On implementing Ω with weak reliability and synchrony assumptions

Marcos K. Aguilera[1] Carole Delporte-Gallet[2] Hugues Fauconnier[2] Sam Toueg[3]

Abstract

We study the feasibility and cost of implementing Ω—a fundamental failure detector at the core of many algorithms—in systems with weak reliability and synchrony assumptions. Intuitively, Ω allows processes to eventually elect a common leader. We first give an algorithm that implements Ω in a weak system S where processes are synchronous, but: (a) any number of them may crash, and (b) only the output links of an unknown correct process are eventually timely (all other links can be asynchronous and/or lossy). This is in contrast to previous implementations of Ω which assume that a quadratic number of links are eventually timely, or systems that are strong enough to implement the eventually perfect failure detector $\diamond\mathcal{P}$. We next show that implementing Ω in S is expensive: even if we want an implementation that tolerates just one process crash, all correct processes (except possibly one) must send messages forever; moreover, a quadratic number of links must carry messages forever. We then show that with a small additional assumption—the existence of some unknown correct process whose asynchronous links are lossy but fair—we can implement Ω efficiently: we give an algorithm for Ω such that eventually only *one* process (the elected leader) sends messages.

Contact author: Marcos K. Aguilera

Student paper: No

Type of submission: Regular presentation only

[1]HP Systems Research Center, 1501 Page Mill Rd, Mail Stop 1250, Palo Alto, CA, 94304, USA, aguilera@hpl.hp.com

[2]LIAFA, Université D. Diderot, 2 Place Jussieu, 75251, Paris Cedex 05, France, {cd,hf}@liafa.jussieu.fr

[3]Department of Computer Science, University of Toronto, Toronto, sam@cs.toronto.edu

1 Introduction

Background, motivation and results

Failure detectors are basic tools of fault-tolerant distributed computing that can be used to solve fundamental problems such as consensus, atomic broadcast, and group membership. For this reason there has been growing interest in the implementation of failure detectors [22, 18, 11, 18, 19, 20, 1, 6, 9, 2].

A failure detector of particular interest is Ω [4]. Roughly speaking, with Ω every process p has a local variable $leader_p$ that contains the identity of a single process that p currently trusts to be operational (p considers this process to be its current leader). Initially, different processes may have different leaders, but Ω guarantees that there is a time after which all processes have the *same*, *non-faulty* leader. Failure detector Ω is important for both theoretical and practical reasons: it has been shown to be the weakest failure detector with which one can solve consensus [4], and it is the failure detector used by several consensus algorithms, including some that are used in practice (e.g., [16, 12, 21, 20, 13]).

In this paper, we study the problem of implementing Ω in systems with weak reliability and synchrony assumptions, and we are particularly interested in communication-efficient implementations.

Our starting point are systems where all links are asynchronous and lossy: messages can suffer arbitrary delays or even be lost. In addition, processes may crash, but we assume that they are synchronous: their speed is bounded and they have local clocks that can measure real-time intervals. We denote such a system by S^-.

Since all messages can be lost or arbitrarily delayed in S^-, it is clear that Ω cannot be implemented in such a system. Thus, we make the following additional assumption: there is at least *one* correct pro-

cess whose *output* links are eventually timely (intuitively, this means that there is an unknown bound δ and a time after which every message sent from that process is received within δ time). We call such a process *an eventually timely source*, or simply a *source*, and we denote by S a system S^- with at least one source. Note that processes do *not* know the identity of the source(s) in S, the time after which their output links become timely, or the corresponding message delay bound(s). Moreover, except for the output links of the source(s), all the other links in S may be asynchronous and/or lossy.

S is a very weak system because processes may be unable to communicate with each other. In S only messages sent *by* the unknown source(s) are guaranteed to be received. All other messages, including all those sent *to* the source(s), can be lost. Thus, processes cannot use sources as "forwarding nodes" to communicate reliably with each other.

Can one implement Ω in system S? Note that Ω requires that processes eventually *agree* on a common leader, and it is not obvious how to achieve this agreement in a system where processes may be unable to communicate with each other. For example, consider a system S where p and q cannot communicate (say, all the messages they send are lost). Suppose there are three other processes s_1, s_2 and s_3, such that s_1 and s_2 behave timely towards p, and s_2 and s_3 behave timely towards q, but the messages from s_1 to q, and those from s_3 to p, are often lost or greatly delayed. For p, the natural leader candidates are s_1 and s_2; while for q the candidates are s_2 and s_3. Any implementation of Ω must ensure that p and q eventually agree on the same leader—a nontrivial task here since p and q cannot communicate with each other (or with any other process).

Our first result is an algorithm that implements Ω in system S. So processes are indeed able to agree on a common leader despite permanent communication failures and any number of crashes. This algorithm for Ω, however, has a serious drawback: it forces *all* the processes to periodically send messages forever. This is undesirable, and perhaps it can be avoided. Intuitively, after a process becomes the common leader,[1] it must periodically send messages forever (because if it crashes, processes must be able to notice this failure and chose a new leader); but from now on no other process needs to be monitored. Thus, after processes agree on a common leader, no process other than the leader should have to send messages. This leads us to the following definition and a related question. An algorithm for Ω is *communication-efficient* if there is a time after which only one process sends messages. Is there a communication-efficient algorithm for Ω in system S?

To answer this question we investigate the communication complexity of implementations of Ω in system S, and we derive two types of lower bounds: one on the number of processes that must send messages forever, and one on the number of links that must carry messages forever. Specifically, we show that for any algorithm for Ω in S: (a) in every run all correct processes, except possibly one, must send messages forever; and (b) in some run at least $(n^2-1)/4$ links must carry messages forever, where n is the number of processes in S. These lower bounds hold even if we assume that at most one process may crash. We conclude that there is no communication-efficient algorithm for Ω in S that can tolerate one crash.

We next consider how to strengthen S so that communication efficiency can be achieved. Specifically, since our impossibility result relies on the lack of reliable communication in S, we consider the following additional assumption: there is at least *one* unknown correct process such that the links to and from that process are *fair*. A fair link may lose messages, but it satisfies the following property: messages can be partitioned into types, and if messages of some type are sent infinitely often, then messages of that type are also received infinitely often. A correct process whose input and output links are fair is said to be a *fair hub*, or simply a *hub*. We denote by S^+ a system S with at least one hub (whose identity is not known).[2]

S^+ is a weak system because it does not ensure pairwise *timely* communication. In fact, in S^+ only messages sent *from* the source(s) are guaranteed to be eventually timely. All other messages, including all those sent *to* the source(s), can be arbitrarily delayed. Thus, processes cannot use sources as intermediate

[1] Note that processes may never know whether this has already occurred.

[2] So S^+ is a system S^- with at least one eventually timely source *and* at least one fair hub, whose identities are not known.

System	*Properties*
S^-	*Links:* asynchronous and lossy *Processes:* synchronous and subject to crashes
S	S^- with at least one *eventually timely source* (i.e., a correct process whose *output* links are eventually timely)
S^+	S with at least one *fair hub* (i.e., a correct process whose *input and output* links may be lossy but are fair)

Table 1: Systems considered in this paper.

System	Ω algorithm	Communication-efficient Ω algorithm
S^-	No	No
S	Yes	No
S^+	Yes	Yes

Table 2: Implementability of Ω.

nodes to communicate with each other in a timely way.

Our next result is a *communication-efficient* algorithm for Ω in S^+. We derive this algorithm in two stages: we first give a simpler algorithm for a system S where *all* the links are fair (rather than just the links of an unknown hub). We then modify this algorithm so that it works in S^+. Both algorithms tolerate any number of crash failures (in addition to message losses in lossy links). Tables 1 and 2 summarize our results on the implementability (and communication efficiency) of Ω.

In summary, our contributions are the following:

1. We study the feasibility and cost of implementing Ω—a fundamental failure detector at the core of many algorithms—in systems with weak reliability and synchrony assumptions.

2. We give the first algorithm that implements Ω in a weak system where only the output links of some unknown correct process are eventually timely (all other links can be asynchronous and/or lossy) . This is in contrast to previous implementations of Ω which require systems with a quadratic number of eventually timely links, or systems that are strong enough to implement $\diamond\mathcal{P}$.

