Failure Detectors and Extended Paxos for k-Set Agreement

Wei Chen
Microsoft Research Asia
weic@microsoft.com

Jialin Zhang*
Tsinghua University
zhanggl02@mails.tsinghua.edu.cn

Yu Chen
Microsoft Research Asia
ychen@microsoft.com

Xuezheng Liu
Microsoft Research Asia
xueliu@microsoft.com

Abstract

Failure detector class Ω_k has been defined in [18] as an extension to failure detector Ω, and an algorithm has been given in [16] to solve k-set agreement using Ω_k in asynchronous message-passing systems. In this paper, we extend these previous work 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. 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 [12], and thus it inherits the efficiency, flexibility, and robustness of the Paxos algorithm. In particular, it has better message complexity than the algorithm in [16]. Both the new failure detectors and the new algorithm enrich our understanding of the k-set agreement problem.

1 Introduction

Failure detectors are introduced in [4] to circumvent the impossibility result of solving asynchronous consensus [9]. 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 $\diamond\mathcal{S}$ [8], 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.

In [12], Lamport designed the Paxos algorithm that also solves consensus in systems with a majority of correct processes. 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. [14, 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, 11, 19], and a number of studies have introduced various failure detectors to circumvent this impossibility result [20, 17, 10, 15, 16]. In [16], 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 [18] for studying wait-free hierarchy in shared memory systems. In [16] an algorithm

*This work was supported in part by the National Natural Science Foundation of China Grant 60553001, and the National Basic Research Program of China Grant 2007CB807900,2007CB807901.

[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.

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 [7], 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 [7].

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 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.

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 [7]. The algorithm in [16] is also parameter-free, so we match its generality in this sense.

Second, and more importantly, 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 [16]. Moreover, same as the Paxos algorithm, our algorithm allows the efficient 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 [16]. 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 be made to tolerate transient failures 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.

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 full technical report [6] contains further details including the correctness proof and the improvement to the algorithm.

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. A failed process does not recover. 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, $\forall t \in \mathcal{T}, \forall p \notin F(t), |H(p,t)| \le k$.

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

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

3.1 Specification of Ω'_k

As described in the introduction, for Ω'_k, we aim at failure detectors in which each process only selects 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 $(\mathit{leader}, \mathit{lbound})$, 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 $H(p,t).\mathit{leader}$ and $H(p,t).\mathit{lbound}$ 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:

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

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

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

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

Several remarks are in order for the above definition. First, one may see that properties (Ω'1) and (Ω'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 (Ω1), 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 (Ω'1), property (Ω'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.

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, $\forall t \in \mathcal{T}, \forall p \notin F(t), H(p,t).\mathit{lbound} \leq k$.

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

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

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

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

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.

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, p sends 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 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 that each process p_i maintains an array $c[\,]$, in which $c[p_j]$ counts the number of times for which p_i sees p_j as a leader (line 6). It periodically sends its $c[\,]$ to all processes (line 7), and merges $c[\,]$ with the ones received from other processes (line 11). Then p_i sorts all processes into an array $A[1..n]$ based on the counter values ($c[A[1]]$ has the highest counter value), and uses process ID to break the tie (line 8). The *Leaders* output of Ω_k is the first *lbound* elements of $A[\,]$ (line 9). This transformation bears some resemblance to the transformation from $\diamond\mathcal{W}$ to Ω in [8].

Lemma 3 *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 9, $|\mathit{Leaders}|$ is bounded by *lbound* value. By property $(\Omega'1)$, we thus see that $|\mathit{Leaders}|$ is at most k at all times. Thus $(\Omega1)$ holds.

```
On node $p_i$:
Global variables:
    $(leader, lbound)$: output of $\Omega'_k$, read-only
    $Leaders$: output of $\Omega_k$, initially $\{p_i\}$
    $c[p_1..p_n]$: counters for all processes, initially 0
Repeat periodically:
    $p \leftarrow leader$; $c[p] \leftarrow c[p] + 1$
    for each $p_j \in P$ do send $c[p_1..p_n]$ to $p_j$
    $A[1..n] \leftarrow$ permutation of $(p_1, \dots, p_n)$, such that
        for all $1 \le x < y \le n$, $(c[A[x]], A[x]) > (c[A[y]], A[y])$
    $Leaders \leftarrow A[1..lbound]$
Upon receipt of $c_j[p_1..p_n]$ from a node $p_j$:
    for each $p_i \in P$ do $c[p_i] \leftarrow \max\{c[p_i], c_j[p_i]\}$
```

Figure 1. Transformation from Ω'_k to Ω_k.

