Ω Meets Paxos: Leader Election and Stability without Eventual Timely Links*

Dahlia Malkhi
Microsoft Research Silicon Valley and the Hebrew University of Jerusalem
Florin Oprea†
Department of Electrical and Computer Engineering, Carnegie Mellon University
Lidong Zhou
Microsoft Research Silicon Valley

July 2005

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.

*This paper is an extended version of our preliminary conference paper [15]. The extensions include a new protocol with reduced message complexity, and correctness proofs of all protocols.

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

1 Introduction

A fundamental design guideline pioneered in the Paxos protocol [10] 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 [8]. Ω is known to be the weakest failure detector [6, 5] that is sufficient for consensus, hence provides the liveness properties of consensus. Ω 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 [13], Frangipani [20], Chain Replication [21], and Boxwood [14].

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]:

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

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

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 [8] 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. The reason is that 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 to capture the requirement 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.

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 [10, 17]. 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, 12] 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, 12, 1, 2, 9, 4, 3]. 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 [1] and to one that has a process whose outgoing links to the rest of the system are eventually timely [2]. In [2], 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 [9, 22, 16] introduces the notion of *Gamma-accurate failure detectors* or *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 $\diamond S$ [4]; finally transform $\diamond S$ to Ω [7].

Aguilera et al. [3] 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. [3] 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. Time-free variants of the $\diamond f$-source condition are presented in [18, 19]. Instead of a timing requirement, their approach requires the source process p to be among the first $n - f$ to respond to any message by the f other processes.

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 [4, 3] 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. [1]. 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 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 $\delta/2$ on the delay of messages, but it does not hold on all channels at all times. What is known is that eventually there is one process that is able to exchange messages within the $\delta/2$ delay with f other processes. We will now make this notion precise.

Definition 3.1 *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 a correct process p to a correct process q at time t arrives within $\delta/2$ time. Note that if q becomes faulty before accepting p's message then by definition the link is not timely.*

Definition 3.2 *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 during $[t, t+\delta/2]$. Note that this not only means that messages sent by p are received by q, but also that if q immediately responds to p then the response arrives by time $t+\delta$.*

Our synchrony requirement is the following.

Definition 3.3 *($\diamond f$-accessibility) There is a time t and a process p such that for all $t' \geq t$, p is f-accessible during 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 [10] and revolving-coordinator consensus [6] 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 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 [3], 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 [6]: Ω provides every process q at any time t with a local hint $\Omega_q(t)$, such that the following holds:

Definition 3.4 (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 $\Omega_q(t') = p$.*

[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 read quorums; operations on $(n-f)$-subsets are replaced with operations on write quorums.

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 6.

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.

4.1 Proof of Correctness

In this section, we show that the protocol in Figure 1 implements Ω. For convenience, we use v_p to denote variable v on process p. We also introduce $\Box_t$ and $\diamond\Box$, where $\Box_t(\texttt{prop})$ holds iff property `prop` holds at any time $t' \geq t$, and $\diamond\Box(\texttt{prop})$ holds iff there exists a time t such that $\Box_t(\texttt{prop})$ holds.

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

Figure 1: Ω with $\diamond f$-accessibility.

Lemma 4.1 *For a non-faulty process p, if p is non-faulty and incurs a finite number of* `roundTripTimer` *timeouts, then there exists an epoch number e_p, such that for any non-faulty process q the following holds:*

$$\Diamond\Box(views_q[p].state.epochNum = e_p \wedge views_q[p].state.expired = \texttt{false})$$

Proof. Because p experiences a finite number of `roundTripTimer` timeouts, there exists time t, after which p experiences no `roundTripTimer` timeouts. Let e_p be the highest value for $registry_p[p].epochNum$. Such e_p exists because $registry_p[p].epochNum$ is advanced only upon a `roundTripTimer` timeout and because `roundTripTimer` timeout will never be triggered after t.

First, we show that, for any non-faulty process q, the following holds:

$$\Diamond\Box(views_q[p].state.epochNum = e_p).$$

Consider the time period after t. Because `roundTripTimer` timeout will never be triggered on p, process p is guaranteed to receive $f+1$ ack messages for every refresh message (containing $registry[p].epochNum = e_p$) it sends. Let t' be the time when process p receives $f+1$ ack messages to its first refresh message. For any non-faulty process q, because there could be at most f faulty processes and links between non-faulty processes are reliable, q is guaranteed to receive $n-f$ status messages for every collect message it sends and then initiate a new collect message at a later time. Therefore, there must exist a collect message q sent after t'.

