rg$; send to q $\langle \mathsf{ack}, p, q, rn \rangle$; **end if**

Upon receiving $\langle \mathsf{ack}, q, p, rn = refreshNum \rangle$:
if $(++ackMsgCount \geq f + 1)$
stop `roundTripTimer`; $registry[p].freshness$ ++;
end if

ADVANCE EPOCH:

Upon `roundTripTimer` timeout: /* no timely links to a quorum */
$views[p].expired :=$ `true`; $registry[p].epochNum.serialNum$ ++;
$epochStartTime :=$ `currentTime`;

COLLECT:

Upon `readTimer` timeout: /* time to read the registries */
$lastReadStartTime :=$ `currentTime`; $readNum$ ++;
$statusMsgCount := 0$; $oldViews := views$; /* store for comparison */
send $\langle \mathsf{collect}, p, readNum \rangle$ to every $q \in P$;

Upon receiving $\langle \mathsf{collect}, q, rn \rangle$: send to q $\langle \mathsf{status}, p, q, rn, registry \rangle$;

Upon receiving $\langle \mathsf{status}, q, p, rn = readNum, qReg \rangle$:
for each $r \in P$ $views[r].state :=$ `max`$(qReg[r], views[r].state)$; **end for**
if $(++statusMsgCount \geq n - f)$ /* responses from a quorum collected */
$lastCompletedReadStartTime := lastReadStartTime$;
for every $r \in P$ /* check if r has refreshed its epoch number */
if $(views[r].state \leq oldViews[r].state)$ $views[r].expired :=$ `true`; **end if**
if $(views[r].state.epochNum > oldViews[r].state.epochNum)$
$views[r].expired :=$ `false`;
end if
end for
$leaderEpoch :=$ `min`$(\{views[q].state.epochNum \mid views[q].expired =$ `false`$\}$
$\cup \{\langle 0,$ `null`$\rangle\})$;
$leader := leaderEpoch.processId$; start `readTimer` with $\Delta + \delta$ time units;
end if

Fig. 1. Ω with $\diamond f$-accessibility

Definition 5. *Process p is a leader at time t iff $\Omega_p(t) = p$.*

Intuitively, this definition is desirable because, once p considers itself a leader, it takes actions as leader and may incur any cost mentioned earlier associated with leadership. Leader stability is then defined simply as follows:

Definition 6 (Leader Stability:). *Let p be a leader at time t, and assume that p is f-accessible during the period $[t - \delta, t + \tau]$. We say that a protocol implementing Ω satisfies leader stability at time $t+\tau$ if p is still a leader at time $t + \tau$, and no other process $q \neq p$ is a leader at time $t + \tau$.*

Ω with Stability

In this section, we introduce changes to the above protocol in order to provide for leader stability. In order for these changes to work, however, we require $n = 2f + 1$.[3]

In the protocol of Figure 1, p considers itself a leader immediately when p sets $leader_p$ to p; that is, when p's current epoch number is the lowest non-expired epoch number in p's view. This is insufficient; the scenario that disrupts stability is as follows. Suppose a process p becomes a leader at time t because its current epoch number e_p is the lowest non-expired epoch number in its view at t. In the meantime, another process q times out on an epoch number $e_q < e_p - 1$ and advances to a new epoch number $e_q + 1 < e_p$. If q now becomes f-accessible, $e_q + 1$ will eventually become the lowest epoch number, demoting leader p even if p has been f-accessible; leader stability is thus violated.

