Failure Detectors and Extended Paxos for k-Set Agreement

Wei Chen[†] Jialin Zhang[‡] Yu Chen[†] Xuezheng Liu[†]

[†]Microsoft Research Asia
{weic,ychen,xueliu}@microsoft.com

[‡]Tsinghua University
zhanggl02@mails.tsinghua.edu.cn

Abstract

Failure detector class Ω_k has been defined in [17] as an extension to failure detector Ω, and an algorithm has been given in [15] to solve k-set agreement using Ω_k in asynchronous message-passing systems. In this paper, we extend these previous works in two directions. First, we define two new classes of failure detectors Ω'_k and Ω''_k, which are new ways of extending Ω, and show that they are equivalent to Ω_k. Class Ω'_k is more flexible than Ω_k in that it does not require the outputs to stabilize eventually, while class Ω''_k does not refer to other processes in its outputs and thus serves as a good basis for the partitioned failure detectors we introduce in [6]. Second, we present a new algorithm that solves k-set agreement using Ω''_k when a majority of processes do not crash. The algorithm is a faithful extension of the Paxos algorithm [11], and thus it inherits the efficiency, flexibility, and robustness of the Paxos algorithm. In particular, it has better message complexity than the algorithm in [15]. Both the new failure detectors and the new algorithm enrich our understanding of the k-set agreement problem. In particular, they serve as the basis of our study on partitioned failure detectors for k-set agreement [6].

MSR-TR-2007-48

1 Introduction

Failure detectors are introduced in [4] to circumvent the impossibility result of solving asynchronous consensus [8]. Their abstractions encapsulate the synchrony conditions of the systems needed to solve asynchronous consensus and other problems in distributed computing. In [4] a rotating-coordinator algorithm is shown to solve consensus in asynchronous systems with a failure detector in class $\diamond\mathcal{S}$ when a majority of processes are correct (i.e., they do not crash). In [3], failure detector class Ω, which is equivalent to $\diamond\mathcal{S}$ [7], is shown to be the weakest failure detector class solving consensus. Class Ω is often referred to as leader electors. It requires that each process outputs one process, and eventually all processes output the same correct process.

Around the same time period, Lamport designed the Paxos algorithm that also solves consensus in systems with a majority of correct processes [11]. Although implicit, the Paxos algorithm essentially uses leader electors Ω. The core of the Paxos algorithm is similar to the rotating-coordinator algorithm in [4], but the Paxos algorithm has a number of attractive features in its efficiency, flexibility, and robustness. Due to these features, the Paxos algorithm has been implemented as a core service in a number of distributed systems (e.g. [13, 2]).

The problem of k-set agreement is introduced in [5] as a generalization of the consensus problem. In k-set agreement, each process from a set of $n > k$ processes proposes a value, and makes an irrevocable decision on one value. It needs to satisfy the following three properties: (1) *Validity*: If a process decides v, then v has been proposed by some process. (2) *Uniform k-Agreement*: There are at most k different decision values. (3) *Termination*: Eventually some correct process decides.[1]

It has been shown that k-set agreement cannot be solved if k processes may crash in the system [1, 10, 18], and a number of studies have introduced various failure detectors to circumvent this impossibility result [19, 16, 9, 14, 15]. In [15], Mostefaoui et.al. summarizes the relationship among these failure detectors and show that class Ω_k is the weakest among them. A failure detector in Ω_k outputs a set of at most k processes and eventually the outputs on all correct processes converge to the same set of processes that contains at least one correct process. It is an extension of Ω, and is originally introduced in [17] for studying wait-free hierarchy in shared memory systems. In [15] an algorithm is also presented to solve k-set agreement using Ω_k in systems with a majority of correct processes.

In this paper, we extend both the study on Ω_k and the study on the algorithm for k-set agreement. We define two new classes of failure detectors Ω'_k and Ω''_k as different ways to extend Ω, and show that they are equivalent to Ω_k by transformations between them in asynchronous systems. Each new class has its own feature. Failure detectors in Ω'_k output a single process, which is required to be a correct process eventually (same as Ω), while the total number of processes appearing in the outputs infinitely often is at most k. Ω'_k is more flexible than Ω_k in that it does not require that the outputs of the failure detector on all processes eventually stabilize. Failure detectors in Ω''_k output a Boolean value indicating whether the process itself is a leader, and eventually the outputs stabilize and the number of leaders is at least one and at most k. Ω''_k differs from Ω_k and Ω'_k in that its outputs do not refer to other processes in the system. This feature is particularly convenient when we introduce partitioned failure detectors in [6], since we do not need to concern about whether the failure detector outputs refer to processes in the same partitioned component or not. Therefore, Ω''_k serves as the basis for our study of partitioned failure detectors in [6].

To show the equivalence of these failure detector classes in the amount of information they provide, we show that they can be transformed into one another. Moreover, we demand that the transformation algorithms be *parameter-free*, which means they do not contain any parameters such as the value of k. In

[1] In asynchronous systems with reliable channels, a correct process that decides can send out its decision value to all processes so that all correct processes eventually decide. Therefore, our Termination property implies a different version that requires all correct processes eventually decide.

other words, the information about the parameter k is contained within the failure detector outputs, not provided by the transformation algorithms. Hence, the transformations are generic ones working for any parameter k. This also leads to an additional output *lbound* in Ω'_k and Ω''_k to replace the fixed parameter k. The *lbound* outputs are numbers of at most k, and they eventually stabilize to a single value, which is the upper bound on the number of leaders eventually appearing in the system. This *lbound* output makes the transformations between Ω_k, Ω'_k and Ω''_k parameter-free.

In the paper, we show the circular transformations from Ω_k to Ω''_k, then from Ω''_k to Ω'_k, and finally from Ω'_k to Ω_k. The first two transformations are very simple, while the third one is significantly more complicated. The simplicity of the first two transformations relates to the eventually stable outputs of the failure detectors in Ω_k and Ω''_k. On the contrary, failure detectors in Ω'_k do not have eventually stable outputs, and thus it requires more communication and processing effort to construct a failure detector with stable outputs in Ω_k. In a precise sense, the transformation from Ω'_k to Ω_k shows the difference in the information complexity between these classes. The implication from this difference is that, it may be more convenient using Ω_k and Ω''_k to solve problems such as k-set agreement, while it may be more convenient to use Ω'_k to show it is implementable in certain systems or transformable from other failure detectors.

Next, we show how to extend the Paxos algorithm using Ω to a new algorithm using Ω''_k to solves k-set agreement, in systems with a majority of correct processes. The key idea of the extension is that, while in the Paxos algorithm each acceptor can only accept one round and thus commit to support only one proposer at a time, our algorithm allows each acceptor to accept up to k rounds and thus commit to support up to k proposers simultaneously. The realization of this idea is not entirely straightforward, and it leads to our full algorithm that handles all possible scenarios.

Our algorithm has several features. First, the algorithm is parameter-free, the set agreement number k that it solves is purely determined by the outputs of failure detectors in Ω''_k. This makes the algorithm generic for solving set agreement with any number k, and makes it as our basis to study the algorithm for k-set agreement with partitioned failure detectors in [6]. The algorithm in [15] is also parameter-free, so we match its generality in this sense.

Second, and more impotantly, our algorithm is a faithful extension to the original Paxos algorithm, and so it inherits the efficiency, flexibility and robustness of the Paxos algorithm. For efficiency, in normal runs where all n processes are correct and the ℓ leaders elected by Ω''_k are stable from the beginning, our algorithm only cost $O(\ell n)$ messages, better than the $O(n^2)$ messages needed by the algorithm in [15]. Moreover, same as the Paxos algorithm, our algorithm allows the efficent batching of many instances of k-set agreement together, so the amortized time complexity of completing one instance of k-set agreement is one round-trip time, which matches that of the algorithm in [15]. For flexibility, our algorithm allows assigning different processes to different roles. In particular, only the proposers need access to failure detectors while acceptors could be purely reactive processes. For robustness, our algorithm can also be made to tolerate transient failure by keeping several key state variables in stable storage as in Paxos. Therefore, we successfully extend the Paxos algorithm and inherits its features to the context of k-set agreement.

Overall, our contributions are both on the study of new failure detectors for k-set agreement, and on the study of extending the Paxos algorithm to solve k-set agreement. We believe that our study enriches the understanding of the k-set agreement and its associated failure detectors. In particular, it serves as the basis for our next-step study on partitioned failure detectors for k-set agreement.

The rest of the paper is organized as follows. Section 2 describes our system model. Section 3 defines the new failure detectors and shows their equivalence. Section 4 presents the extended Paxos algorithm and discusses its features. Section 5 concludes the paper. The correctness proof of the extend Paxos algorithm is provided in the appendix.

2 System Model

We consider asynchronous message passing distributed systems augmented with failure detectors. Our formal model is the same as the model in [3], and we explain the main points in this section.

We consider a system with n ($n > k$) processes $P = \{p_1, p_2, \ldots, p_n\}$. Let $\mathcal{T}$ be the set of time values, which are non-negative integers. Processes do not have access to the global time. A *failure pattern* F is a function from $\mathcal{T}$ to 2^P, such that $F(t)$ is the set of processes that have failed by time t. Let *correct*(F) denote the set of *correct processes*, those that do not crash in F. A *failure detector history* H is a function from $P \times \mathcal{T}$ to an output range $\mathcal{R}$, such that $H(p, t)$ is the output of the failure detector module of process $p \in P$ at time $t \in \mathcal{T}$. A *failure detector* $\mathcal{D}$ is a function from each failure pattern to a set of failure detector histories, representing the possible failure detector outputs under failure pattern F.