3. We show that implementing Ω in this weak system is expensive: all correct processes (except possibly one) must send messages forever; moreover, a quadratic number of links must carry messages forever. This holds even if we want an implementation that tolerates just one process crash.

4. We then show that with a small additional assumption—the existence of some unknown correct process whose asynchronous links are lossy but fair—we can implement Ω efficiently, i.e, such that eventually only one process (the elected leader) sends messages.

As a final remark, we note that the *eventually perfect failure detector* $\diamond\mathcal{P}$[3] can *not* be implemented in system S. So S is an example of a system that is strong enough to implement Ω but too weak to implement $\diamond\mathcal{P}$. This, together with our results on the cost of implementing Ω in S and S^+, partially answers some questions posed by Keidar and Rajsbaum in their 2002 PODC tutorial [15].

Related work

Related work concerns the *use of* Ω to solve agreement problems (e.g., consensus and atomic broadcast), and the *implementation of* Ω in various types of partially synchronous systems [10].

Ω is the weakest failure detector that can be used to solve consensus and atomic broadcast in systems with a majority of correct processes [5, 4], and it is the failure detector required by several algorithms [16, 17, 12, 21, 20, 3, 13].

A simple implementation of Ω consists of implementing $\diamond\mathcal{P}$ first and outputting the smallest process currently not suspected by $\diamond\mathcal{P}$ [16, 8, 15]. But this approach has serious drawbacks. In particular, it requires a system that is strong enough to implement $\diamond\mathcal{P}$ (a failure detector that is strictly stronger than Ω), and it requires *all* processes to send messages forever (just to implement $\diamond\mathcal{P}$).

Several papers have focused on reducing the communication overhead of failure detector implementations. The algorithm in [18] implements failure detector $\diamond\mathcal{S}$[4] in a way that only n links carry messages

[3] Informally, $\diamond\mathcal{P}$ ensures two properties: (a) any process that crashes is eventually suspected by every correct process, and (b) there is a time after which correct processes are never suspected.

[4] Informally, $\diamond\mathcal{S}$ ensures two properties: (a) any process that

forever. But this algorithm requires very strong system properties, namely, that no message is ever lost, and all links are eventually timely in both directions. [19] has an algorithm for Ω, but the paper assumes some strong system properties: all links are eventually reliable and timely.

Another communication-efficient implementation of Ω was given in [1]. In that implementation, only the links to and from some (unknown) correct process need to be eventually timely, all other links can be asynchronous and lossy. This system assumption is weaker than the ones in [18, 19]. But it is stronger than the one we assume here for S^+: indeed it is strong enough to allow the implementation of $\diamond\mathcal{P}$ (which cannot be implemented in S^+).

Another related result is the algorithm that transforms $\diamond\mathcal{S}$ to Ω given in [7]. Note that one way to implement Ω is to first implement $\diamond\mathcal{S}$, and then use this transformation algorithm. But this approach cannot be used to implement Ω in system S: In fact, the transformation algorithm in [7] requires all processes to reliably communicate with each other (which may not be possible in S). Moreover, one must implement $\diamond S$ first, and it is not clear how this can be done by a communication-efficient algorithm in S^+, short of using our algorithm, which already implements Ω.

Roadmap

We first give an informal model of systems S^-, S and S^+ (Section 2). We then consider the problem of implementing Ω in S: we first give an algorithm for Ω in S (Section 3), and then derive lower bounds on the communication complexity of this problem (Section 4). We next strengthen system S to derive communication-efficient algorithms for Ω: the first algorithm assumes that all links are fair (Section 5.1), the second one assumes that the links of some unknown correct process are fair, i.e., it works in system S^+ (Section 5.2). A brief discussion concludes the paper. Because of space limitations, all proofs have been moved to an optional appendix.

crashes is eventually suspected by every correct process, and (b) there is a time after which some correct process is never suspected.

2 Informal model

We consider distributed systems with $n \geq 2$ processes $\Pi = \{0, \ldots, n-1\}$ that can communicate with each other by sending messages through a set of unidirectional links Λ. We now describe the behavior of processes and links in more detail.

Processes. Processes execute by taking steps. In a step a process can either receive a set of messages and then change its state, *or* it can send a message and then change its state.[5] The value of a variable of a process at time t is the value of that variable after the process takes a step at time t. There is a lower and upper bound on the rate of execution (number of steps per time unit) of any non-faulty process. Processes have clocks that are not necessarily synchronized, but we assume that they can accurately measure intervals of time (it is easy to extend our results to clocks with bounded drift rates).

A process can fail by permanently crashing, in which case it stops taking steps. We say that *process p is alive at time t* if it has not crashed by time t. We say a process is *correct* if it is always alive. We say a process is *faulty* if it is not correct. Unless we explicitly state otherwise, we consider systems where *any number of processes may crash*.

Links. We assume that the network is fully connected, i.e., $\Lambda = \Pi \times \Pi$. The unidirectional link from process p to process q is denoted by $p \rightarrow q$. We consider various types of links, all of which satisfy the following property:

- *[Integrity]:* Process q receives a message m from process p at most once, and only if p previously sent m to q.[6]

A link $p \rightarrow q$ is *eventually timely* if it satisfies Integrity and the following property:

- *[Eventual timeliness]:* There exists a δ and a time t such that if p sends m to q at a time $t' \geq t$ and q is correct, then q receives m from p by time $t' + \delta$.

[5] Our lower bounds also hold in a stronger model in which a process can receive, change state, *and* send in a single atomic step.

[6] We assume that messages are unique, e.g., each message contains the id of the sender and a sequence number (this is implicit in all our algorithms).

The maximum message delay δ associated with an eventually timely link is not known.

A link that intermittently loses messages may satisfy a *fairness* property. To define this property, we assume that messages carry a *type* in addition to its *data*. Fairness requires that if a process sends an infinite number of messages of a type through a link then the link delivers an infinite number of messages of that type. More precisely, we assume that messages consists of pairs $m = (type, data) \in \Sigma^* \times \Sigma^*$ where $\Sigma = \{0, 1\}$. A link $p \rightarrow q$ is *fair* if it satisfies Integrity and the following property:

- *[Fairness]*: For every $type$, if p sends infinitely many messages of type $type$ to q and q is correct, then q receives infinitely many messages of type $type$ from p.

We classify correct processes based on the properties of their links. An *eventually timely source* (or simply *source*) is a correct process whose *output* links are all eventually timely. (Only the *outgoing* links need to be timely, hence the word "source".) A *fair hub* (or simply *hub*) is a correct process whose input *and* output links are all fair.

Systems. We consider three systems, S^-, S, and S^+ that differ on the properties of their links (their process properties are those described at the beginning of this section). In all three systems all links satisfy the Integrity property. System S^- has no further requirements; in particular, all links can be asynchronous and/or lossy. In system S, we assume that there is at least one eventually timely source (whose identity is unknown). Except for the output links of the source(s), all other links can be asynchronous and/or lossy, and so most pairs of processes may be unable to communicate in S. System S^+ is a strengthening of system S that allows communication between all pairs of processes through some unknown hub. More precisely, S^+ is a system with at least one eventually timely source *and* at least one fair hub (their identities are not known). Note that in S^+ most pairs of processes may not be able communicate in a timely fashion: even though they are fair, the links of the unknown hub(s) can still be asynchronous and/or lossy.

2.1 Failure detector Ω

The formal definition of failure detector Ω is given in [5, 4]. Informally, Ω outputs, at each process p, a single process denoted $leader_p$, such that the following property holds:

- There exists a correct process ℓ and a time after which, for every alive process p, $leader_p = \ell$.

If at time t, *leader*$_p$ contains the same correct process ℓ for all alive processes p, then we say that ℓ *is the leader at time* t. Note that at any given time processes do not know if there is a leader; they only know that eventually a leader emerges and remains.

2.2 Communication efficiency

We are interested in failure detector algorithms that minimize the usage of communication links. Note that in any reasonable implementation of a failure detector, some process needs to send messages forever. However, not every process needs to do that. We say that an implementation of failure detector Ω is *communication-efficient* if there is a time after which only one process sends messages.

3 Implementing Ω in system S

We now give an algorithm that implements Ω in S. This algorithm ensures that processes eventually agree on a common leader, even though most pairs of processes may be unable to communicate with each other (recall that in S all links can be asynchronous and lossy, except for the *output* links of some unknown correct process).