To achieve stability, for a process p to become a leader, we not only require that p's epoch number be the lowest non-expired epoch number in p's view, but further require that p declare itself a leader only after making sure that no non-expired lower epoch number will cause other processes to claim leadership. This can be achieved by the following two extensions to the first protocol:

1. Whenever a process initiates a new epoch number, rather than incrementing the epoch number by 1, it learns the highest existing epoch number through a timely communication (with bound δ) with $n - f$ processes and then picks an epoch number that is higher than any existing epoch number.
2. Process p not only checks whether its current epoch number is the lowest in its current view, but also waits for sufficiently long to ensure that all non-expiring epoch numbers that can be lower than e_p must have been reflected in p's view.
   To be precise, let t be the time when the current epoch number e_p is chosen, a process p has to wait until the completion of a collect/status round that starts at least $2\Delta + 3\delta$ time units after time t. This is because a non-faulty and f-accessible p will start its first refresh for e_p at $t + \Delta$ and receive $f + 1$ responses before $t + \Delta + \delta$. In order for another process q to pick an epoch

[3] Alternatively, we could require that an accessible process have timely links to $n - f$ processes, rather than $f + 1$ processes.

Start `refreshTimer` with Δ time units; Start `readTimer` with $\Delta + \delta$ time units;

REFRESH: same as in Figure 1

ADVANCE EPOCH:

Upon initialization or `roundTripTimer` timeout:
/* no timely links to a quorum, retrieving existing epoch numbers */
stop `refreshTimer`;
refreshNum ++; *isLeader* := `false`; *views*[*p*].*expired* := `true`;
epochCount := 0;
globalMaxEn := *registry*[*p*].*epochNum*;
seqNum ++;
send $\langle \mathsf{getEpochNum}, p, seqNum \rangle$ to each process $q \in P$;
start `getEpochTimer` with δ time units;

Upon `getEpochTimer` timeout: /* no timely links to a quorum, retry */
seqNum ++;
epochCount := 0;
globalMaxEn := *registry*[*p*].*epochNum*;
send $\langle \mathsf{getEpochNum}, p, seqNum \rangle$ to each process $q \in P$;
start `getEpochTimer` with δ time units;

Upon receiving $\langle \mathsf{getEpochNum}, q, sn \rangle$:
localMaxEn :=max$\{$*registry*[*r*].*epochNum* $\mid r \in P\}$;
send to q $\langle \mathsf{retEpochNum}, p, q, sn, localMaxEn \rangle$;

Upon receiving $\langle \mathsf{retEpochNum}, q, p, sn = seqNum, en \rangle$:
if ($en >$ *globalMaxEn*) *globalMaxEn* := *en*; **end if**
if (++*epochCount* $\geq n - f$) /* epoch numbers from a quorum collected */
registry[*p*].*serialNum* := *globalMaxEn*.*serialNum* + 1;
epochStartTime := `currentTime`;
start `refreshTimer` with Δ time units;
end if

COLLECT: same as in Figure 1

BECOME LEADER:

Upon change to *lastCompletedReadStartTime*
if (*leaderEpoch* = *registry*[*p*].*epochNum* $\wedge$
lastCompletedReadStartTime $-$ *epochStartTime* $\geq 2\Delta + 3\delta$)
isLeader := `true`;
end if

Fig. 2. Stable Leader Election Protocol with $\diamond f$-accessibility

number e_q lower than e_p, q must have started the (timely) communication to learn existing epoch numbers before $t + \Delta + \delta$ and then started epoch e_q at $t + \Delta + 2\delta$; otherwise, due to $n - f + f + 1 > n$, one of the $n - f$ processes reporting existing epoch numbers will be among the $t + 1$ that know e_p and will report an epoch number that is e_p or higher. If q never expires e_q, then it will complete its refresh for e_q at $t + 2\Delta + 3\delta$. Any collect/status round after $t + 2\Delta + 3\delta$ will reflect e_q; therefore, e_p is not the lowest non-expired epoch and p will not become a leader.

To capture the condition under which a process considers itself a leader, we introduce, in addition to variable $leader_p$, a boolean local variable $isLeader_p$ for each process p and define Ω_p as follows:

$$\Omega_p := \begin{cases} p & isLeader_p = \texttt{true} \\ leader_p & leader_p \neq p \wedge leader_p \neq \texttt{null} \\ \texttt{null} & \text{otherwise} \end{cases}$$

The full protocol is given in Figure 2. The correctness proofs appear in the full version of this paper [19].