Processes communicate with each other by sending and receiving messages over communication channels, which are available between every pair of processes. Channels are reliable in that it does not create or duplicate messages, and any message sent to any correct process is eventually received.

A deterministic algorithm A using a failure detector $\mathcal{D}$ executes by taking *steps*. In each step, a process p first receives a message (could be a null message), queries its failure detector module, then changes its local state and sends out a finite number of messages to other processes. Each step is completed at one time point t, but the process may crash in the middle of taking its step. All steps have to be legitimate, which means under failure pattern F and a failure detector history $H \in \mathcal{D}(F)$, if p takes a step at time t and receives a message m from q, then $p \notin F(t)$, p's failure detector query output is $H(p, t)$, and there must be a step before t such that q sends m to p in that step. A *run* of algorithm A with failure detector $\mathcal{D}$ is an infinite sequence of such steps such that (a) every correct process takes an infinite number of steps, and (b) every message sent to a correct process is eventually received.

We consider the asynchronous system model, which means there is no bound on the delay of messages and the delay between steps that a process takes.

We say that a failure detector class $\mathcal{C}_1$ is *weaker than* a failure detector class $\mathcal{C}_2$, if there is a transformation algorithm T such that using any failure detector in $\mathcal{C}_2$, algorithm T implements a failure detector in $\mathcal{C}_1$. In this case, we denote it as $\mathcal{C}_1 \preceq \mathcal{C}_2$ and also refer to it as $\mathcal{C}_2$ can be transformed into $\mathcal{C}_1$. We say that $\mathcal{C}_1$ and $\mathcal{C}_2$ are equivalent if $\mathcal{C}_1 \preceq \mathcal{C}_2$ and $\mathcal{C}_2 \preceq \mathcal{C}_1$.

3 Ω_k-like failure detectors

In this section, we provide the formal specifications of the two new classes of failure detectors Ω'_k and Ω''_k, and then show that they are equivalent to Ω_k. We provide the formal specification of Ω_k first.

Failure detectors in Ω_k outputs a set *Leaders*, which is a set of processes to be considered as leaders. A failure detector $\mathcal{D}$ is in the class Ω_k, if for any failure pattern F and any failure detector history $H \in \mathcal{D}(F)$, we have:

(Ω1) For any output, its size is at most k. Formally, $\forall t \in \mathcal{T}, \forall p \notin F(t), |H(p, t)| \leq k$.

(Ω2) Eventually, all failure detector modules output the same set of processes. Formally, $\exists t_0 \in \mathcal{T}, \forall t_1, t_2 \geq t_0, \forall p_1 \notin F(t_1), \forall p_2 \notin F(t_2), H(p_1, t_1) = H(p_2, t_2)$.

(Ω3) Eventually, at least one process in any output is correct. Formally, $\exists t_0 \in \mathcal{T}, \forall t \geq t_0, \forall p \notin F(t), \exists q \in \mathit{correct}(F), q \in H(p, t)$.

3.1 Specification of Ω'_k

As described in the introduction, for Ω'_k, we aim at failure detectors in which each process only select one process as a leader, not a set of processes as in Ω_k. Moreover, we would like to have more flexible failure detectors whose outputs are not required to eventually stabilize as in Ω_k.

More precisely, the output of Ω'_k is (*leader*, *lbound*), where *leader* is a process that p believes to be the leader at the moment, and *lbound* is a non-negative number that p believes to be the upper bound of the number of possible leaders in the system. We denote $H(p,t).leader$ and $H(p,t).lbound$ the *leader* part and the *lbound* part of outputs respectively for a failure detector history H.

A failure detector $\mathcal{D}$ is in the class Ω'_k if for any failure pattern F and any failure detector history $H \in \mathcal{D}(F)$, we have:

($\Omega'1$) The *lbound* outputs never exceed k. Formally, $\forall t \in \mathcal{T}, \forall p \notin F(t), H(p,t).lbound \leq k$.

($\Omega'2$) Eventually, the *lbound* outputs of all processes do not change and are the same. Formally, $\exists t_0 \in \mathcal{T}, \forall t_1, t_2 \geq t_0, \forall p_1 \notin F(t_1), \forall p_2 \notin F(t_2), H(p_1,t_1).lbound = H(p_2,t_2).lbound$.

($\Omega'3$) Eventually, the leader output on every process is always a correct process. Formally, $\exists t_0 \in \mathcal{T}, \forall t \geq t_0, \forall p \notin F(t), H(p,t).leader \in correct(F)$.

($\Omega'4$) Eventually, the number of leaders is bounded by *lbound*. Formally, $\exists t_0 \in \mathcal{T}, \forall t \geq t_0, \forall p \notin F(t), |\{H(q,t').leader \mid t' > t_0, q \notin F(t')\}| \leq H(p,t).lbound$.

Several remarks are in order for the above definition. First, one may see that properties ($\Omega'1$) and ($\Omega'2$) can be trivially satisfied by hard-coding *lbound* to k. This, however, means that one has to pre-determine the parameter k. This is not the case for Ω_k, because according to ($\Omega1$), the parameter k could be any value that is at least the maximum size of the *Leaders* outputs in a run. The implication is that, if we hard-code *lbound* to k, any transformation from Ω_k to Ω'_k has to know the value of k in advance and it cannot derive k from the outputs of Ω_k. In this case, the transformation is not parameter-free, and Ω'_k is not as general as Ω_k.

Second, one may see that even if we keep *lbound* outputs and property ($\Omega'1$), property ($\Omega'2$) can be satisfied by processes exchanging their *lbound* values and taking the maximum value they see as their own *lbound* outputs. The reason we keep this property is again to match the generality of Ω_k, in which the size of the *Leaders* outputs may decrease. Thus, we prefer that *lbound* values, which essentially match to the sizes of *Leaders* outputs in Ω_k, to be able to decrease.

Third, properties ($\Omega'3$) and ($\Omega'4$) do not require that eventually the *leader* outputs stabilize. Processes may keep changing their *leader* outputs, as long as they point to at most ℓ correct processes, where ℓ is the eventual *lbound* value in the run. This is different from Ω_k, which requires that the outputs of a failure detector eventually stabilize. From the complexity of the transformation from Ω'_k to Ω_k provided in the next section, we can see that generating stable outputs required by Ω_k indeed demands more work. Therefore Ω'_k provides a different way of extending the original Ω failure detector, and it is more flexible in that it does not require the outputs to stabilize eventually.

3.2 Specification of Ω''_k

We now introduce the third class of failure detectors Ω''_k. Failure detectors in Ω''_k outputs $(isLeader, lbound)$, where *isLeader* is a Boolean variable indicating whether this process is a leader or not, and *lbound* is a non-negative integer with the same meaning as in Ω'_k. We say that a process p is an *eventual leader* (in a failure detector history) in Ω''_k if p is correct and there is a time after which p's *isLeader* outputs are always *True*.

A failure detector $\mathcal{D}$ is in the class Ω''_k if for any failure pattern F and any failure detector history $H \in \mathcal{D}(F)$, we have:

($\Omega''1$) The *lbound* outputs never exceed k. Formally, $\forall t \in \mathcal{T}, \forall p \notin F(t), H(p,t).lbound \leq k$.

($\Omega''2$) Eventually, the *lbound* outputs of all processes do not change and are the same. Formally, $\exists t_0 \in \mathcal{T}, \forall t_1, t_2 \geq t_0, \forall p_1 \notin F(t_1), \forall p_2 \notin F(t_2), H(p_1,t_1).lbound = H(p_2,t_2).lbound$.

($\Omega''3$) Eventually the *isLeader* outputs on any correct process do not change. Formally, $\exists t \in \mathcal{T}, \forall p \in correct(F), \forall t' > t, H(p,t).isLeader = H(p,t').isLeader$.

($\Omega''4$) There is at least one eventual leader. Formally, $|\{p \in correct(F) \mid \exists t, \forall t' > t, H(p,t').isLeader = True\}| \geq 1$.

($\Omega''5$) The number of eventual leaders is eventually bounded by the *lbound* outputs. Formally, $\exists t_0 \in \mathcal{T}, \forall t_1 \geq t_0, |\{p \in correct(F) \mid \exists t, \forall t' > t, H(p,t').isLeader = True\}| \leq H(p,t_1).lbound$.

In the specification, the properties about *lbound* outputs are the same. For the *isLeader* outputs, ($\Omega''3$) requires that the *isLeader* output eventually stabilize, while ($\Omega''4$) and ($\Omega''5$) require the number of eventual leaders to be at least one and at most the eventual value of *lbound*.

The main feature of Ω''_k is that its outputs only include a Boolean value that refers to the leader status of each process itself, and it does not refer to other processes as in Ω_k and Ω'_k. This is enough for the original Paxos algorithm and the extended Paxos algorithm in Section 4, since a proposer process only needs to know if itself is a leader to initiate a new proposer round. Moreover, this feature fits particularly well for the partitioned failure detectors we introduce in [6]. When processes are partitioned into multiple components, the *isLeader* outputs of a process p still naturally refer to the leadership status of p itself, while for the *Leaders* outputs in Ω_k and *leader* outputs in Ω'_k of a process p, we have to put extra requirements on whether these referred processes are in the same component as p's or not.

