How to solve consensus in the smallest window of synchrony

Dan Alistarh[1], Seth Gilbert[1], Rachid Guerraoui[1], and Corentin Travers[2]

[1] EPFL LPD, Bat INR 310, Station 14, 1015 Lausanne, Switzerland
[2] Universidad Politecnica de Madrid, 28031 Madrid, Spain

Abstract. This paper addresses the following question: what is the minimum-sized synchronous window needed to solve consensus in an otherwise asynchronous system? In answer to this question, we present the first optimally-resilient algorithm *ASAP* that solves consensus *as soon as possible* in an eventually synchronous system, i.e., a system that from some time *GST* onwards, delivers messages in a timely fashion. *ASAP* guarantees that, in an execution with at most f failures, every process decides no later than round $GST + f + 2$, which is optimal.

1 Introduction

The problem of *consensus*, first introduced in 1980 [22,25], is defined as follows:

Definition 1 (Consensus). *Given n processes, at most t of which may crash: each process p_i begins with initial value v_i and can* decide *on an output satisfying: (1) Agreement: every process decides the same value; (2) Validity: if a process decides v, then v is some process's initial value; (3) Termination: every correct process eventually decides.*

In a seminal paper [10], Dwork et al. introduce the idea of *eventual synchrony* in order to cirumvent the asynchronous impossibility of consensus [11]. They study an asynchronous system in which, after some unknown time GST (*global stabilization time*), messages are delivered within a bounded time. They show that consensus can be solved in this case if and only if $n \geq 2t + 1$.

Protocols designed for the eventually synchronous model are appealing as they tolerate arbitrary periods of asynchrony: in this sense, they are "indulgent" [13]. Such protocols are particularly suited to existing distributed systems, which are indeed synchronous most of the time, but might sometimes experience periods of asynchrony. In practice, the system need not be permanently synchronous after GST; it is necessary only that there be a sufficienly big *window of synchrony* for consensus to complete.

This leads to the following natural question: *For how long does the system need to be synchronous to solve consensus?* In other words, how fast can processes decide in an eventually synchronous system after the network stabilizes? The algorithm presented in [10] guarantees that every process decides within $4(n+1)$ rounds of GST, i.e., the required window of synchrony is of length $4(n + 1)$. On

the other hand, in [7], Dutta and Guerraoui show that, in the worst case, at least $t + 2$ synchronous rounds of communication are needed. They also present an algorithm for $t < n/3$ that matches this lower bound, but they leave open the question of whether there is an optimally resilient algorithm that decides in any synchronous window of size $t + 2$. In this paper, we once and for all resolve this question by demonstrating a consensus algorithm that guarantees a decision within $t + 2$ rounds of GST.

Early decision. Even though, in the worst case, at least $t + 2$ synchronous rounds are needed to solve consensus, in some executions it it is possible to decide faster. Lamport and Fisher [21] showed that, in a synchronous system, if an execution has at most $f \leq t$ failures, it is possible to decide in $f + 2$ rounds. Dolev, Reischuk, and Strong [5] showed that this bound was optimal. It has remained an open question as to whether there is an optimally resilient early deciding protocol for eventually synchronous systems.

Intuitively, eventual synchrony requires one additional round: $t + 1$ synchronous rounds to compute the decision, and one additional round to determine that the execution was, in fact, synchronous. Similarly, early-deciding algorithms require one additional round: $f + 1$ synchronous rounds to compute the decision, and one round to determine that there were only f failures. Thus, the question at hand is whether these rounds can be merged: can we verify in *just one round* both that the execution was synchronous and that there were only f failures? The algorithm presented in this paper achieves exactly that feat, terminating within $f + 2$ rounds after GST in an execution with at most f failures.

Results. In this paper, we present the *ASAP* algorithm which solves consensus and ensures the following properties: (1) *Optimal resilience*: *ASAP* can tolerate up to $t < n/2$ crash failures; notice that no consensus algorithm can tolerate $\geq n/2$ failures in an eventually synchronous system. (2) *Early deciding*: in every execution with at most $f \leq t$ failures, every process decides no later than round $GST + f + 2$; again, notice that this is optimal.

Key ideas. The *ASAP* algorithm consists of three main mechanisms. The first mechanism is responsible for computing a value that is safe to decide; specifically, each process maintains an *estimate*, which is updated in every round based on the messages it receives. The second mechanism is responsible for detecting asynchrony; processes maintain (and share) a log of active and failed processes which helps to discover when asynchronies have occurred. Finally, the third mechanism is responsible for determining when it is safe to decide; specifically, a process decides when it has: (1) observed $\leq f$ failures for some $f \leq t$; (2) observed at least $f + 2$ consecutive synchronous rounds; and (3) observed at least two consecutive rounds in which no process appears to have failed. The *ASAP* algorithm combines these mechanisms within a *full information* protocol, meaning that in each round, each process sends its entire state to every other process. (Optimizing the message complexity is out of the scope of this paper.)