For a collect message q sent after t', because $(n-f)+(f+1) > n$ holds, there exists a process r that is among the $f+1$ processes that received p's refresh message with $rg.epochNum = e_p$ and also among the $n-f$ processes from which q received status messages. Because e_p is the highest epoch number p ever uses, the message from process r will have $registry_r[p].epochNum = e_p$. Therefore, process q will set $views_q[p].state.epochNum$ to e_p. Because p never sends a refresh message with an epoch number higher than e_p and $views_q[p].state.epochNum$ never decreases, $views_q[p].state.epochNum$ will remain e_p afterwards.

Second, for any non-faulty process q, let t_1 be the time when $views_q[p].state.epochNum$ is set to e_p. We show that the following holds:

$$\Box_{t_1}(views_q[p].state.expired = \texttt{false}).$$

Note that, to set $views_q[p].state.expired$ to `true`, there must exist a round of collect/status completed after t_1 such that $views_q[p].state.epochNum = e_p$, $oldViews_q[p].state.epochNum = e_p$, $views_q[p].state.freshness = oldViews_q[p].state.freshness$ hold. We prove that this cannot happen after time t_1.

Consider any time where $views_q[p].state.epochNum = e_p$ holds on q and let cnt be $views_q[p].state.freshness$. This is the result of the previous round of collect/status. Let t_2 be the time when the previous round completes (i.e., when q receives $n-f$ collect messages.) This indicates that a refresh message with $rg = \langle e_p, cnt\rangle$ has been sent before time t_2. Because `refreshTime` is set to Δ and process p experiences no `roundTripTimer` timeout (otherwise, p would have advanced its epoch number), p will send another refresh message with $\langle e_p, cnt+1\rangle$ before $t_2+\Delta$. Again, because p experiences no `roundTripTimer` timeouts, $f+1$ processes will receive that message before $t_2+\Delta+\delta$.

Process q initiates a new collect message $\Delta+\delta$ time units after the previous collect/status round ends. The new connect message is thus sent at or after $t_2+\Delta+\delta$, which is after $f+1$ processes have received the refresh message from p with $\langle e_p, cnt+1\rangle$. Again, due to $(n-f)+(f+1) > n$, process q must have received a message with $qregistry[p]$ that is equal to or greater than $\langle e_p, cnt+1\rangle$, which is greater than $\langle e_p, cnt\rangle$. Therefore, for every non-faulty process q, $views[p].state.expired$ will never be set to `true` when $views[p].state.epochNum$ is e_p. □

Lemma 4.2 *If process p fails or p incurs* `roundTripTimer` *timeouts infinitely often, then for any epoch number e_p, where $e_p.processId = p$, the following holds on any non-faulty process q:*

$$\Diamond\Box(views_q[p].state.epochNum > e_p) \vee \Diamond\Box(views_q[p].state.expired = \texttt{true})$$

Proof. In both cases where p fails or incurs infinitely many `roundTripTimer` timeouts, for every epoch number e_p used by p, there are finitely *state.freshness* values used by p with $state.epochNum = e_p$. For any other process q, $views_q[p]$.
state may only contain epoch values and counter values used by p. Therefore, there are finitely many count values in $views_q[p].state$ as well. Since q executes infinitely many collect loops, eventually every epoch value in $e_p = views_q[p].state.epochNum$ expires and, if p continues to send new refresh messages, is replaced by a higher one (due to reliable links). Because there are only a finite number of epoch numbers that are lower than e_p and $views_q[p].state$.
epochNum never decreases, eventually either $views_q[p].expired$ remains `true` or $views_q[p].state.epochNum > e_p$ holds forever. □

Theorem 4.3 (Ω) *If a process p is $\diamond f$-accessible, then there exists a non-faulty process q, such that for any non-faulty process r, $\diamond\Box(\Omega_r = q)$ holds.*

Proof. Let Q be the set of processes whose `roundTripTimer`'s expire a bounded number of times throughout the execution. Because an f-accessible experiences no `roundTripTimer` timeout and p is $\diamond f$-accessible, Q contains p, hence it is not empty. Furthermore, by Lemma 4.1, for every $q \in Q$, there exists an epoch number e_q, such that for every non-faulty process r, $\diamond\Box views_r[q].state.epochNum = e_q$ and $\diamond\Box views_r[p].state.expired =$ `false`. Let e_q be the lowest such epoch number.