3.3 Equivalence of Ω_k, Ω'_k, and Ω''_k

To show the equivalence, we show three parameter-free transformation algorithms: the first one is from Ω_k to Ω''_k, the second one is from Ω''_k to Ω'_k, and the last one is from Ω'_k to Ω_k.

The transformation from Ω_k to Ω''_k is almost trivial: each process p sets its *lbound* output of Ω''_k to be the size of the *Leaders* outputs of Ω_k, and sets its *isLeader* output to true if any only if p itself appears in the *Leaders* output of Ω_k. This transformation does not involve any messages and is parameter-free. It is straightforward to verify its correctness.

Lemma 1 *Failure detector class Ω_k can be transformed into Ω''_k, for any $k \geq 1$.*

Transforming failure detector class Ω''_k to Ω'_k is also straightforward. The *lbound* of Ω''_k is directly transferred to *lbound* of Ω'_k without change. Each process p periodically checks its *isLeader* value in Ω''_k, and if it is true, send a heartbeat message to all processes. Whenever a process q receives a heartbeat message from p, q sets its *leader* output of Ω'_k to p. Obviously, this transformation is parameter-free, and it is very simple to verify that the transformation is correct. Thus we have:

Lemma 2 *Failure detector class Ω''_k can be transformed into Ω'_k, for any $k \geq 1$.*

We now focus on the transformation from Ω'_k to Ω_k, which are significantly more complicated than the previous two. The complication comes from the requirement of stabilizing the *Leaders* outputs of Ω_k and make sure one of processes in *Leaders* is correct. Figure 1 shows this transformation.

The basic idea is for each process p_i to periodically send their *leader* outputs of Ω'_k to all processes (line 9), and for each p_j, p_i counts the number of times p_i sees p_j as a leader in a message (line 12). Then p_i sorts all processes into an array $A[1..n]$ based on the counter values, and use process ID to break the tie (line 13).

Let L be the set of processes that appear infinitely often in the messages of a run of the tranformation algorithm. Let $\ell = |L|$. Let $\mathsf{setof}(A[i..j])$ denote the set $\{A[x] \mid i \leq x \leq j\}$. Then we have the following property for array $A[]$.

Proposition 3 *Eventually, on all correct processes we have $\mathsf{setof}(A[1..\ell]) = L$ and $A[1]$ is a correct process.*

Proof. This is because eventually, only the counters of processes in L increase. For each process p in L, since it appears infinitely often in the messages, it must appear infinitely often in the messages sent by some correct process, say q. Then, each correct process receives infinite messages from q, so that their counters of p will increase infinitely often. For each process p' not in L, its counter on any correct process can only be incremented a finite number of times. So, eventually, all correct processes have $\mathsf{setof}(A[1..\ell]) = L$. Since Ω'_k always outputs a *leader*, $\ell \geq 1$. By $(\Omega'3)$ we know that $L \subseteq \mathit{correct}(F)$. So $A[1]$ must be a correct process eventually. □

With the above property, if all processes could output $\mathsf{setof}(A[1..\ell])$ as their *Leaders* outputs, the properties of Ω_k would be satisfied. However, processes do not know ℓ, so instead they output $\mathsf{setof}(A[1..s])$ for some index pointer s (line 10), and try to stabilize s and $\mathsf{setof}(A[1..s])$. To do so, each process exchanges its array $A[]$ with other processes (line 9), increases its s value from 1 until it finds a matching $\mathsf{setof}(A[1..s])$ with the received $A[]$ (lines 17 and 19), and if s goes beyond its *lbound* output, it is wrapped around to 1 and a wrap-around counter w is incremented (lines 16 and 18). The result is shown by the following proposition.

Proposition 4 *The values of (w, s) eventually stabilize to the same value on all correct processes, and all $\mathsf{setof}(A[1..s])$'s are the same.*

Proof. For each process p, p either increases s and keeps w (according lines 17 and 19), or increases w when a wrap-around occurs (lines 16 and 18), or takes the higher values it sees (line 15) when exchanging its (w, s) values. Therefore, (w, s) monotonically increases on each process.

Let $(maxw(t), maxs(t))$ be the highest (w, s) value among all correct processes at time t. Values of $(maxw(t), maxs(t))$ are also nondecreasing. We need to show $(maxw(t), maxs(t))$ eventually stops increasing.

By $(\Omega'2)$ eventually the *lbound* outputs on all processes stabilize to a single value. Let b denote this value. By Proposition 3, all correct processes eventually have $\mathsf{setof}(A[1..\ell])$. By $(\Omega'4)$, $\ell \leq b$. Let t_0 be the time such that all of the above properties occur and only correct processes are left. Let $t_1 > t_0$ be the time such that no message sent before t_0 will be received at or after t_1. Time t_1 exists because there are only a finite number of messages sent before t_0. Let $(w_1, s_1) = (maxw(t_1), maxs(t_1))$. We claim that $(maxw(t), maxs(t))$ will not exceed $(w_1 + 1, \ell)$.

We prove this claim by contradiction. Suppose that $(maxw(t), maxs(t))$ eventually exceeds $(w_1 + 1, \ell)$. Let $t_2 > t_1$ be the earliest time when some correct process p_i changes its (w, s) to a value higher than $(w_1 + 1, \ell)$. Process p_i updates its (w, s) only when p_i receives a message $msg = (\mathit{leader}_j, \mathit{lbound}_j, A_j[1..\mathit{lbound}_j], s_j, w_j)$ from a process p_j. By the definition of t_1, we

On node p_i:

1 Global variables:
2     $(\mathit{leader}, \mathit{lbound})$: output of Ω'_k, read-only
3   *Leaders*: output of Ω_k, initially $\{p_i\}$
4     $c[p_1..p_n]$: counters for all processes, initially 0
5     $A[1..n]$: permutation of $(p_1, \ldots, p_n)$, such
        that for all $1 \leq x < y \leq n$,
        $(c[A[x]], A[x]) > (c[A[y]], A[y])$
6   s: an index pointer for array $A[]$, initially 1
7   w: counter for the number of wrap-arounds
        of s, initially 0

8 Repeat periodically:
9   **for each** $p_j \in P$,
        send $(\mathit{leader}, \mathit{lbound}, A[1..\mathit{lbound}], s, w)$ to p_j
10   *Leaders* $\leftarrow \mathsf{setof}(A[1..s])$

11 Upon receipt of $(\mathit{leader}_j, \mathit{lbound}_j, A_j[1..\mathit{lbound}_j],$
        $s_j, w_j)$ from a node p_j:
12     $c[\mathit{leader}_j] \leftarrow c[\mathit{leader}_j] + 1$
13   rearrange $A[1..n]$ based on its sorting order
14   **if** $\mathit{lbound}_j \neq \mathit{lbound}$ **then return**
15     $(w, s) \leftarrow \max((w, s), (w_j, s_j))$
16   **if** $s > \mathit{lbound}_j$ **then** $(w, s) \leftarrow (w + 1, 1)$ **return**
17     $T \leftarrow \{s' \mid s \leq s' \leq \mathit{lbound}_j,$
        $\mathsf{setof}(A[1..s']) = \mathsf{setof}(A_j[1..s'])\}$
18   **if** $T = \emptyset$ **then** $(w, s) \leftarrow (w + 1, 1)$ **return**
19     $s \leftarrow$ smallest s' in T

Figure 1: Transformation from Ω'_k to Ω_k.

know that this message is sent after t_0, and thus p_j must be a correct process, $\mathsf{setof}(A_j[1..\ell]) = L$ and $\mathit{lbound}_j = b$.

There are four lines in the algorithm where p_i may update its (w, s) value, and we now examine each of them and show that none of the lines can be executed by p_i when it receives msg at time t_2. In line 15, p_i sets its (w, s) to the maximum of its local (w, s) value and the received (w_j, s_j). This cannot be the line that increases p_i's (w, s) beyond $(w_1 + 1, \ell)$, because p_i is the first process to do so. In line 16, p_i sets its (w, s) to $(w + 1, 1)$ when $s > \mathit{lbound}_j$. If this is the line that increases p_i's (w, s) beyond $(w_1 + 1, \ell)$, then we have that before executing this line variable w is w_1+1. Since before executing this line, we have $(w, s) \leq (w_1 + 1, \ell)$, we have $s \leq \ell$. Since we already know that $\ell \leq b$ and $\mathit{lbound}_j = b$, condition $s > \mathit{lbound}_j$ does not hold, and thus p_i will not update (w, s) in line 16. Suppose that line 18 is the one that increases p_i's (w, s) beyond $(w_1 + 1, \ell)$. We also have that before executing this line $w = w_1 + 1$ and $s \leq \ell$. Since we know that $\ell \leq b$, $\mathit{lbound}_j = b$, and $\mathsf{setof}(A[1..\ell]) = \mathsf{setof}(A_j[1..\ell]) = L$, ℓ must be in the set T computed in line 17. Thus, $T \neq \emptyset$ and p_i will not update (w, s) in line 18. Finally suppose that line 19 is the one that increases p_i's (w, s) beyond (w_1+1, ℓ). Since in the line only s is changed, we still have that $w = w_1+1$ and $s \leq \ell$. By the same argument as above, we have $\ell \in T$. Therefore, the smallest s' in T must be at most ℓ, and thus after the update we still have $s \leq \ell$. So the update in this line will not increase (w, s) beyond $(w_1 + 1, \ell)$.