The basic idea is that each process selects its leader among the processes that seem to be currently alive. But since almost all links in S may suffer from arbitrary delays and/or losses, there are several problems that must be overcome. In particular: (a) different processes may have different views of which processes are currently alive, and the different views may never converge, (b) some processes may repeatedly alternate between appearing to be alive and dead, and continue to do so forever. Such problems complicate the task of selecting a common and permanent leader. For example, process p cannot simply select as its leader the "smallest" process that seem

to be currently alive: problem (a) may cause different processes to have different leaders (forever), and problem (b) may cause p to change its leader forever.

To overcome these and other similar difficulties, processes maintain "accusation" counters, and they indirectly use the unknown source(s) to help them converge on the same correct process as the leader. The algorithm, described in full in Figure 1, roughly works as follows. Each process p maintains a set $set2$ of processes that it considers to be currently alive: these are p's current candidates for leadership. To select among the different candidates, p also maintains a $counter[q]$ for each process q, which is p's rough estimate of how many times q's was previously suspected of being dead. Process p selects as its leader the process ℓ in its $set2$ that has the smallest $(counter[\ell], \ell)$ tuple. (The leader is recomputed whenever the set $set2$ or the counters are updated).

To help each process maintain its $set2$ and $counter$ variables, every process q sends $(\text{ALIVE}, q, counter[q])$ messages periodically, say every η time, to all other processes (note that most or all of these messages may be lost or delayed arbitrarily). If a process p receives $(\text{ALIVE}, q, counter[q])$ *directly* from q, p relays the message *once* to every other process, and it also resets a local timer, denoted $timer1(q)$, to expire after $Timeout1[q]$ time units[7], which is the maximum delay that p expects until p receives the next $(\text{ALIVE}, q, counter[q])$ *directly* from q.[8] If $timer1(q)$ expires before receiving this message directly from q, then p sends an ACCUSATION message to q. When q receives an ACCUSATION message, it increments its $counter[q]$.

If p receives $(\text{ALIVE}, q, counter[q])$ either *directly* from q or *relayed* by another process, then p adds q to its set $set2$, it updates its $counter[q]$ variable accordingly, and it also resets a timer, denoted $timer2(q)$, for when it expects to receive the next $(\text{ALIVE}, q, counter[q])$ (directly or relayed). If $timer2(q)$ expires before receiving this message, then p removes q from $set2$.[9]

[7]Note that $timer1(q)$ is set to a time interval rather than an absolute time. The timer is decremented until it expires.

[8]In the algorithm's code, shown in Figure 1, p also adds q to a set denoted $set1$, but this set is only used to facilitate the proof of correctness.

[9]Since p does not know the maximum message delays associated with eventually timely links, every time a timer expires p increments the associated timeout.

```
Code for each process p:

procedure updateLeader()
1  leader ← ℓ such that (counter[ℓ], ℓ) =
             min{(counter[q], q) : q ∈ set2}

on initialization:
2  ∀q ≠ p : Timeout1[q] ← η + 1
3  ∀q ≠ p : Timeout2[q] ← η + 1
4  ∀q ≠ p : reset timer1(q) to Timeout1[q]
5  ∀q ≠ p : reset timer2(q) to Timeout2[q]
6  ∀q : counter[q] ← 0
7  set1 ← {p}; set2 ← {p}
8  start tasks 1 and 2

task 1:
9  loop forever
10   send (ALIVE, p, counter[p]) to every process except
               p every η time

task 2:
11 upon receive (ALIVE, q, c) from q′ do
12   if q = q′ then
13     reset timer1(q) to Timeout1[q]
14     set1 ← set1 ∪ {q}
15     send (ALIVE, q, c) to every process except p and q
16   reset timer2(q) to Timeout2[q]
17   set2 ← set2 ∪ {q}
18   counter[q] ← max{counter[q], c}
19   updateLeader()

20 upon expiration of timer1(q) do
21   send ACCUSATION to q
22   set1 ← set1 − {q}
23   Timeout1[q] ← Timeout1[q] + 1
24   reset timer1(q) to Timeout1[q]

25 upon expiration of timer2(q) do
26   set2 ← set2 − {q}
27   Timeout2[q] ← Timeout2[q] + 1
28   reset timer2(q) to Timeout2[q]
29   updateLeader()

30 upon receive ACCUSATION do
31   counter[p] ← counter[p] + 1
```

Figure 1: Implementation of Ω for system S.

Theorem 1 *The algorithm in Figure 1 implements Ω in system S.*

4 Impossibility of communication-efficient Ω in system S

We now consider the communication complexity of implementations of Ω in system S. Specifically we give two types of lower bounds: one is on the *number of processes* that send messages forever, and the other is on the *number of links* that carry messages forever. A corollary of these lower bounds is that there is no communication-efficient implementation of Ω in system S. The bounds that we derive here apply even if we assume that *at most one process may crash.*

Theorem 2 *Consider any algorithm for Ω in a system S with n processes where at most one process may crash.*

1. *In* every *run, all correct processes, except possibly one, send messages forever.*
2. *In* some *run, at least $\lceil \frac{(n^2-1)}{4} \rceil$ links carry messages forever.*

From Theorem 2(1), we immediately get the following:

Corollary 3 *There is no communication-efficient algorithm for Ω in a system S with $n \geq 3$ processes, even if we assume that at most one process may crash.*

5 Communication-efficient implementations of Ω

We now seek algorithms for Ω that require only one process to send messages forever (this also implies that the number of links that carry messages forever is linear rather than quadratic). In order to achieve this, Theorem 2 implies that we must strengthen the system model S. In this section, we first give a communication-efficient algorithm for Ω that assumes that *all* links in S are fair (note that this system is stronger than S^+). We next modify this algorithm so that it works in a system S where only the links to and from some unknown correct process are fair, i.e., it works in system S^+.

5.1 Efficient implementation in a system S where all links are fair

We now seek a communication-efficient algorithm for Ω in a system S (i.e., a system with an eventually timely source) with the extra assumption that all links are fair. One simple attempt to get communication efficiency is as follows. Each process: (a) sends ALIVE messages *only if it thinks it is the leader*, (b) maintains a set of processes, called *Contenders*, from which it received an ALIVE message recently (an adaptive timeout mechanism is used to determine if a message is "recent"), and (c) chooses as leader the process with smallest id in its current set of contenders.[10] Such a simple algorithm would work in a system where *all* correct processes are sources. But in our system, it would fail: if the *only* source happens to be a process with a large id, the leadership could forever oscillate between the smaller correct processes.

One way to fix this problem is to estimate for each process the number of times it was previously accused of being slow. These accusation counters—rather than the process ids—are then used to select the leader among the current set of contenders. More precisely, each process keeps a counter on the number of times it was accused of being slow, and includes this counter in the ALIVE messages that it sends. Every process keeps the most up-to-date counter that it received from every other process, and picks as its leader the process with the smallest counter among the current set of contenders (using process id to break ties). If a process times out on a current contender, it sends an "accusation" message to this contender, which causes the contender to increment its own accusation counter. The hope is that the counter of each *source* remains bounded (because all its links are eventually timely), and so the source with the smallest counter is eventually selected as the leader by all.

This algorithm, however, does not work, because the accusation counter of a source may keep increas-

[10] A process always considers itself to be a contender, so if it does not have recent ALIVE messages from any other process, the process picks itself as leader.

ing forever! To see this, note that a source may stop contending for leadership voluntarily, when it selects another process as its leader (a non-source contender with a smaller counter). When it does so, the source stops sending ALIVE messages (for communication efficiency). Unfortunately, this triggers processes to timeout on the source and send ACCUSATION messages that cause the source to increment its counters. Later, the accusation counter of the non-source may also increase (due to some legitimate accusations), and then the source may retake the leadership. In this way, the leadership may oscillate between the source and some non-sources forever.

To fix this problem, the source should increment its own accusation counter only if it receives a "legitimate" accusation, i.e., one that was caused by the delay or loss of one of its ALIVE message (and not by the fact that the source voluntarily stopped sending them). To determine whether an accusation is legitimate, each process p keeps track of the number of times it has *voluntarily* given up contending for the leadership in the past—this is its current *phase number*—and it includes this number in each ALIVE message that it sends. If any process q times out on p and wants to accuse p, it must now include its own view of p's current phase number in the ACCUSATION that it sends to p; p considers this accusation to be legitimate only if the phase number that it contains matches its own. Furthermore, whenever p gives up the leadership voluntarily, it increments its own phase number: this causes p to ignore all the spurious accusations that result from its silence.