To the contrary, by definition of Q, every process $s \notin Q$ either fails, or incurs an infinite number of `roundTripTimer` timeouts in the execution. Therefore, for any epoch number $e_s < e_q$, with $e_s.processId = s$, due to Lemma 4.2, the following holds for any non-faulty process r:

$$\diamond\Box(views_r[s].state.epochNum > e_s) \vee \diamond\Box(views_r[s].state.expired = \texttt{true})$$

Putting the above together, eventually $views_r[q]$ will have e_q as the lowest non-expired epoch number. Consequently, $leaderEpoch_r$ will be set to e_q (and remain e_q) with $leader_r$ set to q. For the protocol in Figure 1, this implies that $\Omega_r = q$ eventually holds forever and that the protocol implements Ω.

□

5 Reducing Message Complexity

As suggested in [12], a crucial measure of communication complexity is the number of links that are utilized infinitely often in the protocol. The above protocol uses all-to-all communication infinitely often to keep leader information up to date, hence employs $O(n^2)$ infinite-utilization links.

The communication complexity in a *steady* state 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 first provide a high level description of the required changes.

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 its current epoch number 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 epoch number of the presumed leader for some timeout period suspects that the current leader has failed. A leader process p that hears a refresh message carrying a lower epoch number than itself assumes that it does not have up-to-date information about the system, and suspends its own refreshing. It thereby causes any process that considered p leader, including p itself, to time out. Similarly, if a leader p experiences a `roundTripTimer` timeout, it will

also suspends its refreshing and increase its epoch number on the premise that there might exist a process with a lower epoch number and that process should become the leader.

In these three cases (only), a process activates the collect procedure, as in the original protocol. Process p then determines the lowest active epoch number and checks whether its current epoch number is lower. If p's current epoch number is lower than the lowest active epoch number, p enters a leader state and p 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 periodically.

The intuition behind the success of the modified protocol is somewhat similar to our initial 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 perform collect. Because its epoch number is the lowest among the non-crashed processes, it will learn that its epoch number is lower 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 number 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 other processes stop refreshing their epoch numbers and stop invoking collect, hence the communication complexity drops to $O(n)$.

Figure 2 shows the modified protocol with reduced steady-state message complexity. Note that, because there is no notion of epoch expiration in the new protocol, each process maintains a vector *states* that records the *registry* information collected from other processes and stores the old states in vector *oldStates*. We further introduce `leaderTime` for triggering the collect/status procedure and use the *inCollect* flag to prevent the invocation of a new collect/status procedure before $\Delta + \delta$ time units have passed since the end of the previous invocation.

5.1 Proof of Correctness

This section shows that the protocol in Figure 2 implements Ω and achieves $O(n)$ steady-state message complexity.

Lemma 5.1 *If a process p incurs a finite number of* `roundTripTimer` *timeouts, then its epoch number is bounded: there exists an epoch number e_p, such that the following holds:*

$$\diamond\Box(registry_p[p].epochNum \leq e_p)$$

Proof. Because p experiences a finite number of `roundTripTimer` timeouts, there exists a time t, after which p experiences no `roundTripTimer` timeouts. Let e_p be the value of $registry_p[p].epochNum$ at time t. Because
$registry_p[p].epochNum$ is advanced only upon a `roundTripTimer` timeout, it remains e_p. □

Lemma 5.2 *There exists a time t and an epoch number e_p for a non-faulty process p, such that $\Box_t(\texttt{min}\{registry_r[r].epochNum \mid r \in P \setminus F\} = e_p)$ holds, where F is the set of faulty processes.*

Proof. By definition, any non-faulty $\diamond f$-accessible process r incurs a finite number of `roundTripTimer` timeouts. By Lemma 5.1, $registry_r[r].epochNum$ is bounded. Among the non-faulty processes whose epoch numbers are bounded, let p be the process with the lowest bound e_p. Therefore, there exists a time t, at which all non-faulty processes have epoch numbers at least e_p. □

Start `leaderTimer` with $\Delta + \delta$ time units;

/* Code performed only when $leader = p$ ***/**
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 $q \in (P \setminus \{p\})$;
start `roundTripTimer` with δ time units;