We have examined all lines that update (w, s) and conclude that our claim is correct, that is $(maxw(t), maxs(t))$ will not exceed $(w_1 + 1, \ell)$.

Let (w_m, s_m) be the final maximum value obtained on any correct process. Since the channels between the correct processes are reliable, each correct processes eventually receive all messages from other correct processes. Since eventually all correct processes have the same *lbound* value by $(\Omega'2)$, the condition in line 14 will be false and the correct processes always execute line 15. Then we know all correct processes eventually have the same (w_m, s_m) value. Therefore, no process updates its (w, s) in lines 16 and 18 any more, and whenever a process executes line 19, the smallest s' in T is always the same as s, which means $s \in T$ and $\mathsf{setof}(A[1..s]) = \mathsf{setof}(A_j[1..s])$. □

Note that the final value of s could be either less than ℓ or greater than ℓ, so the eventual *Leaders* output may not be L and may contain non-correct processes. Lemma 5 proves the correctness of the transformation.

Lemma 5 *The algorithm in Figure 1 transforms any failure detector in Ω'_k into a failure detector in Ω_k.*

Proof. We fix an arbitrary failure pattern F, an arbitrary failure detector history H of Ω'_k under F, and an arbitrary run of the algorithm in Figure 1 with the failure pattern F and the failure detector history H.

First, according to line 10, $|\mathit{Leaders}|$ is bounded by the index s, which can be either set to 1 or increased in line 19. By line 17, whenever s is increased in line 19, s is bounded by some *lbound* value. By property $(\Omega'1)$, we thus see that s is at most k at all times. Therefore, the *Leaders* output contains at most k processes, and thus $(\Omega1)$ holds.

By Proposition 3, eventually on any correct process $A[1]$ must be a correct process. Therefore, $(\Omega3)$ holds.

By Proposition 4 we know that the values of (w, s) eventually stabilize to the same value on all correct processes, and all $\mathsf{setof}(A[1..s])$'s are the same. Since the *Leaders* output for Ω_k is taken as $\mathsf{setof}(A[1..s])$ (line 10), we know that eventually the *Leaders* output does not change, and they are the same among all correct processes. Therefore property $(\Omega2)$ holds. □

From the algorithm and its proof, we see that a significant amount of information exchange and manipulation is needed to construct Ω_k out of Ω'_k. This indicates that the requirement of Ω_k is rigid and less flexible. Thus, when we study how to implement failure detectors in Ω_k or how to show another class of failure detectors can be transformed into Ω_k, it could be more complicated and require more work. However, with Ω'_k as a more flexible alternative, the above tasks could be simplified.

Moreover, notice that the transformation algorithm is parameter-free, so it is generic for any parameter k, which means Ω'_k contains all the information and the algorithm does not provide any more information to construct Ω_k.

3.4 Summary of Ω_k, Ω'_k, and Ω''_k

With Lemmata 1, 2, and 5, we can now state the following theorem.

Theorem 1 *The failure detector classes Ω_k, Ω'_k, and Ω''_k are equivalent for any $k \geq 1$.*

Thus, we provide two new classes of failure detectors that are equivalent to Ω_k, and they enrich our understanding of the different aspects that Ω_k may bring. Class Ω'_k shows that Ω_k-like leader electors can be made to be single leader output as the original Ω, and can be flexible without eventual stabilization requirements. The complexity of the transformation from Ω'_k to Ω_k shows in a precise way that the cost one may save if one does not need the eventual stabilization requirements and only needs the more flexible Ω'_k. Class Ω''_k shows that one can also use Boolean outputs to avoid refering to other processes in the system, and is suitable for partitioned failure detectors in [6].

Our study aims at parameter-free transformations, so it reflects the true equivalence among the classes of failure detectors. For example, the *lbound* outputs introduced in Ω'_k and Ω''_k are to match the flexible information that Ω_k provides and to allow parameter-free transformations. If we were to replace *lbound* with a fixed value k, then we would lose this flexibility, and it would be difficult to generalize Ω''_k to the partitioned failure detectors in [6].

4 Extended Paxos algorithm

In this section, we present an algorithm that solves k-set agreement problem using Ω''_k in systems with a majority of correct processes. The algorithm is an extension to the Paxos algorithm [11] for solving consensus.

4.1 Algorithm and its description

Figures 2 and 3 present the extended Paxos algorithm for k-set agreement using failure detectors in Ω''_k in a system where a majority of processes are correct. We use similar terminologies as in the Paxos algorithm summarized in [12]. Each process behaves both as a proposer and an acceptor (see Section 4.2 for the extension of this point). Proposers are active participants driving the progress in a round-by-round fashion, while acceptors are passive participants responding to proposers' requests. A proposer p periodically checks its failure detector output to see if it is currently a leader, and if so and it is not already in a round, it starts a new round with round number *p_round* (lines 6–11). Each round of proposer p has two phases: the *preparation phase* and the *acceptance phase*. In the preparation phase (lines 12–19), p sends a PREPARE message to all acceptors, waits for responses from the acceptors, and either quits this round or selects a new *est* value as the candidate for its decision. In the acceptance phase, (lines 20–24), p sends its *est* value in an ACCEPT message to all acceptors, waits for responses from the acceptors, and either decides on *est* when it receives a majority of ACK-ACC messages, or quits this round otherwise. This basic structure is the same as the Paxos algorithm. We now focus on the new extensions to the algorithm.

In the Paxos algorithm, each acceptor can only accept one round at any time, and thus support only one proposer at any time. This works well with Ω failure detectors that elect a single leader eventually to achieve consensus. For k-set agreement with Ω''_k failure detectors, the key extension is that each acceptor can accept multiple rounds at the same time, and thus it may support multiple proposers who believe they are leaders according to Ω''_k. The acceptors need to control the number of rounds it can accept simultaneously. This leads to the introduction of state variables *p_Rounds*, *a_Rounds* and *a_TS*, which we explain below.

Given a set of rounds R and a positive integer m, We define *top*(R, m), $\cup_m$, and $\preceq_m$, such that *top*(R, m) is a function returning the m high-

On proposer p with unique id $i \in \{1, \ldots, n\}$:

Proposer variables:

1 $proposal$: the initial proposal value, read-only
2 (*isLeader*, *lbound*): Ω''_k output, read-only
3 *p_round*: current round number, initially process id i
4 *p_Rounds*: top n rounds that p sees, initially $\{i\}$
5 *taskid*: unique id for each task started, initially 0

Run periodically if not decided yet

6 **if** *isLeader* = *True* **and** no task 1 running **then**
7 *taskid* ← *taskid* + 1;
8 **if** *p_round* ∉ *top*(*p_Rounds*, *lbound*) **then**
9 *p_round* ← *p_round* + $t \cdot n$ such that *p_round* + $t \cdot n >$ max *p_Rounds*
10 *p_Rounds* ← *p_Rounds* $\cup_n$ {*p_round*}
11 start task 1

Task 1: one round of p

12 send (PREPARE, *p_round*, *p_Rounds*, *lbound*, *taskid*) to all acceptors
13 wait until [(1) received (NACK-PREP, R, *taskid*) from an acceptor; **or** (2) received (ACK-PREP, R, *TS*, v, *taskid*) from more than $n/2$ acceptors]
14 M_1 ← {(ACK-PREP, R, *TS*, v, *taskid*) received from acceptors}
15 M_2 ← {(NACK-PREP, R, *taskid*) received from acceptors}
16 *p_Rounds* ← *p_Rounds* $\cup_n$ $(\bigcup_{m \in M_1 \cup M_2} m.R)$
17 **if** (1) $M_2 \neq \emptyset$ **or** (2) some received R's in M_1 are different **then** stop this task
18 **if** $\forall m \in M_1, m.v = \bot$ **then** *est* ← $proposal$
19 **else** *est* ← $m.v$ with $m \in M_1$ and the highest $m.TS$ (based on $\preceq_n$ order)
20 send (ACCEPT, *est*, *p_Rounds*, *taskid*) to all acceptors
21 wait until [(1) received (NACK-ACC, R, *taskid*) from an acceptor; **or** (2) received (ACK-ACC, *taskid*) from more than $n/2$ acceptors]
22 **if** (1) **then**
23 *p_Rounds* ← *p_Rounds* $\cup_n$ R; stop this task
24 decide(est)

Figure 2: Extended Paxos algorithm for k-set agreement using Ω''_k. Part I: proposer thread.

On acceptor q:

Acceptor variables:

25 *a_Rounds*: top n rounds that q sees, initially $\emptyset$
26 *a_est*: estimate of the final value, initially $\bot$;
27 *a_TS*: top n rounds that q sees when q accepts a value, initially $\emptyset$

28 Upon receipt of (PREPARE, r, R, lb, *taskid*) from p
29 *a_Rounds* ← *a_Rounds* $\cup_n$ R
30 **if** r ∉ *top*(*a_Rounds*, lb) **then** send (NACK-PREP, *a_Rounds*, *taskid*) to p
31 **else** send (ACK-PREP, *a_Rounds*, *a_TS*, *a_est*, *taskid*) to p

32 Upon receipt of (ACCEPT, v, R, *taskid*) from p
33 *a_Rounds* ← *a_Rounds* $\cup_n$ R
34 **if** $R \neq$ *a_Rounds* **then** send (NACK-ACC, *a_Rounds*, *taskid*) to p
35 **else**
36 (*a_est*, *a_TS*) ← (v, R)
37 send (ACK-ACC, *taskid*) to p

Figure 3: Extended Paxos algorithm for k-set agreement using Ω''_k. Part II: acceptor thread.

est round numbers in R, $\cup_m$ is an operator such that $R_1 \cup_m R_2 = top(R_1 \cup R_2, m)$, and $\preceq_m$ is a partial order such that $R_1 \preceq_m R_2$ if and only if $R_1 \cup_m R_2 = R_2$.