Perhaps the key innovation in the *ASAP* algorithm is the mechanism by which a process updates its estimate of the decision value. We begin with the naïve approach (as in [24]) in which each process adopts the minimum estimate

received in each round. In a synchronous execution with at most $f \leq t$ failures, this guarantees that every process has the same estimate no later than round $f + 1$. We augment this simple approach (generalizing on [7]) by prioritizing an estimate from a process that is about to decide. Moreover, we break ties among processes about to decide by giving priority to processes that have observed more consecutive synchronous rounds. This helps to ensure that if a process does, in fact, decide, then every process has adopted its estimate. This same prioritization scheme, however, poses a problem when a process that has been given priority (since it is about to decide), finally does *not* decide (due to a newly detected asynchrony). To resolve this issue, we sometimes *waive the priority* on an estimate: when a process p_i receives an estimate from another process p_j that is about to decide, p_i examines the messages it has received to determine whether or not p_j (or any process that has received p_j's message) can decide. If p_i can *prove* that process p_j does not decide, then p_i can treat the estimate from process p_j with normal priority. Otherwise, if p_i cannot be certain as to whether p_j will or will not decide, p_i makes the conservative decision and prioritizes the estimate from p_j. This notion of selective prioritization is at the heart of our *ASAP* algorithm, and may be of use in other contexts, such as k-set agreement and Byzantine agreement.

2 Related Work

Beginning with Dwork et al. [10], a variety of different models have been used to express eventual synchrony, including failure detectors [3, 4] and round-by-round failure detectors (RRFD) [12]. These approaches have led to the concept of indulgent algorithms [7, 13, 14]—algorithms that tolerate unreliable failure detectors, expressed in the RRFD model. More recently, Keidar and Shraer [17, 18] introduced GIRAF, a framework that extends the assumptions of RRFD.

An important line of research has approached the question we address in this paper in a different manner, asking how fast consensus can terminate if there are *no further failures* after the system stabilizes. Keidar, Guerraoui and Dutta [8] show that at least 3 rounds are needed after the system stabilizes and failures cease, and they present a matching algorithm[3]. Two further papers [17, 18] also investigate the performance of consensus algorithms under relaxed timeliness and failure detector assumptions after stabilization.

Paxos-like algorithms that depend on a leader form another class of algorithms in this line of research. Work in [19, 23] and [1, 2] minimizes the number of "stable" synchronous communication rounds after a correct leader is elected

[3] It may appear surprising that we can decide within $f+2$ rounds of GST, as [8] shows that it is impossible to decide sooner than three rounds after failures cease. Indeed, a typical adversarial scenario might involve failing one processor per round during the interval $[GST+1, GST+f]$, resulting in a decision within two rounds of failures ceasing. However, this is not a contradiction as these are worst-case executions in which our algorithm does not decide until 3 rounds after failure cease.

that are needed to reach agreement, matching lower bounds in [20] and [16], respectively. A related algorithm is presented in [9], which guarantees termination within 17 message delays after stabilization, for the case where no failures occur after stabilization. In fact, it is conjectured there that a bound of $O(f)$ rounds is possible in the case where f failures occur after stabilization. Our paper resolves this conjecture in the affirmative.

Note that our approach to network stabilization differs from both of these previous approaches in that it focuses only on the behavior of the network, independent of failures or leader election.

Finally, Guerraoui and Dutta [6, 7] have investigated the possibility of early-deciding consensus for eventual synchrony and have obtained a tight lower bound of $f + 2$ rounds for executions with $f \leq t$ failures, even if the system is initially synchronous. They also present an algorithm for the special case where $t < n/3$ (not optimally resilient) that solves consensus in executions with at most f failures within $f + 2$ rounds of GST, leaving open the question of an optimally resilient consensus algorithm, which we address in this paper.

3 Model

We consider n deterministic processes $\Pi = \{p_1, \ldots, p_n\}$, of which up to $t < n/2$ may fail by crashing. The processes communicate via an *eventually synchronous* message-passing network, modeled much as in [7, 10, 17]: time is divided into *rounds*; however, there is no assumption that every message broadcast in a round is also delivered in that round. Instead, we assume only that if all non-failed processes broadcast a message in some round r, then each process receives at least $n - t$ messages in that round [4]. We assume that the network is *eventually synchronous*: there is some round GST after which every message sent by a non-failed process is delivered in the round in which it is sent.

4 The *ASAP* Consensus Algorithm

In this section, we present an optimally-resilient early-deciding consensus algorithm for the eventually-synchronous model that tolerates $t < n/2$ failures and terminates within $f + 2$ rounds of GST, where $f \leq t$ is the actual number of failures. The pseudocode for $ASAP$ can be found in Figures 1 and 2.

4.1 High-Level Overview

The $ASAP$ algorithm builds on the idea of estimate flooding from the classical synchronous "FloodSet" algorithm (e.g., [24]) and on the idea of detecting asynchronous behavior introduced by the "indulgent" A_{t+2} algorithm of [7].

[4] A simple way to implement this would be for each node to delay its round $r + 1$ message until at least $n - t$ round r messages have been received, and ignoring messages from previous rounds; however, this affects the early-deciding properties of the algorithm, as a correct process can be delayed by asynchronous rounds in which it does not receive $n - t$ messages.

Each process maintains an estimate, along with other state, including: for each round, a set of (seemingly) active processes and a set of (seemingly) failed processes; a flag indicating whether the process is ready to decide; and an indicator for each round as to whether it appears synchronous. At the beginning of each round, processes send their entire state to every other process; *ASAP* is a *full-information protocol*. Processes then update their state and try to decide, before continuing to the next round. We briefly discuss the three main components of the algorithm:

Asynchrony Detection. Processes detect asynchrony by analyzing the messages received in preceeding rounds. Round r is marked as asynchronous by a process p if p learns that a process q is alive in a round $r' > r$, even though it believes [5] q to have failed in round r. Notice that a process p may learn that process q is still alive either directly—by receiving a message from q—or indirectly—by receiving a message from a third process that believes q to be alive. The same holds for determining which processes have failed. Thus, a process merges its view with the views of all processes from which it has received messages in a round, maximizing the amount of information used for detecting asynchrony.