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

ADVANCE EPOCH:
Upon `roundTripTimer` timeout: /* no timely links to a quorum */
$registry[p].epochNum.serialNum$ ++; stop `refreshTimer`;
if $inCollect =$ `false` start `leaderTimer` with $\Delta + \delta$ time units; **end if**

/* Code performed by all processes, including the leader */
REFRESH:
Upon receiving $\langle \mathsf{refresh}, q, rq, rn \rangle$:
send to q $\langle \mathsf{ack}, p, q, rn \rangle$; $registry[q] :=$ `max`$(registry[q], rq)$;
if $(rq.epochNum = leaderEpoch)$ reset `leaderTimer` with $\Delta + \delta$ time units; **end if**
if $(rq.epochNum < leaderEpoch \wedge leader = p)$
stop `refreshTimer`; /* yields to the lower epoch number */
if $inCollect =$ `false` start `leaderTimer` with $\Delta + \delta$ time units; **end if**
end if

COLLECT: /* not receiving beacons from the current leader */
Upon `leaderTimer` timeout:
$inCollect :=$ `true`; $statusMsgCount := 0$; $readNum$ ++;
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 $\langle \mathsf{status}, q, p, rn, qReg \rangle$:
for each $r \in P$ $states[r] :=$ `max`$(qReg[r], states[r])$; **end for**
if $(rn = readNum)$ $statusMsgCount$ ++; /* otherwise, old response */
if $(statusMsgCount \geq n - f)$ /* responses from a quorum collected */
$leaderEpoch :=$ `min`$(\{states[q].epochNum \mid q \in P \setminus \{p\} \wedge (states[q] > oldState[q])\}$
$\cup \{registry[p].epochNum\})$; $leader := leaderEpoch.processId$;
if $(leader = p)$ start `refreshTimer` with 0 time units; /* become leader */
else start `leaderTimer` with $\Delta + \delta$ time units;
end if
$inCollect :=$ **false**;
for each $r \in P$ $oldStates[r] :=$ `max`$(registry[r], states[r])$; **end for**
end if

Figure 2: Protocol with Reduced Steady-State Message Complexity.

Lemma 5.3 *Let e_p denote at any time $\texttt{min}\{registry_r[r].epochNum \mid r \in P \backslash F\}$. If at time t, $leaderEpoch_r < e_p$ for some process r, then there exists a time $t' \geq t$ such that r performs* collect/status *at t'.*

Proof. Denote by $e_q = leaderEpoch_r$ at time t. According to the definition of e_p, for any epoch number $e_q < e_p$, only finitely many refresh messages with e_q are ever sent. This is because the sender q will either fail or advance to a higher epoch number. There exists a time $t_1 \geq t$, after which no more refresh messages with epoch e_q are delivered by r. If at t_1 it still holds that $e_q = leaderEpoch_r$, then there exists a time $t' \geq t_1$ at which the `leaderTimer` times out on r, activating an execution of collect/status. Otherwise, at t_1 already $leaderEpoch_r \neq e_q$. Since the only place in the algorithm that may change $leaderEpoch_r$ is inside collect/status, it must have been activated at some time t' between t and t_1. □

Lemma 5.4 *Let e_q be an epoch number such that, at any time, the following holds:*

$$e_q < \texttt{min}\{registry_r[r].epochNum \mid r \in P \setminus F\}$$

Then if process p performs collect/statuss *indefinitely, the following holds:*

$$\Diamond\Box(states_p[q] \geq \max\{states_r[q] \mid states_r[q].epochNum = e_q \wedge r \in P \setminus F\}).$$

Proof. By assumption, e_q is an epoch number of q for which a finite number of refresh messages were sent. Let rq be the highest state with $rq.epochNum = e_q$ ever received in a refresh message or a status message by any non-faulty process. Let t_1 be a time such that some non-faulty process r receives rq. Since p performs collect/statuss indefinitely, there is a collect/status performed by p at time $t_2 \geq t_1$. The lemma follows from the fact that the status response from r to this collect request is incorporated into $states_p$ whenever it arrives. □

Lemma 5.5 *For any time t, let $e_p = \texttt{min}\{registry_r[r].epochNum \mid r \in P \setminus F\}$, there exists a time $t' \geq t$ such that, for every non-faulty process r, the following holds.*

$$\Box_{t'}(leaderEpoch_r \geq e_p)$$

Proof. Proof by contradiction. Assume that there exists a non-faulty process r, such that, for any time t', there exists a time $t'' \geq t'$ such that $leaderEpoch_r < e_p$ holds at time t''.