Variables *p_Rounds* and *a_Rounds* keep two sets of at most n rounds that proposer p and acceptor q may work with, respectively. Proposers and acceptors exchange their *p_Rounds* and *a_Rounds* values and merge the value received into their own value using operator $\cup_n$ (lines 16, 23, 29, 33). The result is that *p_Rounds* values on proposer p keeps increasing (based on order $\preceq_n$), so do the *a_Rounds* values on acceptor q. Essentially, *p_Rounds* and *a_Rounds* record the top n rounds that p and q see so far, respectively.

Based on the value of *a_Rounds*, acceptor q only accepts a PREPARE message from a proposer p if p's current round number *p_round* is in the top lb rounds that q sees, where lb is the *lbound* output when p sends the message (line 30). If q accepts the round, q sends an ACK-PREP message with its current *a_est* value and a kind of timestamp *a_TS* (to be explained shortly) to p; otherwise q sends a NACK-PREP message to p.

If p receives a NACK-PREP message in its preparation phase, it stops waiting for other messages (line 13), updates its *p_Rounds* value (line 16), and quits the round (line 17). When the next time p starts a task for a new round, it checks to make sure its *p_round* is in *top*(*p_Rounds*, *lbound*), and if not so, it selects a new *p_round* that is higher than any round numbers in *p_Rounds* and merge it into *p_Rounds* (lines 6–11). This is to guarantee that p's round will eventually be accepted by acceptors.

Another case where p may quit its preparation phase is that among the ACK-PREP messages it has received, the *a_Rounds* values from the acceptors are not the same. (line 17, condition (2)). This to ensure that the majority of acceptors are all accepting the same set of rounds for the safety of k-set agreement. For liveness, eventually all *p_Rounds* and *a_Rounds* will converge so proposers will not always quit their preparation phases due to this condition.

If p receives ACK-PREP messages from a majority of acceptors with the same *a_Rounds* values, p can complete its preparation phase by selecting a new candidate value *est* for its decision. If p does not see any value from the acceptors, it uses its own proposal value (line 18). If p sees some values from the acceptors, it selects the value with the highest timestamp *TS* among the messages it received, based on the partial order $\preceq_n$ (line 19). To ensure that this selection can be done, we need to show that all *TS* values form a total order based on $\preceq_n$. This is due to the majority intersection property and the condition (2) in line 17, and is shown in the proof as Corollary 14.

After p selects a new *est* value, it enters the acceptance phase by sending an ACCEPT message with the *est* value to all acceptors (line 20). The purpose is to let at least a majority of acceptors to record this value and support it. When acceptor q receives this message, it first updates its *a_Rounds* (line 33), and then check if the received *p_Rounds* is the same as the updated *a_Rounds* value, and if it is not the same, it rejects the acceptance phase by sending a NACK-ACC message with its *a_Rounds* value back to p (line 34). This is to guarantee that if proposer p successfully decides in its acceptance phase, its *p_Rounds* value must remain the same during the phase, which is important to our proof of the Uniform k-Agreement property. If q passes the check in line 34, it accepts the new *est* value by record it locally to its *a_est* variable, and also records the *a_Rounds* value (same as the *p_Rounds* value of p) into its timestamp variable *a_TS* (line 36). Thus, another interpretation of *p_Rounds* and *a_Rounds* is that they are a kind of progressing times in the system. Variable *a_TS* records the time in this sense when acceptor q accepts the *est* value from a proposer, and these timestamp values are used for proposers on their preparation phases to select a value with the highest timestamp, as we already explained. With this time interpretation, our algorithm is closer to the original Paxos algorithm, whose timestamp is just a single round number.

After q accepts the value from p and records it locally, it sends an ACK-ACC message to p (line 37). When p collects a majority of ACK-ACC messages, it knows that its *est* value has been "locked" into the system, and it can decide on this value (line 24).

The following theorem summarizes the correctness of the algorithm, with the proof included in the appendix.

Theorem 2 *The algorithm in Figures 2 and 3 solves k-set agreement problem with any failure detector in Ω''_k.*

4.2 Features of the algorithm

The algorithm has a number of features that we now explain. First, the algorithm is *parameter-free*, that is, it does not have any information related to the parameter k. The fact that it solves k-set agreement is purely because it uses a failure detector in Ω''_k. If the algorithm is allowed to use parameter k, then it could be simplified such that (a) it does not need the *lbound* outputs of Ω''_k; (b) the variables *p_Rounds*, *a_Rounds*, and *a_TS* only keep the top k rounds; (c) the operator $\cup_n$ is replaced with $\cup_k$; (d) $\preceq_n$ is replaced with $\preceq_k$; and (e) *top*() is not needed in lines 8 and 30. However, parameter-free algorithms are more flexible. If in one run of the algorithm the failure detector in Ω''_k actually behaves like a failure detector in $\Omega''_{k'}$ with $k' < k$, our algorithm will let processes reach a better k'-set agreement instead of k-set agreement. This cannot be achieved if we hard-code k into the algorithm. Moreover, in [6] we extend this algorithm to work with partitioned failure detectors, and in that context the algorithm running in one partitioned component does not know the value of k for the set agreement it is solving. Therefore, a parameter-free algorithm is more generic, and it works with any failure detector in the entire family of $\{\Omega''_z\}_{1\leq z<n}$ to solve set agreement problems. The algorithm of [15] is also parameter-free, so our algorithm matches the flexibility of the algorithm in [15].

Second, and more importantly, the algorithm is a faithful extension of the original Paxos algorithm and inherits its efficiency, flexibility and robustness. Same as the Paxos algorithm, our algorithm has communication only between the leader proposers and the acceptors. In the normal cases when processes do not crash and the failure detector elect $\ell \leq k$ leaders correctly according to the specification of Ω''_k, each leader proposer spends $4n$ messages with the acceptors to reach a decision, so totally it takes $4\ell n$ messages to terminate the algorithm. The algorithm in [15] on the other hand requires communication between any pair of processes, so under the same normal cases, it takes $2n^2$ messages. Therefore, when $\ell < n/2$, our algorithm has better message complexity, and if $\ell << n$, the difference is $O(n)$ verses $O(n^2)$. Due to the exchange of *p_Rounds* and *a_Rounds*, our message size is $O(n)$. This is further reduced to $O(k)$ in Section 4.3. Therefore, our message size matches the algorithm in [15].

Also same as in the Paxos algorithm, when proposers need to execute multiple instances of k-set agreement, each leader proposer can batch multiple preparation phases and execute it once, even before it knows its own proposal for all instances. This is because the proposer does not need to know its own proposal until the beginning of its acceptance phase. As a result, for multiple instances of k-set agreement, our algorithm can further reduce time complexity to one round trip time in normal cases, which matches the Paxos algorithm and the algorithm in [15].

As for flexibility, our algorithm is easily adapted so that proposers and acceptors could be separate processes. Let n be the number of acceptors and m be the number of proposers. All we need to do is to make sure that failure detectors in Ω''_k are among the m proposers, and in line 9 n is replaced with m (or use other ways to generate unique and increasing round numbers among the proposers). Note that the acceptors in our algorithm do not query failure detectors. So we can have a fixed number of n acceptors passively responding to proposer messages and do not need to access failure detectors, while we have a flexible number of proposers with access to failure detectors to initiate k-set agreement process. Therefore, our algorithm matches the flexibility of the Paxos algorithm.

Finally, as in Paxos, our algorithm can also be made robust to transient failures of proposers and acceptors. As long as the proposers and the acceptors keep their key state variables $proposal$, *p_round*, *p_Rounds*, *taskid*, *a_Rounds*, *a_est*, *a_TS* in stable storage that survives transient failures, and proposers restart new rounds after the transient failures, our algorithm is still correct inspite of the loss of other state information such as messages received.

4.3 Improvement to the Algorithm

The algorithm in Figures 2 and 3 has one shortcoming on its message size, which is $O(n)$ because the *p_Rounds* and *a_Rounds* values included in the messages could be of size n. We now show how to change the algorithm such that the sets of rounds included in the messages have size of at most $maxb$, the maximum value of *lbound* in a run. This means the size is $O(k)$ since $maxb \leq k$.

The idea is to keep the local *p_Rounds* and *a_Rounds* values managed by the operator $\cup_n$, but when sending messages, use *top*() to truncate *p_Rounds* and *a_Rounds*. The parameter used in *top*() is the maximum *lbound* value a process sees so far. To ensure the correctness of the algorithm, we need to redefine some ordering among the acceptance phases. The rest are more details about the changes to the algorithm.

First, each process (proposers and acceptors) maintain a local variable b, which maintains the largest *lbound* value it sees so far. This is done by proposers periodically updating its b with its *lbound* value if *lbound* value is higher, and processes piggybacking their b values in the messages, and if the b value in a message is higher than the local b value, updating the local b value with the received b value. The simple property with variable b is that it is non-decreasing on every process, and eventually all processes stabilize to the same b value, which is at most k.