The Ω algorithm that embodies the above ideas is shown in Figure 2.

Theorem 4 *The algorithm in Figure 2 implements Ω in a system S where all links are fair, and it is communication-efficient.*

5.2 Efficient implementation in system S^+

We now describe a communication-efficient algorithm for Ω for a system S where only the links to and from some unknown correct process are fair, i.e., it works in system S^+. In this system the previous algorithm (Figure 2) does not work because some links can now experience arbitrary message losses.

```
   Code for each process p:
procedure updateLeader()
1  leader ← ℓ such that (counter[ℓ], ℓ) =
              min{(counter[q], q) : q ∈ Contenders}

on initialization:
2  ∀q ≠ p : Timeout[q] ← η + 1
3  ∀q ≠ p : timer(q) ← off
4  ∀q : ph[q] ← 0
5  ∀q : counter[q] ← 0
6  Contenders ← {p}
7  leader ← p
8  start tasks 1 and 2

task 1:
9  loop forever
10   while leader = p do
11     send (ALIVE, counter[p], ph[p]) to every
             process except p every η time
12   ph[p] ← ph[p] + 1
13   while leader ≠ p do nop

task 2:
14 upon receive (ALIVE, d, i) from q do
15   Contenders ← Contenders ∪ {q}
16   counter[q] ← max{counter[q], d}
17   ph[q] ← max{ph[q], i}
18   reset timer(q) to Timeout[q]
19   updateLeader()

20 upon expiration of timer(q) do
21   Contenders ← Contenders − {q}
22   send (ACCUSATION, ph[q]) to q
23   Timeout[q] ← Timeout[q] + 1
24   updateLeader()

25 upon receive (ACCUSATION, i) do
26   if i = ph[p] then
27     counter[p] ← counter[p] + 1
28     updateLeader()
```

Figure 2: Communication-efficient implementation of Ω for a system S where all links are fair.

The most obvious problem, and also the easiest one to solve, is that the ACCUSATION messages sent by a process p to another process q may never reach q: the link $p \rightarrow q$ may be dead. The obvious solution is for p to send each ACCUSATION of q to all processes (including the unknown fair hub); any process that receives such a message relays it once to q. This scheme preserves communication efficiency: after the permanent leader emerges, there are no new accusations, and so the relaying stops.

A more subtle problem, and a tougher one to solve, is that two leader contenders p and q may partition the processes in two sets Π_p and Π_q, such that processes in Π_p (including p) and those in Π_q (including q) have p and q as their permanent leader, respectively. This can occur as follows: (a) the source s and the fair hub h are in Π_p, and they are distinct from p, (b) processes in Π_q receive timely ALIVE messages from q, but they never hear from p, (c) processes in Π_p receive timely ALIVE messages from p, but, except for h, they never hear from q, and (d) h receives timely ALIVE messages from both p and q, but choses p as its permanent leader. In this scenario, nobody ever sends ACCUSATION messages to p or q. Moreover, p and q never hear from each other. So both p and q keep thinking of themselves as the leader, forever.

One attempt to solve this problem is to relay all the ALIVE messages (like the ACCUSATION messages) so that the contenders for leadership, such as p and q in the above scenario, can all hear from each other. Although this solution works, it is not communication-efficient because it forces *all* processes to send messages forever: the elected leader does not stop sending ALIVE messages, and each ALIVE is relayed by all.

To prevent partitioning while preserving communication efficiency, we use the following idea: roughly speaking, if a process r has p as its current leader, but receives an ALIVE message from a process $q \neq p$, then r sends a "CHECK" message telling q about the existence of p (and some other relevant information about p). CHECK messages can be lost, but if: (a) r is the fair hub h, (b) q keeps sending ALIVE messages to h, and (c) h continues to prefer p as its leader, then q will eventually receive a CHECK message from h and find out about its "rival" p. If this happens, q "challenges" the leadership of p by sending an ACCUSATIONs to p if p does not appear to be timely. This scheme prevents the problematic scenario mentioned above, and it can be shown to work while preserving communication efficiency: after the common leader is elected, all the ALIVE messages come from that leader, and so there are no more CHECK messages.

All these ideas are incorporated in the algorithm of Figure 3. Note that ACCUSATION and CHECK messages have an extra field containing the originator of the message (as opposed to the relayer).

Theorem 5 *The algorithm in Figure 3 implements Ω in system S^+, and it is communication-efficient.*

6 Final remarks

In their 2002 PODC tutorial [15], Keidar and Rajsbaum propose several open problems related to the implementation of failure detectors in partially synchronous systems. In particular, they ask what is the "weakest timing model where $\diamond\mathcal{S}$ and/or Ω are implementable but $\diamond\mathcal{P}$ is not". As a partial answer to this question, we note that $\diamond\mathcal{P}$ is *not* implementable in system S. In fact, in the full paper we show that this holds even if we strengthen S by assuming that: (a) all the links in S are reliable (i.e., no message is ever lost), and (b) processes know the identity of the source(s) in S. So S is an example of a system that is strong enough to implement Ω but too weak to implement $\diamond\mathcal{P}$. Similarly, S^+ is strong enough for an *efficient* implementation of Ω, but still too weak for implementing $\diamond\mathcal{P}$. Intuitively, this is because the level of synchrony in S and S^+ is not sufficient to get $\diamond\mathcal{P}$: in both systems only the *output* links of some correct process(es) are eventually timely. Note that if we strengthen the synchrony of S by assuming that *both* the input and output links of some correct process are eventually timely, then $\diamond\mathcal{P}$ becomes implementable [1].

In [15] Keidar and Rajsbaum also ask: "Is building $\diamond\mathcal{P}$ more costly than $\diamond\mathcal{S}$ or Ω?". Concerning this question, note that any implementation of $\diamond\mathcal{P}$ (even in a perfectly synchronous system) requires all alive processes to send messages forever, while Ω can be implemented such that eventually only the leader sends messages (even in a weak system such as S^+).

```
    Code for each process p:
procedure updateLeader()
1   leader ← ℓ such that (counter[ℓ], ℓ) =
              min{(counter[q], q) : q ∈ Contenders}

on initialization:
2   ∀q ≠ p : Timeout[q] ← η + 1
3   ∀q ≠ p : timer(q) ← off
4   ∀q : ph[q] ← 0
5   ∀q : counter[q] ← 0
6   Contenders ← {p}
7   leader ← p
8   start tasks 1 and 2

task 1:
9   loop forever
10     while leader = p do
11        send (ALIVE, counter[p], ph[p]) to every
                process except p every η time
12     ph[p] ← ph[p] + 1
13     while leader ≠ p do nop

task 2:
14  upon receive (ALIVE, d, i) from q do
15     Contenders ← Contenders ∪ {q}
16     counter[q] ← max{counter[q], d}
17     ph[q] ← max{ph[q], i}
18     reset timer(q) to Timeout[q]
19     updateLeader()
20     if q ≠ leader then
21        send (CHECK, leader, ph[leader]) to q

22  upon receive (CHECK, q, i) do
23     if timer(q) is off then
24        ph[q] ← max{ph[q], i}
25        reset timer(q) to Timeout[q]

26  upon expiration of timer(q) do
27     Contenders ← Contenders − {q}
28     send (ACCUSATION, q, ph[q]) to every
                process except p
29     Timeout[q] ← Timeout[q] + 1
30     updateLeader()

31  upon receive (ACCUSATION, q, i) do
32     if q = p then
33        if i = ph[p] then
34           counter[p] ← counter[p] + 1
35           updateLeader()
36     else send (ACCUSATION, q, i) to q
```

Figure 3: Communication-efficient implementation of Ω for system S^+.

Finally, it is also worth pointing out that the above results provide an alternative proof that $\diamond\mathcal{P}$ is *strictly* stronger than $\diamond\mathcal{S}$[14]: this can be deduced from the fact that Ω (and hence $\diamond\mathcal{S}$) is implementable in system S but $\diamond\mathcal{P}$ is not.