Then by Lemma 5.3, r performs a collect/statuss indefinitely. According to Lemma 5.4, for any $e_q < e_p$, there is a time t_1 such that $\Box_{t_1}(states_p[q] \geq \max\{states_r[q] \mid states_r[q].epochNum = e_q \wedge r \in P \setminus F\})$. Let t_2 be the maximum time t_1 taken over the above condition for all $e_q < e_p$.

After time t_2, there will never be a status or refresh message containing a state st for any process q, where the following holds:

$$(st[q].epochNum < e_p) \wedge (st[q].epochNum > states_r[q].epochNum) \tag{1}$$

From this, since r sets $oldStates$ to $states_r$ at the end of a collect/status procedure, it follows immediately that the next execution of the collect/status procedure at r, and any subsequent instances there of, will set $leaderEpoch_r$ to an epoch number at least e_p.

According to the above, there is a time $t_3 \geq t_2$ at which r performs a collect/status procedure. Let $t_4 \geq t_3$ be the time that r completes collecting the responses for this collect/status. Then r sets $leaderEpoch_r \geq e_p$ at time t_4. Furthermore, because $leaderEpoch_r$ is modified only at the end of the collect/status procedure, $leaderEpoch_r \geq e_p$ will continue to hold after the the first collect/status procedure initiated after t_4 sets $leaderEpoch_r$. We have that $\Box_{t_4} leaderEpoch_r \geq e_p$. Contradiction. □

Theorem 5.6 **(Ω)** *If a process r is $\Diamond f$-accessible, then there exists a non-faulty process p, such that for any non-faulty process q, $\Diamond\Box(\Omega_q = p)$ holds.*

Proof. Due to Lemma 5.2 and Lemma 5.5, there exists an epoch number e_p for a non-faulty process p and a time t, such that after time t,

1. Process p experiences no `refreshTimer` timeout and $states_p[p].epochNum$ remains e_p.

2. For any non-faulty process r, $registry_r[r] \geq e_p$ always holds.

3. For any non-faulty process r, $leaderEpoch_r \geq e_p$ always holds.

We first show that process p eventually starts sending refresh messages with e_p at the interval of Δ time units.

Due to Condition (1) and the fact that a process p never sets its $leaderEpoch_p$ higher than $registry_p[p]$, $leaderEpoch_p \leq e_p$ holds. Due to Condition (3), $leaderEpoch_p$ is at least e_p. So, $leaderEpoch_p = e_p$ holds after t.

According to the protocol, process p will start refreshing e_p after it sets $leaderEpoch_p$ to e_p. Because p experiences no `roundTripTimer` timeout, the only time p stops refreshing e_p is when it receives a refresh message with a lower epoch number. But the subsequently started collect/status procedure will again set $leaderEpoch_p$ to e_p and start refreshing. After the finitely many refresh messages with a lower epoch number than e_p are delivered on p, p will keep refreshing e_p at the interval of Δ. Let t' be the time when p starts refreshing e_p continuously.

We then show that any process $q \neq p$ eventually stops sending refresh messages. Assume otherwise. Consider a process q that sends refresh messages for an epoch number $e_q > e_p$ (due to Condition 2). Process q eventually receives a refresh message with e_p after time t'' , thereby initiating a collect/status procedure. Therefore, it will set its $leaderEpoch$ to e_p. Because $e_p < e_q$ holds, q will never start the `refreshTimer`.

After time t', if at any time $t'' \geq t'$ a process r has $leaderEpoch_r < e_p$, then by Lemma 5.3, a collect/status procedure will be triggered; and if $leaderEpoch_r = e_q > e_p$, then because we have just shown that q stops sending refresh messages for e_q, r will incur a `leaderTimer` timeout and a collect/status round will be triggered as well. In either case, during such a collect/status, we already have from Condition (3) that $leaderEpoch_r$ will be set to at least e_p. Now we show that $leaderEpoch_r$ is set to a value $\leq e_p$.

Because a new collect/status procedure is started at least $\Delta + \delta$ time units after the previous one ends, $f+1$ processes (including p) must have received a new refresh message with a higher *freshness* value after the previous collect/status ends, but before the new collect/status starts. Due to quorum intersection, the $n - f$ status messages that q receives must contain the new *freshness* value, which is higher than the *freshness* value for p in $oldState$. Therefore, $e_p \in \{states[q].epochNum \mid (q \in P \setminus \{p\}) \wedge (states[q] > oldState[q])\}$ holds. Consequently, $leaderEpoch_r \leq e_p$ holds.