Second, whenever a proposer wants to send its *p_Rounds* value in a message, instead it sends $top(p\_Rounds, b)$. Similarly, whenever an acceptor wants to send its *a_Rounds* value in a message, it sends $top(a\_Rounds, b)$ instead.

Effectively, what we are doing is to use $(top(p\_Rounds, b), b)$ on proposers, and $(top(a\_Rounds, b), b)$ on acceptors as logical times to order the events, and we call them *working sets* in the proof. Processes exchanges their working sets in the messages, and use their working sets in all comparisons. The following are the changes to the algorithm for working set comparisons.

At line 17, we need to require that received (R, b) to be the same instead of just R, and also they need to be the same as $(top(p\_Rounds, b), b)$. At line 34, instead of checking $R \neq a\_Rounds$, check if (R, b) received is the same as $(top(a\_Rounds, b), b)$ maintained locally. Finally, the timestamp variable *a_TS* is now (R, b), and is set in line 36 to (R, b), which is the received value and is equal to the local $(top(a\_Rounds, b), b)$ value.

To compare timestamps, we define the following order. Let (R_1, b_1) and (R_2, b_2) be two working sets. We define $(R_1, b_1) \preceq (R_2, b_2)$ as $b_1 \leq b_2$ and $R_1 \preceq_{b_2} R_2$. Note that in general we cannot claim that $\preceq$ is a partial order because the transitivity property may not hold. For example, $(\{1\}, 1) \preceq (\{2\}, 1)$ and $(\{2\}, 1) \preceq (\{2, 4\}, 3)$, but we do not have $(\{1\}, 1) \preceq (\{2, 4\}, 3)$. However, the algorithm avoids such behavior by guaranteeing that the working sets of all acceptance phases (and all timestamp *a_TS* values) are totally ordered. The proof of the correctness of this improvement is included in the appendix.

5 Conclusion

In this paper, we study new failure detectors that are equivalent to Ω_k and study the extension of the Paxos algorithm to k-set agreement. The new failure detectors help us to understand various aspects that are related to Ω_k, while the extended Paxos algorithm provides us an efficient way to solve k-set agreement. It would be interesting to further this research to study how those Ω_k-like failure detectors can be implemented with weak synchrony requirements, and how to apply the efficient k-set agreement algorithms to solve distributed system problems that may have weaker consistency requirements than consensus and may be modeled in the k-set agreement context.

References

[1] E. Borowsky and E. Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *Proceedings of the 25th ACM Symposium on Theory of Computing*, pages 91–100. ACM Press, May 1993.

[2] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating System Design and Implementation*, Nov. 2006.

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

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

[5] S. Chaudhuri. More *choices* allow more *faults*: Set consensus problems in totally asyn-

chronous systems. *Information and Computation*, 105(1):132–158, July 1993.

[6] W. Chen, J. Zhang, Y. Chen, and X. Liu. Partition approach to failure detectors for k-set agreement. Technical Report MSR-TR-2007-49, Microsoft Research, May 2007.

[7] F. C. Chu. Reducing Omega to Diamond W. *Inf. Process. Lett.*, 67(6):289–293, 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, Apr. 1985.

[9] M. Herlihy and L. D. Penso. Tight bounds for k-set agreement with limited scope accuracy failure detectors. *Distributed Computing*, 18(2):157–166, 2005.

[10] M. Herlihy and N. Shavit. The topological structure of asynchronous computability. *Journal of the ACM*, 46(6):858–923, 1999.

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

[12] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):51–58, 2001.

[13] 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 Symposium on Operating System Design and Implementation*, San Francisco, CA, USA, Dec. 2004.

[14] A. Mostefaoui, S. Rajsbaum, and M. Raynal. The combined power of conditions and failure detectors to solve asynchronous set agreement. In *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing*, pages 179–188, July 2005.

[15] A. Mostefaoui, S. Rajsbaum, M. Raynal, and C. Travers. Irreducibility and additivity of set agreement-oriented failure detector classes. In *Proceedings of the 25th ACM Symposium on Principles of Distributed Computing*, pages 153–162, July 2006. Full version in technical report 1758, IRISA, 2005.

[16] A. Mostefaoui and M. Raynal. k-set agreement with limited accuracy failure detectors. In *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing*, pages 143–152, July 2000.

[17] G. Neiger. Failure detectors and the wait-free hierarchy. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, pages 100–109, Aug. 1995.

[18] M. Saks and F. Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. *SIAM Journal on Computing*, 29(5):1449–1483, 2000.

[19] J. Yang, G. Neiger, and E. Gafni. Structured derivations of consensus algorithms for failure detectors. In *Proceedings of the 17th ACM Symposium on Principles of Distributed Computing*, pages 297–306, June 1998.

Appendix

A Proofs for the Extended Paxos Algorithm

A.1 Properties of $\preceq_m$

Proposition 6 *Given two sets R_1 and R_2 containing round numbers and a positive integer m, $R_1 \preceq_m R_2$ if and only if one of the following condition holds: (a) $|R_2| < m$ and $R_1 \subseteq R_2$; or (b) $|R_2| = m$ and $\forall x \in R_1 \setminus R_2, \forall y \in R_2, x < y$.*

Proposition 7 *Given two sets R_1 and R_2 containing round numbers and a positive integer m, $R_1 \preceq_m R_2$ if and only if there exists a set R such that $R_1 \cup_m R = R_2$.*

Proposition 8 *Among all the sets of rounds with at most m elements, relation $\preceq_m$ is a partial order.*

A.2 Proof of the safety property

In this section we prove the correctness of the Agreement property (the key safety property) as stated below.

- *Uniform k-Agreement*: There are at most k different values decided by proposers (correct or not) in a run.

All statements in the following proof refer to a run of the algorithm with a failure pattern F and a failure detector history H. We start with some terminologies and notations used in the proof. When a proposer p is in its task 1, we say that it is in a *preparation phase* Φ^P if it is executing in lines 12–19; we say that it is in an *acceptance phase* Φ^A if it is executing in lines 20–24. Proposer p ends its preparation phase by either stopping its current task or entering its acceptance phase, and it ends its acceptance phase by either stopping its current task or deciding a value in line 24. We call Φ^A *successful acceptance phase* if it decides a value in line 24.

For any phase Φ (either a preparation phase or an acceptance phase) of a proposer p and any variable v of p, we denote $\Phi.v$ as the value of the variable v when p enters the phase (i.e., when p executes line 12 for the preparation phase or line 20 for the acceptance phase).

The algorithm guarantees the following properties.

Proposition 9 *Each acceptance phase Φ^A has a unique tuple (Φ^A.p_round, Φ^A.p_Rounds).*

Proof. For different proposers p_1 and p_2, let p_1 be in acceptance phase $\Phi^A_{p_1}$ and p_2 be in acceptance phase $\Phi^A_{p_2}$. Then, since $\Phi^A_{p_1}$.*p_round* $\equiv p_1 \pmod n$ and $\Phi^A_{p_2}$.*p_round* $\equiv p_2 \pmod n$ by Line 3 and Line 9, so $\Phi^A_{p_1}$.*p_round* $\neq$ $\Phi^A_{p_2}$.*p_round*.

For a single proposer p, if it decides at line 24, it will not start task 1 again, that is, it will not enter any other acceptance phases. Otherwise, it must have received a NACK-ACC message. According to line 34, we know that Φ^A_p.*p_Rounds* $\preceq_n R$ and Φ^A_p.*p_Rounds* $\neq R$. Thus, p's *p_Rounds* must increase after executing line 23. □

Proposition 10 *On any acceptor, the sequence of a_Rounds values ordered by time form a total order with respect to $\preceq_n$.*

Proof. This is guaranteed by the way *a_Rounds* variable is updated in lines 29 and 33, and by Proposition 7. □

Proposition 11 *Before executing line 16, the value of p_Rounds on a proposer is less than or equal to (with respect to $\preceq_n$) any R received in any* ACK-PREP *or* NACK-PREP *messages with the current taskid. If all the messages contain the same R, then p_Rounds $= R$ after line 16 is executed.*

Proof. On reception of a PREPARE message, every acceptor updates their *a_Rounds* varible at line 29, and then embeds the new *a_Rounds* value as R into the ACK-PREP or NACK-PREP message. By Proposition 7, we know *p_Rounds* $\preceq_n R$. Suppose the value of *p_Rounds* $= R_1$ before line 16, and *p_Rounds* $=$

R_2 after line 16. When all the messages contain the same R , we have $R_2 = R_1 \cup_n R$. According to line 29, there exists a R' such that $R = R' \cup_n R_1$, in which R' is the value of the *a_Rounds* variable on an acceptor. So $R_2 = R_1 \cup_n (R' \cup_n R_1) = R' \cup_n R_1 = R$. □

The above two propositions are the results of the $\cup_m$ operator. These two propositions are important to guarantee that the algorithm correctly operates on sets of rounds. Proposition 11 guarantees that if a proposer receives a majority of ACK-PREP messages and their R's are the same, then after executing line 16, the *p_Rounds* must be the same as the R value, which is the value of all *a_Rounds*. Then together with Proposition 10, it is guaranteed that all acceptance phases form a total order (Lemma 13), which is the key for the safety property. Moreover, Proposition 11 guarantees that if a proposer receives a majority of ACK-PREP messages but their R's are different, then after executing line 16, the *p_Rounds* must increase to a value higher than all received R's and the previous *p_Rounds*. This in turn guarantees that eventually the proposer will have the highest *p_Rounds* so that it make all acceptors have the same *a_Rounds* values. This is the key to the liveness property.