References

[1] M. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Stable leader election (extended abstract). In *Proceedings of the 15th International Symposium on Distributed Computing*, LNCS 2180, pages 108–122. Springer-Verlag, 2001.

[2] M. Bertier, O. Marin, and P. Sens. Implementation and performance evaluation of an adaptable failure detector. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, June 2002.

[3] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, Nov. 2002.

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

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

[6] W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. *IEEE Transactions on computers*, 1(51):13–32, Jan. 2002.

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

[8] R. De Prisco, B. Lampson, and N. A. Lynch. Revisiting the Paxos algorithm. In *Proceedings of the 11th Workshop on Distributed Algorithms*, LNCS 1320, pages 11–125. Springer-Verlag, Sept. 1997.

[9] B. Deianov and S. Toueg. Failure detector service for dependable computing. In *Proceedings of the 2000 International Conference on Dependable Systems and Networks (ICDSN/FTCS-30)*, pages B14–B15. IEEE computer society press, June 2000.

[10] C. Dwork, N. A. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, Apr. 1988.

[11] C. Fetzer, M. Raynal, and F. Tronel. A failure detection protocol based on a lazy approach. Research Report 1367, IRISA, Nov. 2000.

[12] E. Gafni and L. Lamport. Disk paxos. In *Proceedings of the 14th International Symposium on Distributed Computing*, LNCS 1914, pages 330–344. Springer-Verlag, 2000.

[13] R. Guerraoui and P. Dutta. Fast indulgent consensus with zero degradation. In *Proceedings of the 4th European Dependable Computing Conference*, Oct. 2002.

[14] V. Hadzilacos, 2002. Comparison between $\diamond S$ and $\diamond P$, personal communication.

[15] I. Keidar and S. Rajsbaum. On the cost of fault-tolerant consensus when there are no faults-a tutorial. In *Tutorial 21th ACM Symposium on Principles of Distributed Computing(http://theory.lcs.mit.edu/ idish/ftp/podc02-tutorial.ppt)*, July 2002.

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

[17] L. Lamport. Paxos made simple. *SIGACT News*, 32(4):18–25, Dec. 2001.

[18] M. Larrea, S. Arévalo, and A. Fernández. Efficient algorithms to implement unreliable failure detectors in partially synchronous systems. In *Proceedings of the 13th International Symposium on Distributed Algorithms*, LNCS 1693, pages 34–48, Sept. 1999.

[19] M. Larrea, A. Fernández, and S. Arévalo. Optimal implementation of the weakest failure detector for solving consensus. In *Proceedings of the 19th Symposium on Reliable Distributed Systems*, pages 52–59. IEEE Computer Society Press, Oct. 2000.

[20] M. Larrea, A. Fernandez, and S. Arevalo. Eventually consistent failure detectors. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 326–327, 2001.

[21] A. Mostéfaoui and M. Raynal. Leader-based consensus. *Parallel Processing Letters*, 11(1):95–107, 2001.

[22] R. van Renesse, Y. Minsky, and M. M. Hayden. A gossip-based failure detection service. In *Proceedings of Middleware'98*, Sept. 1998.

Optional appendices

A Proof of Theorem 1

We now show correctness of the algorithm in Figure 1. Let s be an eventually timely source in S. Since all the output links of s are eventually timely (and the rate of process execution is bounded) there is a constant Δ and a time after which every message sent by s takes at most Δ time to be received and processed.

Lemma 6 *For every process p, we always have $p \in set1_p$ and $p \in set2_p$.*

Proof. Every process p initializes its sets $set1_p$ and $set2_p$ to $\{p\}$. Since p does not set timers for itself, it never removes itself from these sets. □

Lemma 7 *For every correct process q, there is a time after which $s \in set1_q$ and $s \in set2_q$. Furthermore, $counter_s[s]$ is bounded.*

Proof. By Lemma 6, $s \in set1_s$ and $s \in set2_s$. Now consider any correct process $q \neq s$. Process s sends an $(\text{ALIVE}, s, *)$ to q every η time. Since s is an eventually timely source, there is a time after which every $(\text{ALIVE}, s, *)$ that s sends is directly received by q, and it is received within $\eta + \Delta$ time from the time q received the previous $(\text{ALIVE}, s, *)$ from s. Since q increases its timers $timeout1(s)$ and $timeout2(s)$ every time they expire, there is a time after which they will cease expiring. Thenceforth, $s \in set1_q$ and $s \in set2_q$, and q ceases to send ACCUSATION messages to s. Since eventually all correct processes stop sending ACCUSATION messages to s, $counter_s[s]$ is bounded. □

Lemma 8 *For every correct process p, if $counter_p[p]$ is bounded then there exists a time after which $p \in set1_s$.*

Proof. By Lemma 6, the lemma obviously holds for the case that $p = s$. Now consider a correct process $p \neq s$. We prove the contrapositive of the lemma. Suppose that $p \notin set1_s$ infinitely often. There are two possible cases. (A) Process p is added to and removed from $set1_s$ infinitely often. In this case, every time s removes p from $set1_s$, s sends an ACCUSATION message to p, and so s sends ACCUSATION messages to p infinitely often. (B) There is a time after which p is never in $set1_s$. In this case, there is a time after which s never receives $(\text{ALIVE}, p, *)$ messages directly from p. Thus $timer1(p)$ expires infinitely often at s, and so s sends ACCUSATION messages to p infinitely often. Since s is a source, the link from s to p is eventually timely. Thus, in both cases (A) and (B), p receives ACCUSATION messages from s infinitely often, and p increments $counter_p[p]$ infinitely often, so $counter_p[p]$ is not bounded. □

Lemma 9 *For every correct process p, if there exists a time after which $p \in set1_s$, then there exists a time after which for every correct process q, $p \in set2_q$.*

Proof. Let p be a correct process. If $p = s$, then, by Lemma 7, eventually $p \in set2_q$ for every correct process q. Now assume $p \neq s$. Suppose that there exists a time after which $p \in set1_s$. Thus, there is a time after which $timer1(p)$ at s never expires. This implies there is some time interval ζ and a time after which s periodically receives $(\text{ALIVE}, p, *)$ messages, directly from p, at least once every ζ time. So there is a time after which $p \in set2_s$. Moreover, every time s receives a $(\text{ALIVE}, p, *)$ message directly from p, it sends $(\text{ALIVE}, p, *)$ to all processes q except for s and p. Therefore, eventually every correct process q such that $q \neq p$ and $q \neq s$ receives $(\text{ALIVE}, p, *)$ from s at least once every $\zeta + \Delta$ time. So, there exists a time after which for every correct process q such that $q \neq p$, $p \in set2_q$. For the case $q = p$, note that by Lemma 6, $q \in set2_q$ (always) and this concludes the proof. □

Henceforth, var_p^t denotes the value of the local variable var of p at time t.

Lemma 10 *For every process p and every correct process q, either there is a time after which $p \notin set2_q$ or for every time t, there is a time after which $counter_q[p] \geq counter_p^t[p]$.*

Proof. For $p = q$, the lemma is trivial. Now assume $p \neq q$ and suppose that $p \in set2_q$ infinitely often. Thus, q must receive messages of type $(\text{ALIVE}, p, counter[p])$ infinitely often. Let t be any time. There must be a time $t' > t$ when q receives

(ALIVE, p, c) originally sent by p after time t, so $c \geq counter_p^t[p]$. Then at time t', q sets its $counter_q[p]$ to c, and so we have: $counter_q[p] \geq counter_p^t[p]$. The lemma now follows since $counter_q[p]$ is monotonically nondecreasing. □

Lemma 11 *For every correct process p:*

1. *If $counter_p[p]$ is bounded, then there exists a value v_p and a time after which for every correct process q, $p \in set2_q$ and $counter_q[p] = v_p$.*
2. *If $counter_p[p]$ is not bounded, then for every correct process q, either $counter_q[p]$ is not bounded or there is a time after which $p \notin set2_q$.*