□

Theorem 5.7 (Message Complexity) *Eventually, only one non-faulty process sends* refresh *messages to all processes at an interval of δ time units. If that process has eventually timely links with all non-faulty processes, eventually no* collect *messages will be sent.*

Proof. Because of Ω, as proven in Theorem 5.6, every non-faulty process $r \neq p$ will have $leaderEpoch_r$ set to e_p. Due to $e_p.processId \neq r$, process r must have stopped its `refreshTimer` and will never restart it.

Eventually, no process will start execution of the collect/status procedure because none of the three possible conditions for starting such an instance hold.

1. For any non-faulty process r, eventually, it will stop receiving any refresh message containing an epoch number lower than e_p.

2. Because p eventually sends refresh messages with e_p to all processes every Δ time units, r will not incur any `leaderTimer` timeout.

3. Eventually, no non-faulty process experiences `roundTripTimer` timeout, because p never does and eventually all other non-faulty processes stop their `refreshTimer`.

□

6 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:

Definition 6.1 *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 6.2 (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

[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

Figure 3: Stable Leader Election Protocol with $\Diamond f$-accessibility.

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 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 3.

6.1 Proof of Correctness

This section provides a proof that the protocol in Figure 3 achieves Ω as well as Leader Stability.

We first observe that Lemma 4.1 and Lemma 4.2 from Section 4.1 hold in Figure 3 precisely the same way as in Figure 1, and the proofs are identical.

Lemma 6.3 *For the protocol in Figure 3, if a process p sets $isLeader_p$ to* `true` *at time t, then for any process $q \neq p$ the following holds:*

$$\Box_t(isLeader_q = \texttt{false} \vee leaderEpoch_q > e_p).$$

Proof. The proof is by contradiction. Let e_p be the value of $leaderEpoch_p$ at time t. Assume there exists a process $q \neq p$ and an epoch number $e_q < e_p$, such that $isLeader_q = \texttt{true}$ and $leaderEpoch_q = e_q$ at time $t' \geq t$.

Because process p sets *isLeader* to `true` with $leaderEpoch = e_p$ at time t, according to the protocol, the following holds:

$$lastCompletedReadStartTime - epochStartTime \geq 2\Delta + 3\delta \tag{2}$$

Let t' be $epochStartTime_p$ for epoch e_p (i.e., when $registry_p[p].epochNum$ is set to e_p). At time $t'+\Delta$, p initiates a refresh message with $registry[p].epochNum = e_p$. Because p has no `roundTripTimer` timeouts on e_p up to time t (otherwise, p would not set $isLeader_q$ `true` with $leaderEpoch_q = e_q$), at time $t'+\Delta+\delta$, $f+1$ processes must have received the refresh message with $registry[p].epochNum = e_p$. Therefore, $t'+\Delta+\delta$ is the latest time that process q can initiate *getEpochNum*, while still choosing its local epoch number to be e_q (lower than e_p.) This is because, if a *getEpochNum* message is sent after $t'+\Delta+\delta$, one of the corresponding *retEpochNum* messages will contain *localMaxEn* that is at least e_p (due to $f+1+(n-f)>n$).

So, the latest time a process q sets its $registry[q].epochNum$ to e_q ($< e_p$) is $(t'+\Delta+\delta)+\delta$. Because q is non-faulty and never gets a `roundTripTimer` timeout on e_q up to time $t' > t$ (otherwise, q will not set $isLeader_q$ `true` with $leaderEpoch_q = e_q$) then at time $t'+\Delta+2\delta+(\Delta+\delta)$, $f+1$ processes must have received a refresh message from q with $rg.epochNum = e_q$.