Decision. A process can decide only when it is certain that every other process has adopted the same estimate. There are two steps associated with coming to a decision. If a process has observed f failures, and the previous $f + 1$ rounds are perceived as synchronous, then it sets a "ready to decide" flag to *true*. A process can decide in the following round under the following circumstances: (i) it has observed f failures; (ii) the last $f + 2$ rounds appear synchronous; and (iii) there are no new failures observed in the last two rounds. Once a process decides, it continues to participate, informing other processes of the decision.

Updating the Estimate. The procedure for updating the estimate is the key to the algorithm. Consider first the simple rule used by the classic synchronous consensus protocol, where each process adopts the minimum estimate received in every round. This fails in the context of eventual synchrony since a "slow" process may maintain the minimum estimate even though, due to network delays, it is unable to send or receive messages; this slow process can disrupt later decisions and even cause a decision that violates safety. A natural improvement, which generalizes the approach used in [7], is to prioritize the estimate of a process that is about to decide. Notice that if a process is about to decide, then it believes that it has seen at least one failure-free synchronous round, and hence its estimate should be the minimum estimate in the system. However, this too fails, as there are situations where a process has a synchronous view of $f + 1$ rounds with f failures without necessarily holding the smallest estimate in the system. Thus, we award higher priority to messages from processes that are ready to decide, but allow processes to de-prioritize such estimates if they can *prove* that no process decides after receiving that estimate in the current round.

It remains to describe how a process p can *prove* that no process decides upon receiving q's message. Consider some process s that decides upon receiving

[5] Note that, throughout this paper, we use terms like "knowledge" and "belief" in their colloquial sense, not in the knowledge-theoretical sense of [15].

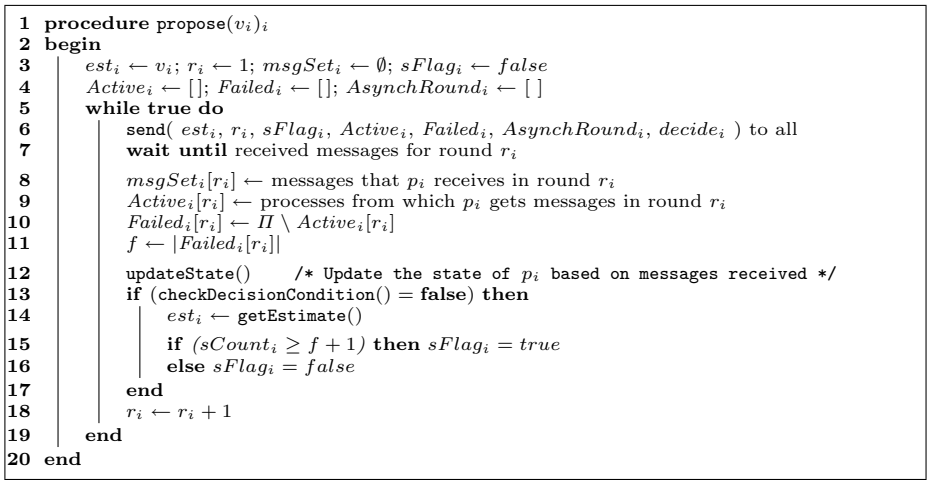

Fig. 1. The *ASAP* algorithm, at process p_i.

q's message. If p can identify a process that is believed by q to be alive and which *does not support* the decision being announced by q, then p can be certain that s will not decide: either s receives a message from the non-supporting process and cannot decide, or s does not receive its message and thus observes a new failure, which prevents s from deciding. Thus, a sufficient condition for discarding q's flag is the existence of a third process that: (i) q considers to be alive in the previous round, and (ii) receives a set of messages other than q's in $r-1$ (Proposition 9). Although this condition does not ensure that p discards all flags that do not lead to decision, it is enough for *ASAP* to guarantee agreement.

4.2 Detailed Description

We now describe the pseudocode in Figures 1 and 2. When consensus is initiated, each process invokes procedure propose() (see Figure 1) with its initial value. A decision is reached at process p_i when $decide_i$ is first set to *true*; the decision is stored in est_i. (For simplicity, the algorithm does not terminate after a decision; in reality, only one further round is needed.)

State Variables. A process p_i maintains the following state variables: (a) r_i is the current round number, initially 1. (b) est_i is p_i's estimate at the end of round r_i. (c) $Active_i[\,]$ is an array of sets of processes. For each round $r' \leq r_i$, $Active_i[r']$ contains the processes that p_i believes to have sent at least one message in round r'. (d) $Failed_i[\,]$ is an array of sets of processes. For each round $r' \leq r_i$, $Failed_i[r']$ contains the processes that p_i believes to have failed in round r'. (e) $msgSet_i$ is the set of messages that p_i receives in round r_i. (f) $AsynchRound_i[\,]$ is an array of flags (booleans). For each round $r' \leq r_i$, $AsynchRound_i[r'] = true$ means that r' is seen as asynchronous in p_i's view at round r_i. (g) $sCount_i$ is an integer

denoting the number of consecutive synchronous rounds p_i sees at the end of r_i. More precisely, if $sCount_i = x$, then rounds in the interval $[r_i - x + 1, r_i]$ are seen as synchronous by p_i at the end of round r_i. (h) $sFlag_i$ is a flag that is set to *true* if p_i is *ready to decide* in the next round. (i) $decided_i$ is a flag that is set to *true* if process p_i has decided.