6 Reducing Message Complexity

As suggested in [9], a crucial measure of communication complexity is the number of links that are utilized infinitely often in the protocol. Our protocols use all-to-all communication infinitely often to keep leader information up to date, hence employ $O(n^2)$ infinite-utilization links.

For the protocol in Figure 1, the steady-state communication complexity can be reduced to $O(n)$, where in a steady state there exists a unique f-accessible leader that is never suspected by any non-faulty process. We briefly sketch the required changes here. The full paper [19] contains a precise protocol description and its correctness proof.

The first change is related to the refreshing of epoch numbers. A process p that is not currently the leader need not refresh its own epoch number; it can simply let it become *inactive*, since it is not contending for the leadership. Therefore, we disable the periodic refresh at p when it is not a leader. A process increments its epoch number only when it experiences a `roundTripTimer` timeout, as in the original protocol, and may "revive" an inactive epoch number when becoming a leader.

The second change is related to the monitoring of epoch numbers in the system. In a steady state, there is no reason for a process p to monitor the states of all other processes. Therefore, we disable periodic collect altogether.

A process p that does not obtain any refresh message carrying the current presumed leader's epoch number for some timeout period suspects that the current leader has failed. Likewise, a process p that hears a refresh message carrying

a lower epoch number than the current presumed leader's epoch number assumes that it does not have up-to-date information about the current leader .

In these two cases (only), a process activates the collect procedure twice, where the second one is activated at least $\Delta + \delta$ time units after the first one completes, as in the original protocol. Process p then determines the lowest active epoch number and compares it with its current epoch number. If p's current epoch number is no higher than the lowest active epoch number, p becomes a leader and activates refresh periodically as in the original protocol. Otherwise, p will consider the process owning the lowest epoch number as the leader and expect to receive refresh messages from that process periodically.

The intuition behind the success of the modified protocol is somewhat similar to our original protocol, but with crucial differences. As before, consider a process p that, after a certain time t, always manages to write its registry to f other processes within δ. It follows that eventually p stops increasing its epoch number. Note that this is true for any $\diamond f$-accessible process.

Now, consider a non-crashed process q with the lowest current epoch number in the system. If q is not the leader yet, then q believes that there exists a lower active epoch number than its own. Because such an epoch number no longer exists, eventually q times out on that epoch number and performs two collects. Because its epoch number is the lowest among the non-crashed processes, it will learn that its epoch number is no higher than the lowest active epoch number in the system and become a leader. If q is not $\diamond f$-accessible, eventually it will fail updating its own freshness counter and will increase its epoch number.

Together, we have that, on the one hand, the $\diamond f$-accessible processes stop increasing their epoch numbers. On the other hand, any non $\diamond f$-accessible process either crashes or increases its own epoch number to be higher than the lowest epoch number in the system. As before, the process p whose epoch stops increasing at the lowest value in the system becomes a permanent leader.

In terms of message complexity, once an f-accessible leader is elected and all processes receive its refresh messages without suspecting the leader, eventually all non-leader processes stop refreshing their epochs and stop reading, hence the communication complexity drops to $O(n)$.

7 Discussion

The condition we introduced to uphold stability in this paper, namely $n = 2f+1$, is stronger than what is required in practice. It is worth noting that, for both Paxos and our stable leader election protocol, it suffices for a leader p to interact in a timely fashion *once* with $n - f$ processes. Subsequently, p can maintain its leadership and proceed with consensus decisions, provided that it can interact at any time with $f + 1$ processes.

Stability also appears to be in conflict with the ability to reduce the steady-state message complexity to $O(n)$. Intuitively, the reduced message complexity forces a process to decide whether to become a leader based on less accurate information, thereby creating opportunities for unnecessary demotion. For example,

in our protocol, to ensure stability, a process becomes a leader only when it is certain that no process can have a lower active epoch number. This is hard because epoch numbers can remain inactive (and unknown to other processes) before they are revived. It is left as an open question whether a stable leader protocol exists under $\diamond f$-accessibility with $O(n)$ steady-state message complexity.

8 Conclusion

It is our firm belief that leader election algorithms that implement Ω should be studied in the context of practical coordination schemes that realize consensus. This paper makes two contributions toward this goal.