Condition (2) implies that the last completed collect/status round performed by p must be initiated after $f+1$ processes have received a refresh message from q with $rg.epochNum = e_q$. Therefore, we have that $views_p[q].state.epochNum \geq e_q$. Because process q never experiences a `roundTripTimer` timeout on e_q up to time t, $views_p[q].state.epochNum = e_q$ and $views[q].expired = \texttt{false}$ hold at time t. Therefore, $leaderEpoch_p \leq e_q < e_p$ holds on p at time t. In this case, p would not have set $leader_p$ to p and $leaderEpoch_p$ to e_p because $e_q < e_p$ holds. Contradiction. □

Theorem 6.4 (Ω) *If a process p is $\diamond f$-accessible, then there exists a non-faulty process q, such that for any non-faulty process r, $\diamond\Box(\Omega_r = q)$ holds.*

Proof. Let Q be the set of processes whose `roundTripTimer`'s expire a bounded number of times throughout the execution. Because an f-accessible experiences no `roundTripTimer` timeout and p is $\diamond f$-accessible, Q contains p, hence it is not empty. Furthermore, by Lemma 4.1, for every $q \in Q$, there exists an epoch number e_q, such that for every non-faulty process r, $\diamond\Box views_r[q].state.epochNum = e_q$ and $\diamond\Box views_r[p].state.expired = \texttt{false}$. Let e_q be the lowest such epoch number.

To the contrary, by definition of Q, every process $s \notin Q$ either fails, or incurs an infinite number of `roundTripTimer` timeouts in the execution. Therefore, for any epoch number $e_s < e_q$, with $e_s.processId = s$, due to Lemma 4.2, the following holds for any non-faulty process r:

$$\diamond\Box(views_r[s].state.epochNum > e_s) \vee \diamond\Box(views_r[s].state.expired = \texttt{true})$$

Putting the above together, eventually $views_r[q]$ will have e_q as the lowest non-expired epoch number. Consequently, $leaderEpoch_r$ will be set to e_q (and remain e_q) with $leader_r$ set to q. For the protocol in Figure 1, this implies that $\Omega_r = q$ eventually holds forever and that the protocol implements Ω.

To prove that the protocol in Figure 3 implements Ω, it remains to show that $isLeader_q$ will be set to `true` and that, for all $r \neq q$, $\diamond\Box(isLeader_r = \texttt{false})$.

We first show that eventually q sets *isLeader* to `true`. It suffices to show that the following eventually holds on q.

$$lastCompletedReadStartTime - epochStartTime \geq 2\Delta + 3\delta.$$

This is obvious: *epochStartTime* never changes after a time t because q has no `roundTripTimer` timeouts on e_p; while *lastCompletedReadStartTime* increases by at least $\Delta + \delta$ after each collect/status round and a new round will always be initiated. Therefore, we have

$$\diamond\Box(isLeader_q = \texttt{true} \wedge leaderEpoch_q = e_q) \tag{3}$$

Now we show that, for all $r \neq q$, $\diamond\Box(isLeader_r = \texttt{false})$. Due to $\diamond\Box(leaderEpoch_r = e_q)$, even if $isLeader_r$ is set `true` with $leaderEpoch_r < e_q$, r eventually sets $isLeader_r = \texttt{false}$ upon a `roundTripTimer` timeout. Consider any r that sets $isLeader_r$ to `true` with $e_r = leaderEpoch_r > e_q$ at time t'. By Lemma 6.3, $\Box_{t'}(isLeader_q = \texttt{false} \vee leaderEpoch_q > e_r)$. This contradicts (3). Therefore, the protocol implements Ω. □

Theorem 6.5 (Leader Stability) *For the protocol in Figure 3, if a process p sets $\Omega_p(t) = p$ at time t and p remains f-accessible during $[t-\delta, t+\tau]$, then $\Omega_p(t+\tau) = p$ holds, and for no process $q \neq p$ does $\Omega_q(t+\tau) = q$ hold.*

Proof. Because p remains f-accessible from $t-\delta$ to $t+\tau$, p will not experience a `roundTripTimer` timeout between $[t, t+\tau]$. Therefore, $isLeader_p = \texttt{true}$ continues to hold at time $t+\tau$. By definition, $\Omega_p(t+\tau) = p$.