Proof. Let p be a correct process.
(1) Suppose $counter_p[p]$ is bounded. Then, by Lemma 8, there exists a time after which $p \in set1_s$. Therefore, by Lemma 9, there exists a time after which for every correct process q, $p \in set2_q$. Thus, by Lemma 10, for all correct processes q, for all t there exists a time after which $counter_q[p] \geq counter_p^t[p]$. Since $counter_p[p]$ is bounded and monotonically nondecreasing, there exists a value v_p and a time after which $counter_p[p] = v_p$. Moreover, it is always true that $counter_p[p] \geq counter_q[p]$. Therefore, there exists a time after which, for all correct processes q, $counter_q[p] = v_p$.
(2) Suppose $counter_p[p]$ is not bounded. Let q be any correct process. Either there is a time after which $p \notin set2_q$, or $p \in set2_q$ infinitely often. In the latter case, Lemma 10 implies that $counter_q[p]$ is also not bounded. □

Lemma 12 *If process p is not correct then for every correct process q there is a time after which $p \notin set2_q$.*

Proof. After p crashes, it stops sending ALIVE messages. So there is a time after which no process receives $(\text{ALIVE}, p, \dots)$ from any process. Thus, for every correct process q, there is a time after which $p \notin set2_q$. □

Lemma 13 *There exists a correct process ℓ and a time after which, for every correct process q, $leader_q = \ell$.*

Proof. Let B be the set of correct processes p such that $counter_p[p]$ is bounded. By Lemma 7, $s \in B$, thus B is not empty. By Lemma 11(1), for every process $p \in B$, there is a corresponding integer v_p and a time after which for every correct process q, $p \in set2_q$ and $counter_q[p] = v_p$ (forever). Let ℓ denote the process p in B with the smallest corresponding tuple (v_p, p). We now show that eventually every correct process q selects ℓ as its leader (forever).

Note that correct process q selects its leader from the set $set2_q$ (by Lemma 6, this set is never empty). Since there is a time after which $\ell \in set2_q$, eventually ℓ is a permanent candidate for leadership at q. We now show that for any other process $p \neq \ell$: (*) there is a time after which either $p \notin set2_q$ or $(counter_q[p], p) > (counter_q[\ell], \ell)$. This implies that eventually q selects ℓ as its leader, forever.

To show (*) holds, consider the following 3 possible cases. If p is not correct then, by Lemma 12, eventually $p \notin set2_q$ (forever). Now suppose that p is correct. If $counter_p[p]$ is bounded, then p is in B; so, by our selection of ℓ in B, eventually $(counter_q[p] = v_p, p) > (counter_q[\ell] = v_\ell, \ell)$ forever. Finally, if $counter_p[p]$ is not bounded, then, by Lemma 11(2), there is a time after which $p \notin set2_q$, or $counter_q[p] > counter_q[\ell] = v_\ell$ (because $counter_q[p]$ is unbounded and monotonically nondecreasing). In all cases (*) holds. □

From the above, Theorem 1 follows.

B Proof of Theorem 2

The theorem trivially holds if $n = 1$. Henceforth, we assume that $n \geq 2$. Consider any algorithm that implements Ω in a system S where at most one process may crash. We first observe the following:

Fact 14 *For any run and any correct process p, if there is a time after which p does not receive any messages from other processes, then there is a time after which the leader of p is p (forever).*

To see this, consider a run R such that after some time t, some correct process p does not receive any messages. Without loss of generality, we can assume that no process crashes in R (because if any process f crashes at some time t' in R, we can modify R to

get a similar run where f never crashes, but all its outgoing links die permanently at time t'; this modified run is indistinguishable from R to all processes, except for process f who is now correct). Suppose, by contradiction, that in R there is a time after which the leader of p is a process $q \neq p$. Let R' be a run identical to R up to time t, and such that at some time $t' > t$: (a) process q crashes, and (b) all the *input* links of p "die" permanently, while the *output* links of p become timely and stop losing messages (p is a source). Process p receives exactly the same messages in R and R'. Since p cannot distinguish between R and R', in R' there is a time after which the leader of p is q, even though q crashes—a contradiction that concludes the proof of Fact 14.

We now prove part (1) of the theorem. Suppose, by contradiction, that there is a run R such that two correct processes p and q do not send any messages after some time t. Without loss of generality, we can assume that in R: (a) all the output links of p and q are *eventually* timely (and so both p and q are sources), and (b) no process crashes (the argument is as before: we can "replace" the crash of a process, by the simultaneous and permanent failure of all its outgoing links).

We first show that in R there is a time after which the leader of q is not p. To see this, let R' be a run identical to R except that p crashes in R' after time t. Note that, except for p, no process can distinguish between runs R and R'. Since p is faulty in R', in R' there is a time after which the leader of q is not p; thus, in R there is a time after which the leader of q is not p.

Now let R'' be a run identical to R, except that in R'' after time t: (1) all the output links of p die permanently, and (2) all the input links of p die permanently, except for the link from q to p (which, as in run R, is eventually timely). Note that, except for p, no process can distinguish between runs R and R''. Thus, in R'' there is a time after which the leader of q is not p (as it was the case in run R). In R'', p ceases to receive messages, and so, by Fact 14, there is a time after which the leader of p is p. Thus, in R'' correct processes p and q do not reach agreement on a common leader—a contradiction that concludes the proof of the first part of the theorem.

We now prove part (2) of the theorem. Partition the set of processes of S into set A with $\lceil \frac{n}{2} \rceil$ processes, and set B with $\lfloor \frac{n}{2} \rfloor$ processes. Consider run R such that: (a) all the n processes are correct, (b) all the links between processes in A are eventually timely, (c) A has a source s, so all the links from s to processes in B are eventually timely, (d) for every process $r \neq s$ in A, all the links from r to processes in B are permanently dead, and (e) the output links from every process in B are permanently dead. So in run R, s is the only process that is able to communicate with any process $p \in B$; all messages sent by other processes to p are lost.

Note that in run R there is a time after which the leader of any correct process q is not p. Intuitively, this is because p may eventually crash, and since p's output links are permanently dead, q would not be able to notice this crash (we omit this proof as it is similar to one given above).

We claim that in R, every process in A sends messages forever to every process in B. Suppose, for contradiction, that in R some process $q \in A$ does not send messages forever to some process $p \in B$. We consider two possible cases.

Suppose $q = s$. Recall that in R, q ($= s$) is the only process able to communicate with p. Since in R there is a time after which q does not send messages to p, then eventually p stops receiving messages. So, by Fact 14, in R there is a time after which the leader of p is p. Recall that in R there is a time after which the leader of q is *not* p. Thus, in run R correct processes p and q do not reach agreement on a common leader—a contradiction.

Now suppose $q \neq s$. Let R' be a run which is similar to R, except that the source happens to be q rather than s. More precisely, R' is like R, except that all the links from s to processes in B are permanently dead, and all the links from q to processes in B are eventually timely. Since no process in B can communicate (their output links are permanently dead in both R and R'), processes in A cannot distinguish between runs R and R'. Thus, in R' (as in R) there is a time after which: (a) the leader of q is not p, and (b) q does not send messages to p. Since the link from q to p is the only input link of p that is not permanently dead in R', then there is a time after which p does not receive any message in R'. So, by Fact 14,

in R' there is a time after which the leader of p is p. Thus, in R' correct processes p and q do not reach agreement on a common leader—a contradiction.

Thus we proved our claim that in run R every process in A sends messages forever to every process in B. Since $|A| = \lceil \frac{n}{2} \rceil$ and $|B| = \lfloor \frac{n}{2} \rfloor$, this implies that at least $\lceil \frac{n}{2} \rceil \cdot \lfloor \frac{n}{2} \rfloor = \lceil \frac{(n^2-1)}{4} \rceil$ links carry messages forever in run R.

C Proof of Theorem 4

We now show correctness and communication efficiency of the algorithm in Figure 2. Let s be an eventually timely source. Since all the output links of s are eventually timely there exists a value Δ and a time T_0 after which every message sent by s takes at most Δ time to be received and processed.

Lemma 15 *For every process p, we always have $p \in Contenders_p$.*

Proof. Every process p initializes its set $Contenders_p$ to $\{p\}$. Since p does not set a timer for itself, it never removes itself from this set. □

Definition 16 *When a process p receives a message $(\text{ACCUSATION}, i)$ from q, we say the message is up-to-date if $i = ph_p[p]$.*

Note that if p receives an ACCUSATION message that is not up-to-date, it will ignore it.