By property $(\Omega'2)$, *lbound* output of all processes are stable, let this value be lb. Let L be the set of processes that appear infinitely often in the *leader* output of line 6 in the fixed run. Let $\ell = |L|$, then by property $(\Omega'4)$, $1 \le \ell \le lb \le k$. For process $p \in L$, $c[p]$ increases infinitely often in the algorithm. For process $p \notin L$, $c[p]$ eventually stops increasing. So eventually $L \subseteq$ *Leaders* for all correct processes. By property $(\Omega'3)$, all processes in L are correct processes. Thus $(\Omega 3)$ holds.

For process $p \notin L$, by line 11, $c[p]$ of all correct processes are eventually the same, so the other $lb - \ell$ processes in *Leaders* of all correct processes are the same. Thus $(\Omega 2)$ 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 3, we can now state the following theorem.

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

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 referring to other processes in the system.

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. We would lose this flexibility if we were to replace *lbound* with a fixed value k.

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 [12] for solving consensus.

4.1 Algorithm 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 [13]. 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

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

Proposer variables:
1  $proposal$: the initial proposal value, read-only
2  ($isLeader$, $lbound$): $\Omega''_k$ 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      $taskid \leftarrow taskid + 1$;
8      if $p\_round \notin top(p\_Rounds, lbound)$ then
9          $p\_round \leftarrow p\_round + t \cdot n$
               such that $p\_round + t \cdot n > \max p\_Rounds$
10         $p\_Rounds \leftarrow p\_Rounds \cup_n \{p\_round\}$
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 $M_1 \leftarrow \{$(ACK-PREP, $R$, $TS$, $v$, $taskid$) received
       from acceptors$\}$
15 $M_2 \leftarrow \{$(NACK-PREP, $R$, $taskid$) received from acceptors$\}$
16 $p\_Rounds \leftarrow p\_Rounds \cup_n (\bigcup_{m \in M_1 \cup M_2} m.R)$
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 $est \leftarrow proposal$
19 else $est \leftarrow m.v$ 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     $p\_Rounds \leftarrow p\_Rounds \cup_n R$; stop this task
24 decide($est$)
```

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

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 highest 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 $R_1 \cup_m R_2 = R_2$.

```
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     $a\_Rounds \leftarrow a\_Rounds \cup_n R$
30     if $r \notin top(a\_Rounds, lb)$ then
31         send (NACK-PREP, $a\_Rounds$, $taskid$) to $p$
32     else send (ACK-PREP, $a\_Rounds$, $a\_TS$, $a\_est$, $taskid$) to $p$

33 Upon receipt of (ACCEPT, $v$, $R$, $taskid$) from $p$
34     $a\_Rounds \leftarrow a\_Rounds \cup_n R$
35     if $R \neq a\_Rounds$ then
36         send (NACK-ACC, $a\_Rounds$, $taskid$) to $p$
37     else
38         $(a\_est, a\_TS) \leftarrow (v, R)$
39         send (ACK-ACC, $taskid$) to $p$
```

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

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, 34). 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 is 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.

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 34), 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 36). 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 35, 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 38). 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 39). 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.

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 [7] 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 [16] is also parameter-free, so our algorithm matches the flexibility of the algorithm in [16].

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 [16] 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 [6]. Therefore, our message size matches the algorithm in [16].

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 [16].

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. 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 in spite of the loss of other state information such as messages received.

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

[6] W. Chen, J. Zhang, Y. Chen, and X. Liu. Failure detectors and extended Paxos for k-set agreement. Technical Report MSR-TR-2007-48, Microsoft Research, May 2007.

[7] W. Chen, J. Zhang, Y. Chen, and X. Liu. Weakening failure detectors for k-set agreement via the partition approach. In *Proceedings of the 21st International Symposium on Distributed Computing*, Sept. 2007.

[8] F. C. Chu. Reducing Omega to Diamond W. *Inf. Process. Lett.*, 67(6):289–293, 1998.

[9] 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.

[10] 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.

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

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

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

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.