First, it contributes to the study of weak synchrony conditions that enable leader election. $\diamond f$-accessibility, the synchrony condition we require, is new and surprisingly weak, in that it requires no eventual timely links. It is incomparable to (but also not stronger than) previously known conditions for leader election. The condition is derived by our observations on Paxos, leading to an implementation of Ω under f-accessibility.

Second, it provides practical and stable leader election protocol that eliminates unnecessary and potentially expensive leader changes. The paper therefore provides Paxos with a "good" leader election protocol; this was left as an open problem in Lamport's original Paxos paper [1].

References

1. Lamport, L.: The part-time parliament. ACM Transactions on Computer Systems **16** (1998) 133–169
2. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. Journal of the ACM **32** (1985) 374–382
3. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM **43** (1996) 225–267
4. Chandra, T.D., Hadzilacos, V., Toueg, S.: The weakest failure detector for solving consensus. Journal of the ACM **43** (1996) 685–722
5. Lee, E.K., Thekkath, C.: Petal: Distributed virtual disks. In: Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 1996). (1996) 84–92
6. Thekkath, C., Mann, T., Lee, E.K.: Frangipani: A scalable distributed file system. In: proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP 1997). (1997) 224–237
7. van Renesse, R., Schneider, F.B.: Chain replication for supporting high throughput and availability. In: Proceedings of the 6th Usenix Symposium on Operating System Design and Implementation (OSDI 2004). (2004) 91–104
8. MacCormick, J., Murphy, N., Najork, M., Thekkath, C.A., Zhou, L.: Boxwood: Abstractions as the foundation for storage infrastructure. In: Proceedings of the 6th Usenix Symposium on Operating System Design and Implementation (OSDI 2004). (2004) 105–120

9. Larrea, M., Fernndez, A., Arvalo, S.: Optimal implementation of the weakest failure detector for solving consensus. In: Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS 2000). (2000) 52–59
10. Prisco, R.D., Lampson, B., Lynch, N.: Revisiting the Paxos algorithm. In: Proceedings of the 11th Workshop on Distributed Algorithms(WDAG). (1997) 11–125
11. Larrea, M., Arvalo, S., Fernndez, A.: Efficient algorithms to implement unreliable failure detectors in partially synchronous systems. In: Proceedings of the 13th International Symposium on Distributed Computing (DISC 1999). (1999) 34–48
12. Aguilera, M., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: Stable leader election. In: Proceedings of the 15th International Symposium on Distributed Computing (DISC 2001). (2001)
13. Aguilera, M.K., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: On implementing Omega with weak reliability and synchrony assumptions. In: Proceedings of the Twenty-Second Annual ACM Symposium on Principles of Distributed Computing (PODC 2003), ACM Press (2003) 306–314
14. Anceaume, E., Fernndez, A., Mostefaoui, A., Neiger, G., Raynal, M.: A necessary and sufficient condition for transforming limited accuracy failure detectors. J. Comput. Syst. Sci. **68** (2004) 123–133
15. Aguilera, M.K., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: Communication-efficient leader election and consensus with limited link synchrony. In: Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing (PODC 2004), ACM Press (2004) 328–337
16. Yang, J., Neiger, G., Gafni, E.: Structured derivations of consensus algorithms for failure detectors. In: Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing (PODC 1998). (1998) 297–308
17. Mostefaoui, A., Raynal, M.: Unreliable failure detectors with limited scope accuracy and an application to consensus. In: Proceedings of the 19th International Conference on Foundations of Software Technology and Theoretical Computer Science (FST&TCS99), Springer-Verlag LNCS #1738 (1999) 329–340
18. Chu, F.: Reducing Ω to $\diamond W$. Information Processing Letters **67** (1998) 298–293
19. Malkhi, D., Oprea, F., Zhou, L.: Omega meets Paxos: Leader election and stability without eventual timely links. Technical Report MSR-TR-2005-93, Microsoft Research, Redmond, WA (2005)