To the contrary, suppose that at some process $q \neq p$, $isLeader_q = \texttt{true}$ holds at time $t+\tau$ as well. Denote by e_p the $leaderEpoch_p$ at time t. By Lemma 6.3, if there is any time t', such that $t \leq t' \leq t+\tau$, and such that at time t', $isLeader_q = \texttt{true}$, then already $leaderEpoch_q > e_p$. However, applying Lemma 6.3 again, this time with q leader at time $t+\tau \geq t'$, process p has $isLeader_p = \texttt{false}$ or $leaderEpoch_p(t+\tau) > leaderEpoch_q(t') > e_p$ at $t+\tau$. Because p does not experience a `roundTripTimer` timeout between $[t, t+\tau]$, $isLeader_p = \texttt{true}$ and $leaderEpoch_p \leq views_p[p].state.epochNum = e_p$ hold at $t+\tau$. Contradiction. □

7 Discussion

The condition we introduced to uphold stability in this paper, namely $n = 2f + 1$, is stronger than what is required in practice. The precise conditions under which Paxos can make progress, and likewise, our stable leader election protocol is guaranteed to operate, are left unspecified. It is worth noting that for both 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 steady-state message complexity $O(n)$.

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.

9 Acknowledgement

We would like to thank Marcos K. Aguilera, Gregory Chockler, Leslie Lamport and Doug Terry for discussions on this topic. We are also grateful to the anonymous reviewers for their insightful comments that helped improve the paper.

References

[1] M. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Stable leader election. In *Proceedings of the 15th International Symposium on Distributed Computing (DISC 2001)*, 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 the Twenty-Second Annual ACM Symposium on Principles of Distributed Computing (PODC 2003)*, 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 the 23rd Annual ACM Symposium on Principles of Distributed Computing (PODC 2004)*, pages 328–337. ACM Press, 2004.

[4] E. Anceaume, A. Fernndez, A. Mostefaoui, G. Neiger, and M. Raynal. A necessary and sufficient condition for transforming limited accuracy failure detectors. *J. Comput. Syst. Sci.*, 68(1):123–133, 2004.

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

[6] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[7] F. Chu. Reducing Ω to $\diamond W$. *Information Processing Letters*, 67(6):298–293, 1998.

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

[9] R. Guerraoui and A. Schiper. Γ-accurate failure detectors. In O. Babaoglu and K. Marzullo, editors, *Proceedings of the 10th International Workshop on Distributed Algorithms, Lecture Notes on Computer Science 1151*, pages 269–286, Bologna, Italy, October 1996. Springer-Verlag.

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

[11] M. Larrea, S. Arvalo, and A. Fernndez. Efficient algorithms to implement unreliable failure detectors in partially synchronous systems. In *Proceedings of the 13th International Symposium on Distributed Computing (DISC 1999)*, pages 34–48, 1999.

[12] M. Larrea, A. Fernndez, and S. Arvalo. Optimal implementation of the weakest failure detector for solving consensus. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS 2000)*, pages 52–59, 2000.

[13] E. K. Lee and C. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 1996)*, pages 84–92, 1996.

[14] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *Proceedings of the 6th Usenix Symposium on Operating System Design and Implementation (OSDI 2004)*, pages 105–120, 2004.

[15] D. Malkhi, F. Oprea, and L. Zhou. Omega meets Paxos: Leader election and stability without eventual timely links. In *19th Intl. Symposium on Distributed Computing (DISC)*, pages 199–213, Cracow, Poland, September 2005.

[16] A. Mostefaoui and M. Raynal. 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)*, pages 329–340. Springer-Verlag LNCS #1738, 1999.

[17] R. D. Prisco, B. Lampson, and N. Lynch. Revisiting the Paxos algorithm. In *Proceedings of the 11th Workshop on Distributed Algorithms(WDAG)*, pages 11–125, 1997.

[18] M. Raynal, A. Mostefaoui, and E. Mourgaya. Asynchronous implementation of failure detectors. In *Int. IEEE Conference on Dependable Systems and Networks (DSN'03), (Track : Dependable Computing and Communications Symposium)*, pages 351–360, San Francisco (CA), June 2003. IEEE Computer Society Press.

[19] M. Raynal, A. Mostefaoui, and C. Travers. Crash-resilient time-free eventual leadership. In *Proceedings of the 23th IEEE Symposium on Reliable Distributed Systems (SRDS'04)*, pages 208–217, Florianopolis (Brasil), October 2004. IEEE Computer Society Press.

[20] C. Thekkath, T. Mann, and E. K. Lee. Frangipani: A scalable distributed file system. In *proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP 1997)*, pages 224–237, 1997.

[21] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th Usenix Symposium on Operating System Design and Implementation (OSDI 2004)*, pages 91–104, 2004.

[22] J. Yang, G. Neiger, and E. Gafni. Structured derivations of consensus algorithms for failure detectors. In *Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing (PODC 1998)*, pages 297–308, 1998.