Observation 17 *For every processes p and q, $counter_p[q]$ and $ph_p[q]$ are monotonically nondecreasing with time.*

Proof. Clear from the way $counter_p[q]$ and $ph_p[q]$ are updated. □

Lemma 18 *For every process p and every correct process q, either (1) there is a time after which $p \notin Contenders_q$ or (2) for every time t, there is a time after which $counter_q[p] \geq counter_p^t[p]$ and $ph_q[p] \geq ph_p^t[p]$. Moreover, if p is faulty then (1) holds.*

Proof. If $p = q$, condition (2) holds because $counter_p[p]$ and $ph_p[p]$ are monotonically nondecreasing. Now assume $p \neq q$. If (1) does not hold then p is in $Contenders_q$ infinitely often so q receives ALIVE messages from p infinitely often. Thus, at some time t', q receives one ALIVE message sent after time t. Since $counter_p[p]$ and $ph_p[p]$ are monotonically nondecreasing, the $counter$ and ph values in that message are at least as great as $counter_p^t[p]$ and $ph_p^t[p]$, respectively. Thus $counter_q^{t'}[p] \geq counter_p^t[p]$ and $ph_q^{t'}[p] \geq ph_p^t[p]$. Condition (2) now follows since $counter_q[p]$ and $ph_q[p]$ are monotonically nondecreasing. Finally, note that if p is faulty then there is a time after which q never receives an ALIVE message from p. Soon after, q removes p from $Contenders_q$ and never adds it back again. □

Lemma 19 *$counter_s[s]$ is bounded.*

Proof. To obtain a contradiction, suppose that $counter_s[s]$ grows unboundedly. Then s receives infinitely many up-to-date ACCUSATIONS. Note that an up-to-date ACCUSATION can only be received by s if s is looping in lines 10–11: else, s has increased $ph_s[s]$ to a value greater than anything any process has ever received. We can find a process p that infinitely often sends an up-to-date ACCUSATION to s and increments $Timeout_p[s]$. So $Timeout_p[s]$ grows to infinity. We now get to a contradiction by showing that if $Timeout_p[s] > \eta + \Delta$ at some time $t > T_0$ then p does not send an up-to-date ACCUSATION to s at time $t + \eta + \Delta$. Indeed, suppose it did, and let x be the value of $ph_s[s]$ at time $t + \eta + \Delta$. Then, (*) p did not receive an ALIVE message from s during times $[t, t + \eta + \Delta]$. Moreover, (**) p must have received $(\text{ALIVE}, \ldots, x)$ from s before time $t + \eta + \Delta$ (else it would not send an up-to-date ACCUSATION at time $t + \eta + \Delta$). By (*) and (**), p must have received $(\text{ALIVE}, \ldots, x)$ from s before time t. Therefore s sends such a message before time t—say at a time t_0. Thus, from time t_0, s continues looping in lines 10–11 until at least time $t + \eta + \Delta$. Therefore, s sends $(\text{ALIVE}, \ldots, x)$ during time $[t, t + \eta]$. Since s is a timely source and $t > T_0$, some ALIVE message is received and processed by p during times $[t, t + \eta + \Delta]$—a contradiction. □

Definition 20 *Let L_p be the largest value of $counter_p[p]$ in the execution (or ∞ if $counter_p[p]$ is unbounded). Let ℓ to be the correct process with*

the smallest L_p (break ties by process id) and let C be its value of L_p.

Note that by Lemma 19, $C < \infty$.

Lemma 21 *There is a time after which $leader_\ell = \ell$.*

Proof. By Lemma 15, note that ℓ will pick itself as leader as long as it does not find another process $p \neq \ell$ in $Contenders_\ell$ with smaller $counter_\ell[p]$. So consider a process $p \neq \ell$. By Lemma 18 and definition of C, either there is a time after which $p \notin Contenders_\ell$ or $counter_\ell[p]$ becomes larger than C (breaking ties using process id). □

Corollary 22 *There is a time after which $ph_\ell[\ell]$ stops changing.*

Proof. Indeed, $ph_\ell[\ell]$ can only change when ℓ relinquishes leadership, which can only happen a finite number of times by Lemma 21. □

Definition 23 *Let $\ell phase$ be the final value of $ph_\ell[\ell]$.*

Note that since $ph_\ell[\ell]$ is monotonically nondecreasing, $\ell phase$ is also the largest value of $ph_\ell[\ell]$.

Lemma 24 *For every correct process p there is a time after which $\ell \in Contenders_p$.*

Proof. By Lemma 21 and the definition of C and $\ell phase$, note that ℓ sends $(\text{ALIVE}, C, \ell phase)$ infinitely often. Hence, since links are fair, p receives $(\text{ALIVE}, C, \ell phase)$ from ℓ infinitely often. We claim that p can only remove ℓ from $Contenders_p$ finitely often, which immediately implies the lemma. We show the claim by contradiction: if p removes ℓ from $Contenders_p$ infinitely often, then p sends $(\text{ACCUSATION}, \ell phase)$ messages to ℓ infinitely often. Since links are fair, ℓ receives $(\text{ACCUSATION}, \ell phase)$ infinitely often, and so it eventually increments $counter_\ell[\ell]$ to a value greater than C—a contradiction. □

Lemma 25 *There is a time after which for every correct process p, $leader_p = \ell$.*

Proof. By Lemmas 18 and 24 and the fact that $counter_\ell[\ell]$ eventually always equals C, we conclude that there is a time after which $counter_p[\ell] = C$ at every correct process p. By Lemma 24, there is a time after which $\ell \in Contenders_p$, so that p picks ℓ as leader as long as there no other process $q \neq \ell$ in $Contenders_p$ with smaller $counter_p[q]$. Consider such a process $q \neq \ell$. By Lemma 18 either (1) there is a time after which $q \notin Contenders_p$ or (2) $counter_p[q]$ becomes larger than C (breaking ties using process id). □

Lemma 26 *There is a time after which only ℓ sends messages.*

Proof. There are only two types of messages: ALIVE and ACCUSATION. ALIVE messages are only sent by a process if it thinks itself is the leader, so by Lemma 25, (*) there is a time after which only ℓ sends ALIVE messages. We now claim that there is a finite number of ACCUSATION messages sent. Indeed, an ACCUSATION message is only sent to p if $timer(p)$ is started, which can only happen if $(\text{ALIVE}, \dots)$ is received from p. Thus, because of (*), there is a time after which the only ACCUSATION messages sent are sent to ℓ. When a process q sends $(\text{ACCUSATION}, \dots)$ to ℓ it removes ℓ from $Contenders_q$ and so, by Lemma 24, this can only happen finitely often. □

From Lemmas 25 and 26, Theorem 4 follows.

D Proof of Theorem 5

We now show correctness and communication efficiency of the algorithm in Figure 3. Let s be an eventually timely source and let h be a fair hub. Since all the output links of s are eventually timely there exists a value Δ and a time T_0 after which every message sent by s takes at most Δ time to be received and processed.

Lemma 27 *For every process p, we always have $p \in Contenders_p$.*

Proof. Identical to the proof of Lemma 15. □

Definition 28 *When a process p receives a message $(\text{ACCUSATION}, p, i)$ from q, we say the message is up-to-date if $i = ph_p[p]$.*

Observation 29 *For every processes p and q, $counter_p[q]$ and $ph_p[q]$ are monotonically nondecreasing with time.*

Proof. Clear from the way $counter_p[q]$ and $ph_p[q]$ are updated. □

Lemma 30 *For every process p and every correct process q, either (1) there is a time after which $p \notin Contenders_q$ or (2) for every time t, there is a time after which $counter_q[p] \geq counter_p^t[p]$ and $ph_q[p] \geq ph_p^t[p]$. Moreover, if p is faulty then (1) holds.*