Main algorithm. We now describe *ASAP* in more detail. We begin by outlining the structure of each round (lines 5-18, Figure 1). Each round begins when p_i broadcasts its current estimate, together with its other state, to every process (line 6); it then receives messages for round r_i (line 7). Process p_i stores these messages in $msgSet_i$ (line 8), and updates $Active_i[r_i]$ and $Failed_i[r_i]$ (lines 9–11).

Next, p_i calls the updateState() procedure (line 12), which merges the newly received information into the current state. It also updates the designation of which rounds appear synchronous. At this point, checkDecisionCondition is called (line 13) to see if a decision is possible. If so, then the round is complete. Otherwise, it continues to update the estimate (line 14), and to update its $sFlag_i$ (line 15–16). Finally, process p_i updates the round counter (line 18), and proceeds to the next round.

Procedure updateState(). The goal of the updateState() procedure is to merge the information received during the round into the existing *Active* and *Failed* sets, as well as updating the *AsynchRound* flag for each round. More specifically, for every message received by process p_i from some process p_j, for every round $r' < r_i$: process p_i merges the received set $msg_j.Active_j[r']$ with its current set $Active_i[r']$. The same procedure is carried out for the *Failed* sets. (See lines 3-8 of *updateState*(), Figure 2).

The second part of the *updateState* procedure updates the *AsynchRound* flag for each round. For all rounds $r' \leq r_i$, p_i recalculates $AsynchRound_i[r']$, marking whether r' is asynchronous in its view at round r_i (lines 9-14). Notice that a round r is seen as asynchronous if some process in $Failed_i[r]$ is discovered to also exist in the set $Active_i[k]$ for some $k > r$, i.e., the process did not actually fail in round r, as previously suspected. Finally, p_i updates $sCount_i$, with the number of previous consecutive rounds that p_i sees as synchronous (line 15).

Procedure checkDecisionCondition(). There are two conditions under which p_i decides. The first is straightforward: if p_i receives a message from another process that has already decided, then it too can decide (lines 3–6). Otherwise, process p_i decides at the end of round r_d if: (i) p_i has seen $\leq f$ failures; (ii) p_i observes at least $f + 2$ consecutive synchronous rounds; and (iii) the last two rounds appear failure-free, i.e. $Active_i[r_d] = Active_i[r_d - 1]$ (line 8). Notice that the size of $Failed_i[r_i]$ captures the number of failures that p_i has observed, and $sCount_i$ captures the number of consecutive synchronous rounds.

Procedure getEstimate(). The *getEstimate*() procedure is the key to the workings of the algorithm. The procedure begins by identifying a set of processes that have raised their flags, i.e., that are "ready to decide" (lines 3–4). The next portion of the procedure (lines 5-13) is dedicated to determining which of these flagged messages to prioritize, and which of these flags should be "discarded," i.e., treated with normal priority. Fix some process p_j whose message is

being considered. Process p_i first calculates which processes have a view that is incompatible with the view of p_j (line 6); specifically, these processes received a different set of messages in round $r_i - 1$ from process p_j. None of these processes can support a decision by any process that receives a message from p_j.

Next p_i fixes f_j to be the number of failures observed by process p_j (line 7), and determines that p_j's flag should be waived if the union of the "non-supporting" processes and the failed processes is at least $f_j + 1$ (line 8). In particular, this implies that if a process p_s receives p_j's message, then one of three events occurs: (i) process p_s receives a non-supporing message; (ii) process p_s receives a message from a process that was failed by p_j; or (iii) process p_s observes at least $f_j + 1$ failures. In all three cases, process p_s cannot decide. Thus it is safe for p_i to waive p_j's flag and treat its message with normal priority (lines 9-11).

At the end of this discard process, p_i chooses an estimate from among the remaining flagged messages, if any such messages exist (lines 14-19). Specifically, it chooses the minimum estimate from among the processes that have a maximal *sCount*, i.e., it prioritizes processes that have seen more synchronous rounds. Otherwise, if there are no remaining flagged messages, p_i chooses the minimum estimate that it has received (line 18).

5 Proof of Correctness

In this section, we prove that *ASAP* satisfies validity, termination and agreement. Validity is easily verified (see, for example, Proposition 2), so we focus on termination and agreement.

5.1 Definitions and Properties

We begin with a few definitions. Throughout, we denote the round in which a variable is referenced by a superscript: for example, est_i^r is the estimate of p_i at the end of round r. First, we say that a process perceives round r to be asynchronous if it later receives a message from a process that it believes to have failed in round r.

Definition 2 (Synchronous Rounds). *Given $p_i \in \Pi$ and rounds r, r_v, we say that round r is* asynchronous in p_i's view at round r_v *if and only if there exists round r' such that $r < r' \leq r_v$ and $Active_i^{r_v}[r'] \cap Failed_i^{r_v}[r] \neq \emptyset$. Otherwise, round r is* synchronous in p_i's view at round r_v.