Proposition 12 *For any acceptance phase Φ^A, Φ^A.p_round $\in \Phi^A$.p_Rounds.*

Proof. Consider the preparation phase Φ^P preceding the acceptance phase Φ^A, for a proposer p. Since p enters the acceptance phase in the same task, we know that none of the conditions in line 17 is true. Therefore, all messages received are ACK-PREP with the same R value. According to line 30 on the acceptor thread, we know that Φ^A.*p_round* $\in R$. By Proposition 11, we have Φ^A.*p_Rounds* $= R$ and thus Φ^A.*p_round* $\in \Phi^A$.*p_Rounds*. □

Lemma 13 *For any two acceptance phases Φ_1^A and Φ_2^A, Φ_1^A.p_Rounds and Φ_2^A.p_Rounds are comparable.*

Proof. Suppose proposer p_1 enters Φ_1^A and proposer p_2 enters Φ_2^A. To enter an acceptance phase, both p_1 and p_2 must receive ACK-PREP from a majority of acceptors. So there must be an acceptor q in the intersection of the two majority sets.

According to the condition in line 17, when p_1 and p_2 executes line 19, all R's received are the same. Suppose p_1 receives R_1 and p_2 receives R_2. According to line 29, R_1 is the value of *a_Rounds* of q at time t_1 and R_2 is the one at time t_2. By Proposition 10, R_1 and R_2 are comparable. By proposition 11, Φ_1^A.*p_Rounds* $= R_1$ and Φ_2^A.*p_Rounds* $= R_2$. So Φ_1^A.*p_Rounds* and Φ_2^A.*p_Rounds* are comparable. □

A direct consequence is the following:

Corollary 14 *At line 19, all TS values form a total order.*

This is because *TS* is always assigned to a Φ^A.*p_Rounds* (line 36). Therefore, at line 19, it is always possible to find the highest *TS*.

Another result is that, we can arrange all acceptance phases into a sequence $\mathcal{AS} = \{\Phi_1^A, \Phi_2^A, \ldots\}$, s.t. $\forall i < j$, Φ_i^A.*p_Rounds* $\preceq_n \Phi_j^A$.*p_Rounds*. Consider the successful acceptance phase with minimum sequence number in $\mathcal{AS}$, namely Φ_u^A (u is the sequence number). We show that all other successful acceptance phase can only use the *est* value belonging to $\{\Phi_i^A.\textit{est} | \Phi_i^A.\textit{p\_Rounds} = \Phi_u^A.\textit{p\_Rounds}\}$.

Lemma 15 *For any acceptance phase Φ_i^A with $i > u$ and Φ_u^A.p_Rounds $\neq \Phi_i^A$.p_Rounds, there must exist Φ_j^A such that Φ_i^A.est $= \Phi_j^A$.est, Φ_j^A.p_Rounds $\preceq_n \Phi_i^A$.p_Rounds and Φ_u^A.p_Rounds $\preceq_n \Phi_j^A$.p_Rounds.*

Proof. Suppose it was proposer p which enters Φ_i^A. Consider the preparation phase Φ_i^P preceding Φ_i^A. Proposer p must receive (ACK-PREP, R, *TS*, v, *taskid*) messages from a majority of the acceptors in Φ_i^P. On the other hand, proposer p_u also receive ACK-ACC messages from a majority of the acceptors in Φ_u^A. Assume acceptor q is in the intersection of the two majority sets. So at time t_1 q sents ACK-ACC message to p_u for Φ_u^A, and at time t_2 q sents ACK-PREP message to p for Φ_i^A. Let R_1 be the value of *a_Rounds* of q at t_1 and R_2 be the value at t_2. According to line 34, $R_1 = \Phi_u^A.\mathit{p\_Rounds}$. By Proposition 11, $R_2 = \Phi_i^A.\mathit{p\_Rounds}$.

Since $\Phi_u^A.\mathit{p\_Rounds} \precneqq_n \Phi_i^A.\mathit{p\_Rounds}$, we have $t_1 < t_2$ by Proposition 10. Because Φ_u^A is a successful acceptance phase, $q.\mathit{a\_est} \neq \perp$ at t_1, thus $q.\mathit{a\_est} \neq \perp$ at t_2. Therefore, proposer p receives a $(\text{ACK-PREP}, \Phi_i^A.\mathit{p\_Rounds}, \mathit{TS}_1, v_1, \mathit{taskid})$ message from q with $v \neq \perp$ and $\mathit{TS} \succeq \Phi_u^A.\mathit{p\_Rounds}$. According to line 19, proposer p cannot use its own *proposal* value in phase Φ_i^A. Suppose $\Phi_i^A.\mathit{est}$ is taken from a value sent by an acceptor q'. And the v_2 in the $(\text{ACK-PREP}, \Phi_i^A.\mathit{p\_Rounds}, \mathit{TS}_2, v_2, \mathit{taskid})$ message sent by q' comes from an acceptance phase Φ_j^A. We know $\Phi_j^A.\mathit{p\_Rounds} = \mathit{TS}_2$. Because TS_2 is the *a_Rounds* value of q' some time before, $\Phi_j^A.\mathit{p\_Rounds} \preceq_n \Phi_i^A.\mathit{p\_Rounds}$. Since p selects v_2 as its *est*, it must be the case that $\mathit{TS}_1 \preceq_n \mathit{TS}_2$. Therefore, we have $\Phi_u^A.\mathit{Rounds} \preceq_n \mathit{TS}_1 \preceq_n \mathit{TS}_2 = \Phi_j^A.\mathit{p\_Rounds}$. □

Proposition 16 *Given any set R of round numbers, there are at most k different acceptance phases Φ^A with $\Phi^A.\mathit{p\_Rounds} = R$.*

Proof. By Proposition 9, different acceptance phases Φ^A with $\Phi^A.\mathit{p\_Rounds} = R$ must have different $\Phi^A.\mathit{p\_round}$. By Proposition 12, these *p_round* must be in R. According to the algorithm, R is the value of *a_Rounds* when acceptors execute line 31 and send to a proposer the ACK-PREP message. Because of the condition in line 30, $\mathit{p\_round} \in \mathit{top}(R, b)$, where b is the *lbound* value of some proposers. By $\Omega''1$, the *lbound* values never exceeds k. Therefore, there are at most k different acceptance phases Φ^A with $\Phi^A.\mathit{p\_Rounds} = R$. □

Now we can prove the following lemma for safety property.

Lemma 17 (Uniform k-Agreement) *There are at most k different values that have been decided by the algorithm.*

Proof. Let Φ_u^A be the first successful acceptance phase in sequence $\mathcal{AS}$. Let $R = \Phi_u^A.\mathit{p\_Rounds}$, and the decision value set $D = \{\Phi_j^A.\mathit{est} \mid \Phi_j^A.\mathit{p\_Rounds} = R\}$.

Now consider a proposer p that decides on acceptance phase Φ_i^A. If $\Phi_i^A.\mathit{p\_Rounds} = R$, we know the decision value belongs to D. Otherwise, $R \precneqq_n \Phi_i^A.\mathit{p\_Rounds}$ since Φ_u^A is the first successful acceptance phase. From Lemma 15, p uses a value from another round, namely $\Phi_{i(1)}^A$, where $R \preceq_m \Phi_{i(1)}^A.\mathit{p\_Rounds} \preceq_m \Phi_i^A.\mathit{p\_Rounds}$. Moreover, the event that a proposer enters acceptance phase $\Phi_{i(1)}^A$ must be causally before the events that proposer p enters acceptance phase Φ_i^A. If $\Phi_{i(1)}^A.\mathit{p\_Rounds} = R$, we know $\Phi_i^A.\mathit{est} = \Phi_{i(1)}^A.\mathit{est} \in D$. Othewise, we apply Lemma 15 on $\Phi_{i(1)}^A$ to find $\Phi_{i(2)}^A$. In this way, we can get a sequence of acceptance phases $\Phi_{i(1)}^A, \Phi_{i(2)}^A, \ldots$, such that $R \preceq_n \cdots \preceq_n \Phi_{i(2)}^A.\mathit{p\_Rounds} \preceq_n \Phi_{i(1)}^A.\mathit{p\_Rounds} \preceq_n \Phi_i^A.\mathit{p\_Rounds}$, and the events of a proposer entering $\Phi_{i(j+1)}^A$ are causally before the events of $\Phi_{i(j)}^A$. Since the number of acceptance phases causally before Φ_i^A is finite, the sequence cannot be infinitely long, and it should stop at an acceptance phase Φ_v^A with $\Phi_v^A.\mathit{p\_Rounds} = R$. Because all phases in the sequence have the same *est* value, we have $\Phi_i^A.\mathit{est} = \Phi_v^A.\mathit{est} \in D$. Therefore, the *est* values of all successful acceptance phases belong to the set D.

By Proposition 16, there are at most k different acceptance phase with $\mathit{p\_Rounds} = R$. By Proposition 9, each proposer only enters one acceptance phase once, and thus cannot decide more than one value in one acceptance phase. Therefore, there are at most k different values. □

A.3 Proof of the liveness property

We need to prove the following.