Proof. Identical to the proof of Lemma 18. □

Lemma 31 *$counter_s[s]$ is bounded.*

Proof. (Similar to the proof of Lemma 19) To obtain a contradiction, suppose that $counter_s[s]$ grows unboundedly. Then s receives infinitely many up-to-date $(\text{ACCUSATION}, s, \dots)$ messages. Note that an up-to-date ACCUSATION can only be received by s if s is looping in lines 10–11: else, s has increased $ph_s[s]$ to a value greater than anything any process has ever received. We can find a process p that infinitely often sends an up-to-date ACCUSATION to s and increments $Timeout_p[s]$. So $Timeout_p[s]$ grows to infinity. We now get to a contradiction by showing that if $Timeout_p[s] > \eta + \Delta$ at some time $t > T_0$ then p does not originate an up-to-date ACCUSATION to s at time $t + \eta + \Delta$. Indeed, suppose it did, and let x be the value of $ph_s[s]$ at time $t + \eta + \Delta$. Then (1) p did not receive an ALIVE message during times $[t, t + \eta + \Delta]$ and (2) if p receives a CHECK message during $[t, t + \eta + \Delta]$ then $timer(s)$ is on at the receipt time. Moreover, (3) before time $t + \eta + \Delta$, p must either (a) have received $(\text{ALIVE}, \dots, x)$ from s or (b) have received (CHECK, s, x) while $timer(s)$ is off: indeed, if (3) did not hold then p would not send an up-to-date ACCUSATION at time $t + \eta + \Delta$. By (1), (2) and (3), p must have received before time t either $(\text{ALIVE}, \dots, x)$ from s or (CHECK, s, x). In either case, s sends $(\text{ALIVE}, \dots, x)$ at some time $t_0 < t$ (a small induction argument shows that a process can only send (CHECK, s, x) if s previously sent $(\text{ALIVE}, \dots, x)$). From time t_0, s continues looping in lines 10–11 until at least time $t + \eta + \Delta$. Therefore, s sends $(\text{ALIVE}, \dots, x)$ during time $[t, t + \eta]$. Since s is a timely source and $t > T_0$, some ALIVE message is received and processed by p during times $[t, t + \eta + \Delta]$—a contradiction. □

Definition 32 *Let L_p be the largest value of $counter_p[p]$ in the execution (or ∞ if $counter_p[p]$ is unbounded). Let ℓ to be the correct process with the smallest L_p (break ties by process id) and let C be its value of L_p.*

Note that by Lemma 31, $C < \infty$.

Lemma 33 *There is a time after which $leader_\ell = \ell$.*

Proof. Identical to the proof of Lemma 21. □

Corollary 34 *There is a time after which $ph_\ell[\ell]$ stops changing.*

Proof. Identical to the proof of Corollary 22. □

Definition 35 *Let $\ell phase$ be the final value of $ph_\ell[\ell]$.*

Note that since $ph_\ell[\ell]$ is monotonically nondecreasing, $\ell phase$ is also the largest value of $ph_\ell[\ell]$.

Lemma 36 *No process sends $(\text{ACCUSATION}, \ell, \ell phase)$ messages infinitely often.*

Proof. To obtain a contradiction, suppose that some process p sends infinitely many $(\text{ACCUSATION}, \ell, \ell phase)$ messages. Of those, infinitely many get relayed through h and reach ℓ, since h is a fair hub. Therefore ℓ eventually increments $counter_\ell[\ell]$ to a value greater than C—a contradiction. □

Lemma 37 *There is a time after which $leader_h = \ell$ and $ph_h[\ell] = \ell phase$.*

Proof. If $h = \ell$ the result follows from Lemma 33 and the definition of $\ell phase$. Now assume $h \neq \ell$. By Lemma 33 and the fact that h is a fair hub, h receives an infinite number of ALIVE messages from ℓ. It follows that (*) there is a time after which $ph_h[\ell] = \ell phase$. Moreover, ℓ is added to $Contenders_h$ infinitely often. By Lemma 30 and the definition of

ℓ and C, there is a time after which for every process $q \neq \ell$, either (a) $q \notin Contenders_h$, or (b) $(counter_h[q], q) > (C, \ell)$. Therefore h chooses ℓ as leader infinitely often. We claim that h removes ℓ from $Contenders_h$ only finitely often, and so the lemma follows. We show the claim by contradiction. Suppose that h removes ℓ from $Contenders_h$ infinitely often. Then, h sends ACCUSATION messages to ℓ infinitely often. By (*), infinitely many such messages are $(\text{ACCUSATION}, \ell, \ell phase)$. This violates Lemma 36. □

Lemma 38 *There is a time after which only ℓ sends* ALIVE *messages.*

Proof. By contradiction, suppose some process $p \neq \ell$ sends ALIVE messages forever. We first claim that $p \neq h$. Indeed, if $p = h$ then there are two cases: $h = \ell$—which contradicts $p \neq \ell$—and $h \neq \ell$. In the latter case, we get a contradiction by Lemma 37 and the fact that h can only send $(\text{ALIVE}, \dots)$ if h thinks itself as the leader. This shows the claim that $p \neq h$. We now claim that (*) p only receives finitely many ALIVE messages from ℓ. Indeed, from the definition of ℓ, there is a time after which p can only consider itself as leader if ℓ is not in $Contenders_p$. So, if p receives infinitely many ALIVE messages from ℓ, then infinitely many of those have phase equal to $\ell phase$. So p adds ℓ to $Contenders_p$ infinitely often, and so p removes ℓ from $Contenders_p$ infinitely often, and so p sends $(\text{ACCUSATION}, \ell, \ell phase)$ to ℓ infinitely often. This violates Lemma 36—a contradiction that shows (*). Now since p sends ALIVE messages infinitely often, h receives such messages infinitely often. By Lemma 37, there is a time after which ℓ is the leader of h. After that time, each time h receives ALIVE from p, h sends $(\text{CHECK}, \ell, \dots)$ to p. Since h is a fair hub, p receives such messages infinitely often. An infinite number of such messages contains a phase equal to $\ell phase$ (because there is a time after which $ph_h[\ell] = \ell phase$ by Lemma 37). Therefore (**) there is a time after which $ph_p[\ell] = \ell phase$. Furthermore, the CHECK messages ensure that p starts a timer on ℓ infinitely often. Then, because of (*), p times out on ℓ infinitely often and sends infinitely many ACCUSATIONS to ℓ. Infinitely many of those have phase $\ell phase$ due to (**). This violates Lemma 36—a contradiction. □

Lemma 39 *There is a time after which for every correct process p, $leader_p = \ell$.*

Proof. By Lemma 38, there is a time after which for every correct process p, only p and ℓ can be in $Contenders_p$ (p is in its own $Contenders_p$ by Lemma 27, but p needs to receive an ALIVE message from $q \neq p$ for q to be in $Contenders_p$). Hence only p and ℓ can be leaders at p. We now claim it is impossible for leadership to switch between p and ℓ infinitely often. Indeed, by definition of ℓ, there is a time after which p can choose itself as leader only if $\ell \notin Contenders_p$. Hence if leadership switches between p and ℓ infinitely often then ℓ is added to and removed from $Contenders_p$ infinitely often, and so (1) p eventually receives an ALIVE message from ℓ with phase $\ell phase$ and so there is a time after which $ph_p[\ell] = \ell phase$, and (2) p sends an infinite number of ACCUSATIONS to ℓ, and infinitely many have phase equal to $\ell phase$. This violates Lemma 36—a contradiction that shows the claim. Therefore there is a time after which either ℓ or p is always the leader at p. But p cannot always be the leader at p else p sends ALIVE messages infinitely often, contradicting Lemma 38. □

Lemma 40 *There is a time after which only ℓ sends messages.*

Proof. There are three types of messages: ALIVE, CHECK and ACCUSATION. By Lemma 38, (*) there is a time after which only ℓ sends ALIVE messages. By Lemma 39, this implies that (**) there is a time after which no CHECK messages are sent: indeed, such messages are only sent when a process p receives an ALIVE message from q and q is not p's leader. We now claim that there is a finite number of ACCUSATIONS sent. Indeed, an $(\text{ACCUSATION}, p, \dots)$ message is sent either when *timer*(p) expires, or when it is relayed in line 36. The message can be relayed at most once per process. Now *timer*(p) expires only if it is started, which can happen only if an ALIVE message is received from p or if a $(\text{CHECK}, p, \dots)$ message is received. Thus, because of (*) and (**), there is a time after which the only ACCUSATION messages sent are

$(\text{ACCUSATION}, \ell, \dots)$. If this message is relayed in line 36, then some process previously sent it in line 28. When a process q sends it in line 28, q removes ℓ from $Contenders_q$, which can only happen finitely often by Lemma 39 (note that q can only have ℓ as leader while $\ell \in Contenders_q$). □

From Lemmas 39 and 40, Theorem 5 follows.