A process perceives a round r as failure-free if it sees the same set of processes as alive in rounds r and $r + 1$.

Definition 3 (Failure-free Rounds). *Given $p_i \in \Pi$ and rounds r, r_v, we say that round $r \leq r_v$ is* failure-free in p_i's view at round r_v *if and only if* $Active_i^{r_v}[r] = Active_i^{r_v}[r + 1]$.

Note that, by convention, if a process p_m completes round r but takes no steps in round $r + 1$, p_m is considered to have failed in round r. We now state two simple, yet fundamental properties of *ASAP*:

```
1  procedure updateState()
2  begin
3      for every msg_j ∈ msgSet_i[r_i] do
           /* Merge newly received information */
4          for round r from 1 to r_i − 1 do
5              Active_i[r] ← msg_j.Active_j[r] ∪ Active_i[r]
6              Failed_i[r] ← msg_j.Failed_j[r] ∪ Failed_i[r]
7          end
8      end
9      for round r from 1 to r_i − 1 do
           /* Update AsynchRound flag */
10         AsynchRound_i[r] ← false
11         for round k from r + 1 to r_i do
12             if (Active_i[k] ∩ Failed_i[r] ≠ ∅) then AsynchRound_i[r] ← true
13         end
14     end
15     sCount_i ← max_ℓ(∀r_i − ℓ ≤ r' ≤ r_i, AsynchRound_i[r'] = true)
16 end
```

```
1  procedure checkDecisionCondition()
2  begin
3      if ∃msg_p ∈ msgSet_i s.t. msg_p.decided_p = true then
4          decide_i ← true
5          est_i ← msg_p.est_p
6          return decide_i
7      end
       /* If the previous f + 2 rounds are synchronous with at most f failures */
8      if (sCount ≥ |Failed_i[r_i]| + 2) and (Active_i[r_i] = Active_i[r_i − 1]) then
9          decide_i ← true
10         return decide_i
11     end
12 end
```

```
1  procedure getEstimate()
2  begin
3      flagProcSet_i ← {p_j ∈ Active_i[r_i] | msg_j.sFlag_j = true}
4      flagMsgSet_i ← {msg_j ∈ msgSet_i | msg_j.sFlag_j = true}
       /* Try to waive the priority on flagged messages. */
5      for p_j ∈ flagProcSet_i do
           /* Find the set of processes that disagree with p_j's view. */
6          nonSupport_i^j ← {p ∈ Active_i[r_i] : msg_p.Active_p[r_i − 1] ≠ msg_j.Active_j[r_i − 1]}
7          f_j ← |msg_j.Failed_j[r_i − 1]|
8          if (|nonSupport_i^j ∪ Failed_j[r_i − 1]| ≥ f_j + 1) then
9              msg_j.sFlag_j[r_i − 1] ← false
10             flagMsgSet_i ← flagMsgSet_i \ {msg_j}
11             flagProcSet_i ← flagProcSet_i \ {p_j}
12         end
13     end
       /* Adopt the min estimate of max priority; higher sCount has priority. */
14     if (flagMsgSet_i ≠ ∅) then
           /* The set of processes that have the highest sCount */
15         highPrSet ← {p_j ∈ flagMsgSet_i | msg_j.sCount_j = max_{p_l ∈ flagMsgSet_i}(sCount_l)}
16         est ← min_{p_j ∈ highPrSet}(est_j)
17     else
18         est ← min_{p_j ∈ msgSet_i}(est_j)
19     end
20     return est
21 end
```

Fig. 2. *ASAP* procedures.

Proposition 1 (Uniformity). *If processes p_i and p_j receive the same set of messages in round r, then they adopt the same estimate at the end of round r.*

Proposition 2 (Estimate Validity). *If all processes alive at the beginning of round r have estimate v, then all processes alive at the beginning of round $r+1$ will have estimate v.*

These properties imply that if the system remains in a bivalent state (in the sense of [11]), then a failure or asynchrony has to have occured in that round. Proposition 7 combines these properties with the asynchrony-detection mechanism to show that processes with synchronous views and distinct estimates necessarily see a failure for every round that they perceive as synchronous.

5.2 Termination

In this section, we show that every correct process decides by round $GST+f+2$, as long as there are no more than $f \leq t$ failures. Recall that a process decides when there are two consecutive rounds in which it perceives no failures. By the pigeonhole principle, it is easy to see that there must be (at least) two failure-free rounds during the interval $[GST+1, GST+f+2]$; unfortunately, these rounds need not be consecutive. Even so, we can show that at least one correct node must *perceive* two consecutive rounds in this interval as failure-free.

We begin by fixing an execution α with at most f failures, and fixing GST to be the round after which α is synchronous. We now identify two failure-free rounds in the interval $[GST+1, GST+f+2]$ such that in the intervening rounds, there is precisely one failure per round.