Lemma 18 (Termination) *Eventually some correct proposer decides.*

Proof. Assume that no correct proposer decides. Because a majority of the processes are correct, the proposers will not be stuck on the "wait-until" statements at line 13 and line 21.

Let L be the set of eventual correct leaders according to failure detector Ω''_k. Let b be the eventual stable value of *lbound*. Let t be the time after which both *isLeader* and *lbound* do not change on all processes. We know that $1 \le |L| \le b$ by Ω''_k' properties. According to the algorithm, proposers not in L will not be running task 1 after a time $t_0 > t$.

When a proposer p starts task 1, the task could be stopped without deciding because of one of the following conditions: 1) p receives a NACK-PREP message in prepare phase; 2) p does not receive any NACK-PREP messages in prepare phase, but some Rs in the messages are different; 3) p receives a NACK-ACC message in the acceptance phase. For case 1, after t we have $p.p\_round \notin top(q.a\_Rounds, b)$ on some acceptor q according to line 30. When p starts task 1 next time, since $p\_round \notin top(p\_Rounds, b)$, *p_round* will be increased. For case 2, by Proposition 11, its *p_Rounds* increases after it executes line 16. For case 3, either its *p_round* changes or its *p_Rounds* changes (by Proposition 11). Therefore, after time t, every time proposer p starts task 1, either its *p_round* or its *p_Rounds* increases. If its *p_round* stops changing at some time, its *p_Rounds* should also stops changing. Otherwise, *p_Rounds* has to increase w.r.t $\preceq_n$. After a finite number of times, *p_round* will be out of $top(p\_Rounds, b)$. Therefore, if p cannot decide, its *p_round* has to change infinitely often.

After t_0 proposers not in L stop increasing their *p_round*. So there must exists a time t_1 at and after which $p_1.p\_round > p_2.p\_round$ for all $p_1 \in L$ and $p_2 \in P \setminus L$. If p_2 crashes, we use its *p_round* value just before it crashes. Suppose p has the largest *p_round* value at t_1 among all the proposers. Its *p_round* value will stay in $top(p.p\_Rounds, b)$ from now on. This is because our algorithm ensures that a new *p_round* is chosen only when $p'.p\_round$ does not in $top(p'.p\_Rounds, b)$. So for all $p' \in L \setminus \{p\}$, p' only needs to pick the smallest *p_round* value larger than $p.p\_round$ to make $p'.p\_round \in top(p'.p\_Rounds, b)$. Since $|L \setminus \{p\}| \le b - 1$, there are at most $b - 1$ round numbers larger than $p.p\_round$. Therefore, $p.p\_round$ stops change after t_1. This is contradictory. □

Theorem 2 *The algorithm in Figures 2 and 3 solves k-set agreement problem with any failure detector in Ω''_k.*

Proof. For every acceptance phase Φ^A, $\Phi^A.est$ either comes from the process's proposal or from another acceptance phase. Therefore, the validity property is ensured. Together with Lemma 17 and Lemma 18, our protocol k-set agreement problem. □

B Changes to the Proof for the Improvement

First, we need to define the working set. At any point in time, the working set on a proposer is its current value of $(top(p\_Rounds, b), b)$; the working set on an acceptor is its current $(top(a\_Rounds, b), b)$ value. Recall that b is the local variable which maintains the largest *lbound* value the proposer sees so far. By the algorithm, we still have $\Phi^A.p\_round \in \Phi^A.working\_set$ (of course, "$\in$" here means in the set part of the working set). We use notations $\Phi^A.set$ and $\Phi^A.bound$ to represent the $top(p\_Rounds, b)$ and b parts of the working set of Φ^A. Clearly, we have $|\Phi^A.set| \le \Phi^A.bound$. With the above definition, we can also say that the proposers exchange their working sets in the messages. The working set is now becoming the timestamp to order the events.

The order between working sets have been defined in Section 4.3. We need the following properties for the correctness of the algorithm.

Proposition 19 *Given two sets of rounds R_1 and R_2 and two positive integers m and b such that $1 \le b \le m$ and $R_1 \preceq_m R_2$, for any subset $R_3 \subseteq R_1$ such that $|R_3| \le b$, we have $R_3 \preceq_b top(R_2, b)$.*

Proof. If $|R_2| < b$, then $|R_2| < m$ and $R_1 \subseteq R_2$. Therefore, $R_3 \subseteq R_1 \subseteq R_2$, which implies $R_3 \preceq_b R_2 = top(R_2, b)$. If $|R_2| \ge b$, then $|top(R_2, b)| = b$. If $R_3 \subseteq top(R_2, b)$, then it is fine. Otherwise, for any $x \in R_3 \setminus top(R_2, b)$, we have $x \in R_1 \setminus top(R_2, b)$. If $x \in R_1 \setminus R_2$, then $x < \min R_2$ since $R_1 \preceq_m R_2$, and thus $x < \min top(R_2, b)$. If $x \in R_2 \setminus top(R_2, b)$, we also have $x < \min top(R_2, b)$. Therefore, $R_3 \preceq_b top(R_2, b)$. □

Proposition 20 *Let m be a positive integer. Suppose $A_0 = \emptyset$. Suppose that for any $i = 1, 2, \ldots$ (a) positive integers b_i such that $b_i \leq b_{i+1}$ and $b_i \leq m$; (b) sets of rounds R_i such that $|R_i| \leq b_i$; and (c) sets of rounds A_i such that $A_i = A_{i-1} \cup_n R_i$ ($n \geq m$), then we have for any i and j such that $1 \leq i < j$, $(top(A_i, b_i), b_i) \preceq (top(A_j, b_j), b_j)$.*

Proof. We already know that $b_i \leq b_j$, so we only need to show that $top(A_i, b_i) \preceq_{b_j} top(A_j, b_j)$. By the property of $\preceq_n$ and $\cup_n$ (Propositions 7 and 8), we know that $A_i \preceq_n A_j$. Since $b_i \leq b_j$, we have $top(A_i, b_i) \preceq_{b_j} top(A_j, b_j)$. directly from Proposition 19. □

Since $n \geq k$, the variables *a_Rounds* and b on an acceptor evolve exactly like A_i's and b_i's in the above proposition. So we have

Corollary 21 *The working sets on an acceptor increase and form a total order. (Compare to Proposition 10)*

Lemma 22 *The working sets of all acceptance phases form a total order, and thus all timestamp a_TS values form a total order.*

Proof. This is because the working set of an acceptance phase must be the same as the working set of some acceptors (due to the condition in line 17 with the modifications), and two working sets of two acceptance phases must be two working sets on a single acceptor (due to the majority requirement on ACK-PREP responses). As for *a_TS* value, it is always the same as the working set of some acceptance phase (due to condition in line 34 with the modification specified above). □

The above lemma is the key one to ensure that the new algorithm still satisfies the safety property. The rest of the proof of the Uniform k-Agreement property is the same as the proof of Lemma 17, except we need to use our current definition of the working set instead of *p_Rounds*.

The following properties are to ensure the liveness property.

Proposition 23 *Let R_1, R_2 be two sets of rounds and b_1 and b_2 are two positive integers such that $|R_1|, |R_2|, b_1, b_2 \leq m$. Let $R_3 = R_2 \cup_n top(R_1, b_1)$ ($n \geq m$) and $b_3 = \max(b_1, b_2)$. Then we have $(top(R_1, b_1), b_1) \preceq (top(R_3, b_3), b_3)$.*

Proof. Since $R_3 = R_2 \cup_n top(R_1, b_1)$, we have $top(R_1, b_1) \preceq_n R_3$. Then $(top(R_1, b_1), b_1) \preceq (top(R_3, b_3), b_3)$ is a direct result of Proposition 19. □

Corollary 24 *Let (R_1, b_1) be the working set that an acceptor sends to a proposer (in an ACK-PREP, NACK-PREP, or NACK-ACC message). Let R_2 and b_2 be the values of variables p_Rounds and b after a proposer receives the message but before updating its local p_Rounds and b variable. Then $(top(R_2, b_2), b_2) \preceq (R_1, b_1)$. Let $R_3 = R_2 \cup_n R_1$ and $b_3 = \max(b_2, b_1)$. Then $(R_1, b_1) \preceq (top(R_3, b_3), b_3)$. Moreover, if $(R_1, b_1) \neq (top(R_3, b_3), b_3)$, then $(top(R_2, b_2), b_2) \prec (top(R_3, b_3), b_3)$ (here $\prec$ means $\preceq$ and $\neq$).*

Proposition 25 *If no proposer decides, then eventually every time a proposer starts task 1, either its working set $(top(p\_Rounds, b), b)$ increases, or its p_round increases.*

Proof. There are three reasons to stop task 1 and restart it later: (a) it receives a NACK-PREP message; (b) it receives a majority of ACK-PREP but some of them are different or they are different to $top(p\_Rounds, b)$ after the proposer updates its local variable; and (c) it receives a NACK-ACC message. For case (a), eventually *lbound* stabilizes, so an NACK-PREP message means $p\_round \notin top(p\_Rounds, lbound)$, and thus the proposer will increase *p_round*. For case (b) and (c), by Corollary 24, *p_Rounds* strictly increases. □

With the above proposition, the rest of liveness proof should be the same as the proof of Lemma 18.