Proposition 3. *There exists a round $r_0 > GST$ and a round $r_\ell > r_0$ such that: (a) $r_\ell \leq GST + f + 2$; (b) rounds r_0 and r_ℓ are both failure free; (c) for every $r : r_0 < r < r_\ell$, there is exactly one process that fails in r; and (d) $\forall i > 0$ such that $r_0 + i < r_\ell$, there are no more than $(r_0 + i) - GST - 1$ failures by the end of round $r_0 + i$.*

The claim follows from a simple counting argument. Now, fix rounds r_0 and r_ℓ that satisfy Proposition 3. For every $i < \ell$: denote by r_i the round r_0+i; let q_i be the process that fails in round r_i; let $q_\ell = \perp$. Let S_i be the set of processes that are not failed at the beginning of round r_i. We now show that, for every round r in the interval $[r_1, r_{\ell-1}]$, if a process in S_r receives a message from q_r, then it decides at the end of round r. This implies that either every process decides by the end of r_ℓ, or, for all rounds r, no process in S_r receives a message from q_r.

Lemma 1. *Assume $r_0 + 1 < r_\ell$, and some process in S_ℓ does not decide by the end of r_ℓ. Then $\forall i : 0 < i < \ell$:*

(i) For every process $p \in S_{i+1} \setminus \{q_{i+1}\}$, process p does not receive a message from q_i in round r_i.

(ii) If process $q_{i+1} \neq \perp$ receives a message from q_i in round r_i, then process q_{i+1} decides at the end of r_i.

We can now complete the proof of termination:

Theorem 1 (Termination). *Every correct process decides by the end of round* $GST + f + 2$.

Proof (sketch). If $r_0 + 1 = r_\ell$, then it is easy to see that every process decides by the end of r_ℓ, since there are two consecutive failure-free rounds. Otherwise, we conclude by Lemma 1 that none of the processes in S_ℓ receive a message from $q_{\ell-1}$ in round $r_{\ell-1}$. Thus every process receives messages from $S_{\ell-1} \setminus \{q_{\ell-1}\}$ both in rounds $r_{\ell-1}$ and r_ℓ, which implies that they decide by the end of r_ℓ.

5.3 Agreement

In this section, we prove that no two processes decide on distinct values. Our strategy is to show that once a process decides, all non-failed processes adopt the decision value at the end of the decision round (Lemma 2). Thus, no decision on another value is possible in subsequent rounds.

Synchronous Views. The key result in this section is Proposition 7, which shows that in executions perceived as synchronous, there is at least one (perceived) failure per round. The idea behind the first preliminary proposition is that if an estimate is held by some process at round r, then there exists at least one process which "carries" it for every previous round.

Proposition 4 (Carriers). *Let* $r > 0$ *and* $p \in \Pi$. *If* p *has estimate* v *at the end of round* r, *then for all rounds* $0 \leq r' \leq r$, *there exists a process* $q^{r'} \in Active_p^r[r']$ *such that* $est_{q^{r'}}[r' - 1] = v$.

Next, we prove that processes with synchronous views see the same information, with a delay of one round. This follows from the fact that processes communicate with a majority in every round.

Proposition 5 (View Consistency). *Given processes* p_i *and* p_j *that see rounds* $r_0 + 1, \ldots, r_0 + \ell + 1$ *as synchronous:* $\forall r \in [r_0 + 1, r_0 + \ell], Active_i^{r_0+\ell+1}[r + 1] \subseteq Active_j^{r_0+\ell+1}[r]$.

The next proposition shows that if a process observes two consecutive synchronous rounds r and $r + 1$ with the same set of active processes S, then all processes in S receive the same set of messages during round r.

Proposition 6. *Let* r, r_c *be two rounds such that* $r_c > r$. *Let* p *be a process that sees round* r *as synchronous from round* r_c. *If* $Active_p^{r_c}[r] = Active_p^{r_c}[r + 1]$, *then all processes in* $Active_p^{r_c}[r]$ *receive the same set of messages in round* r.

The next proposition is the culmination of this section, and shows that in periods of perceived synchrony, the amount of asynchrony in the system is limited. It captures the intuition that at least one process fails in each round in order to maintain more than one estimate in the system. Recall, this is the key argument for solving consensus in a synchronous environment.

Proposition 7. *Given processes p_i, p_j that see rounds $r_0+1, \ldots, r_0+\ell+1$ as synchronous and adopt distinct estimates at the end of round $r_0+\ell+1$, then for all $r \in [r_0+1, r_0+\ell]$, $|Active_i^{r_0+\ell+1}[r+1]| < |Active_i^{r_0+\ell+1}[r]|$.*

Proof (sketch). We proceed by contradiction: assume there exists a round $r \in [r_0+1, r_0+\ell]$ such that $Active_i^{r_0+\ell+1}[r+1] = Active_i^{r_0+\ell+1}[r]$. This implies that all processes in $Active_i^{r_0+\ell+1}[r]$ received the same set of messages in round r by Proposition 6. Proposition 1 then implies that all processes in $Active_i^{r_0+\ell+1}[r]$ have adopted the same estimate at the end of round r, that is, they have adopted $est_i^{r_0+\ell+1}$.

Proposition 4 implies that there exists a process $p \in Active_j^{r_0+\ell+1}[r+1]$ that adopts estimate $est_j^{r_0+\ell+1}$ at the end of r. By the above, this process is not in $Active_i^{r_0+\ell+1}[r]$. This, together with the fact that $est_i^{r_0+\ell+1} \neq est_j^{r_0+\ell+1}$ implies that $p \in Active_j^{r_0+\ell+1}[r+1] \setminus Active_i^{r_0+\ell+1}[r]$, which contradicts Proposition 5.

Decision Condition. In this section, we examine under which conditions a process may decide, and under what conditions a process may not decide. These propositions are critical to establishing the effectiveness of the estimate-priority mechanism. The following proposition shows that every decision is "supported" by a majority of processes with the same estimate. Furthermore, these processes have a synchronous view of the previous rounds.

Proposition 8. *Assume process p_d decides on v_d at the end of r_0+f+2, seeing $f+2$ synchronous rounds and f failures (line 10 of* checkDecisionCondition*). Let $S := Active_d^{r_0+f+2}[r_0+f+2]$. Then:*

(i) For all $p \in S$, $Active_p^{r_0+f+1}[r_0+f+1] = S$ and $est_p^{r_0+f+1} = v_d$.
(ii) At the end of r_0+f+1, processes in S see rounds $r_0+1, r_0+2, \ldots, r_0+f+1$ as synchronous rounds in which at most f failures occur.

The proposition follows from a careful examination of the decision condition. Next, we analyze a sufficient condition to ensure that a process *does not decide*, which is the basis for the flag-discard rule:

Proposition 9. *Let p be a process with $sFlag = true$ at the end of round $r > 0$. If there exists a process q such that $q \in Active_p^r[r]$ and $Active_q^r[r] \neq Active_p^r[r]$, then no process that receives p's message in round $r+1$ decides at the end of round $r+1$.*

Notice that if a process receives a message from p and not from q, then it sees q as failed; otherwise, if it receives a message from both, it sees a failure in $r-1$. In neither case can the process decide. The last proposition is a technical result that bounds a process's estimate in rounds in which it receives a flagged estimate:

Proposition 10. *Let $r > 0$ and $p \in \Pi$. Let $flagProcSet_p^r$ be the set of processes in $Active_p^r[r]$ with $sFlag = true$. Assume $flagProcSet_p^r$ is non-empty, and let q be a process such that, $\forall s \in flagProcSet_p^r, est_q^{r-1} \leq est_s^{r-1}$, also $q \notin Failed_s^{r-1}[r-1]$ and p receives a message from q in round r. Then $est_p^r \leq est_q^{r-1}$.*

Safety. We now prove the key lemma which shows that if some process decides, then every other non-failed process has adopted the same estimate. The first part of the proof uses Propositions 5 and 7 to determine precisely the set of processes that remain active just prior to the decision, relying on the fact that there must be one new failure per round. The remainder of the proof carefully examines the behavior in the final two rounds prior to the decision; we show that in these rounds, every process must adopt the same estimate. This analysis depends critically on the mechanism for prioritizing estimates, and thus relies on Proposition 10.

Lemma 2 (Safety). *Let r_d be the first round in which a decision occurs. If process p_d decides on value v in round r_d, then every non-failed process adopts v at the end of round r_d.*

Proof (sketch). Assume for the sake of contradiction that there exists a process q such that $est_q^{r_d} = u \neq v$. Fix f to be the number of failures observed by process p_d and fix round $r_0 > 0$ such that $r_d = r_0 + f + 2$. The case where $f \in \{0, 1\}$ needs to be handled separately; in the following, we assume that $f > 1$.

Since p_d decides at the end of r_0+f+2, Proposition 8 implies that there exists a support set S of at least $n-f$ processes such that p_d receives a message in round $r_0 + f + 2$ from all processes in S, and $\forall p \in S, Active_p^{r_0+f+1}[r_0 + f + 1] = S$. Furthermore, processes in S have $sCount \geq f + 1$ and $est = v$ at the end of $r_0 + f + 1$. Since process q receives at least $n - t$ messages in round $r_0 + f + 2$, it necessarily receives a message from a process in S. Denote this process by p_i. We make the following claim:

Claim. Process q receives a message from some process p_j in round $r_0 + f + 1$ such that $est_j = u$, $p_j \notin S$, $sFlag_j = true$ and $sCount_j \geq f + 1$.

The claim follows from the observation that q cannot discard p_i's flag (as per Proposition 9), therefore there has to exist a process p_j with estimate u and flag set with priority at least as high as p_i's. Hence, at the end of round r_0+f+1 we have two processes p_i and p_j that see rounds $r_0+1, \ldots, r_0+f+1$ as synchronous and adopt distinct estimates. This leads to the following claim:

Claim. For every process $p \in S \cup \{p_j\}$, $Active_p^{r_0+f+1}[r_0 + f] = S \cup \{p_j\}$.

In particular, Proposition 7 implies that p_j sees one failure per round, and hence $|Active_j^{r_0+f+1}[r_0 + f]| \leq n - f + 1$. Since $Active_i^{r_0+f+1}[r_0 + f + 1] = S$, Proposition 5 implies that $S \cup \{p_j\} \subseteq Active_j^{r_0+f+1}[r_0 + f]$. Since $p_j \notin S$, we conclude that $S \cup \{p_j\} = Active_j^{r_0+f+1}[r_0 + f]$. A similar argument yields that, for all $p \in S, Active_p^{r_0+f+1}[r_0 + f] = S \cup \{p_j\}$.

In the remaining portion of the proof, we show that no process in $S \cup \{p_j\}$ adopts estimate $max(u, v)$ at the end of r_0+f+1, which leads to a contradiction. Let $m := min(u, v)$ and $M := max(u, v)$. Proposition 4 ensures that there exist processes $p_m, p_M \in S \cup \{p_j\}$ such that $est_m^{r_0+f-1} = m$ and $est_M^{r_0+f-1} = M$. Let $f_j = |Failed_j^{r_0+f+1}[r_0 + f + 1]|$. We can then conclude:

Claim. There exists a set S' of at least $n - f_j - 1$ processes in S such that every process in $S \cup \{p_j\}$ receives messages from S' in round $r_0 + f + 1$ and processes in S' have $est^{r_0+f} \leq min(u, v)$.

To see this, notice that process p_j receives exactly $n - f_j$ messages in round $r_0 + f + 1$; one of these messages must have been sent by p_j itself, while the remaining $n - f_j - 1$ of these messages were sent by processes in S. We denote these processes by S'. Notice that the processes in S' are not considered failed by other processes in S in round $r_0 + f + 1$ since they support p_d's decision in round $r_0 + f + 2$. It follows that the processes in S' have received messages from every process in $S \cup \{p_j\}$ in round $r_0 + f$. With some careful analysis, we can apply Proposition 10 to conclude that for all $s \in S'$, $est_s^{r_0+f} \leq m$, from which the claim follows. Finally, we show that, because of S', no process in $S \cup \{p_j\}$ can adopt M at the end of $r_0 + f + 1$, which contradicts the existence of either p_i or p_j, concluding the proof.

Claim. For every process p in $S \cup \{p_j\}$, $est_p^{r_0+f+1} \leq m$.

This follows because every process p in S receives a message from a process $s \in S'$ in round $r_0 + f + 1$, and no other process in S could have failed s in $r_0 + f$; thus we can again apply Proposition 10 to conclude that $est_p^{r_0+f+1} \leq est_s^{r_0+f} \leq m$, and the claim follows, which concludes the proof of Lemma 2.

We can now complete the proof of agreement:

Theorem 2 (Agreement). *No two processes decide on different estimates.*

Proof (sketch). Let r_d be the first round in which a decision occurs. Since majority support is needed for a decision (see Proposition 8), all processes deciding in r_d decide on the same value. Lemma 2 shows that all processes adopt the same estimate at the end of r_d, and by Proposition 2, no other value is later decided.

6 Conclusions and Future Work

We have demonstrated an optimally-resilient consensus protocol for the eventually synchronous model that decides *as soon as possible*, i.e., within $f + 2$ rounds of *GST* in every execution with at most f failures. It remains an interesting question for future work as to whether these techniques can be extended to k-set agreement and Byzantine agreement. In particular, it seems possible that the mechanism for assigning priorities to estimates based on what a process can *prove* about the system may be useful in both of these contexts. Indeed, there may be interesting connections between this technique and the knowledge-based approach (see, e.g., [15]).

References

1. R. Boichat, P. Dutta, S. Frolund, and R. Guerraoui. Deconstructing paxos. *SIGACT News*, 34(1):47–67, 2003.
2. R. Boichat, P. Dutta, S. Frolund, and R. Guerraoui. Reconstructing paxos. *SIGACT News*, 34(2):42–57, 2003.
3. T. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, 1996.
4. T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.
5. D. Dolev, R. Reischuk, and H. R. Strong. Early stopping in byzantine agreement. *J. ACM*, 37(4):720–741, 1990.
6. P. Dutta and R. Guerraoui. The inherent price of indulgence. In *PODC*, pages 88–97, 2002.
7. P. Dutta and R. Guerraoui. The inherent price of indulgence. *Distributed Computing*, 18(1):85–98, 2005.
8. P. Dutta, R. Guerraoui, and Idit Keidar. The overhead of consensus failure recovery. *Distributed Computing*, 19(5-6):373–386, 2007.
9. P. Dutta, R. Guerraoui, and L. Lamport. How fast can eventual synchrony lead to consensus? In *DSN*, pages 22–27, 2005.
10. C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.
11. M. Fisher, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.
12. E. Gafni. Round-by-round fault detectors: Unifying synchrony and asynchrony (extended abstract). In *PODC*, pages 143–152, 1998.
13. R. Guerraoui. Indulgent algorithms (preliminary version). In *PODC*, pages 289–297, 2000.
14. R. Guerraoui and M. Raynal. The information structure of indulgent consensus. *IEEE Transactions on Computers*, 53(4):453–466, 2004.
15. J. Y. Halpern and Y. Moses. Knowledge and common knowledge in a distributed environment. *J. ACM*, 37(3):549–587, 1990.
16. I. Keidar and S. Rajsbaum. On the cost of fault-tolerant consensus when there are no faults: preliminary version. *SIGACT News*, 32(2):45–63, 2001.
17. I. Keidar and A. Shraer. Timeliness, failure-detectors, and consensus performance. In *PODC*, pages 169–178, 2006.
18. I. Keidar and A. Shraer. How to choose a timing model? In *DSN*, pages 389–398, 2007.
19. L. Lamport. Generalized consensus and paxos. *Microsoft Research Technical Report MSR-TR-2005-33*, March 2005.
20. L. Lamport. Lower bounds for asynchronous consensus. *Distributed Computing*, 19(2):104–125, 2006.
21. L. Lamport and M. Fisher. Byzantine generals and transaction commit protocols. Unpublished, April 1982.
22. L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.
23. Leslie Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.
24. N. Lynch. *Distributed Algorithms*. Morgan Kaufman, 1996.
25. M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.