HAL
open science

On implementing Omega in systems with weak reliability and synchrony a ssumptions

Marcos Kawazoe Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, Sam Toueg

To cite this version:

Marcos Kawazoe Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, Sam Toueg. On implementing Omega in systems with weak reliability and synchrony a ssumptions. 2007. hal-00259018

HAL Id: hal-00259018

https://hal.science/hal-00259018v1

Preprint submitted on 26 Feb 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

On implementing Omega in systems with weak reliability and synchrony assumptions

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

November 26, 2007

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 (a) except for some unknown timely process s, all processes may be arbitrarily slow or may crash, and (b) only the output links of s are eventually timely (all other links can be arbitrarily slow and lossy). Previously known algorithms for Ω worked only in systems that are strictly stronger than S in terms of reliability or synchrony assumptions.

We next show that algorithms that implement Ω in system S are necessarily expensive in terms of communication complexity: all correct processes (except possibly one) must send messages forever; moreover, a quadratic number of links must carry messages forever. This result holds even for algorithms that tolerate at most one crash.

Finally, we show that with a small additional assumption to system S — the existence of some unknown correct process whose links can be arbitrarily slow and lossy but fair — there is a communication-efficient algorithm for Ω such that eventually only *one* process (the elected leader) sends messages.

Some recent experimental results indicate that two of the algorithms for Ω described in this paper can be used in dynamically-changing systems and work well in practice [ST07].

[1]HP Laboratories, 1501 Page Mill Road, Palo Alto, CA, 94304, USA, marcos.aguilera@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, Canada, sam@cs.toronto.edu

1 Introduction

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 [vRMH98, LAF99, DT00, LFA00, LFA01, ADGFT01, FRT01, CTA02, BMS02, MMR03, ADGFT04, MOZ05, HMSZ05, FR06].

One failure detector of particular interest is Ω [CHT96]. Roughly speaking, with Ω every process p has a local variable, denoted $leader_p$, 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 is the weakest failure detector for solving consensus and consensus-like problems such as atomic broadcast [CHT96], and it is at the core of several consensus algorithms that are used in practice [Lam98, GL03, CGR07]. It is also used in the solution of other problems, such as non-blocking atomic commit [DGFG$^+$04], In this paper, we study the problem of implementing Ω in systems with weak reliability and synchrony assumptions. We also investigate in which systems such implementations can be communication-effi cient.

Our starting point is a system where (a) all processes can be arbitrarily slow and crash, but they have a maximum execution speed, and (b) all links can be arbitrarily slow and lossy. 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 S^-.

Thus, we consider a system that is slightly stronger than S^-, namely a system S^- with the following additional assumption: there is at least *one* process that is timely and whose *output* links are eventually timely. Roughly speaking, this means that the process has a minimum execution speed, and there is a bound δ and a time after which every message sent from that process is delivered within δ time. We call such a process an *eventually timely source*, and we denote by S a system S^- with at least one eventually timely source. Note that in system S processes do *not* know the identity of the eventually timely source(s), the time after which the output links of the eventually timely source(s) become timely, or the corresponding bounds on message delivery time.

S is a very weak type of partially synchronous system in terms of the timeliness of processes and the timeliness and reliability of links. In S, only the links *from* the eventually timely source(s) are reliable; all other links, including those *to* the eventually timely source(s), can drop messages arbitrarily. Thus, processes cannot use eventually timely sources as "forwarding nodes" to communicate reliably with each other. Moreover, in S, the timeliness assumptions apply only to the unknown eventually timely source(s) and their output links. All other processes and links can be arbitrarily slow.

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 such an agreement when some processes cannot even communicate, as it may happen in system S. For example, consider a system S with 5 processes, denoted s_1, s_2, s_3, p and q, that behaves as follows (see Figure 1): (a) all the processes are correct and timely, (b) all the output links of p and q are lossy and drop every message that p and q send (hence p and q cannot communicate at all), (c) all the output links of s_2 are timely, i.e., they are reliable and deliver all the messages sent by s_2 in a timely way (so s_2 is a an eventually timely source), (d) all the output links of s_1 are timely, except for the link from s_1 to q which loses all messages, and (e) all the output links of s_3 are timely, except for the link from s_3 to p which loses all messages. Note that for process p, the natural leader candidates are the two processes from which it gets timely messages, namely s_1 and s_2. Symmetrically, for q the natural

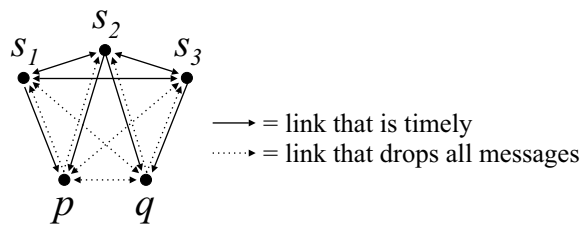

Figure 1: Processes p and q cannot communicate but must agree on the leader among s_1, s_2 and s_3.

leader candidates are s_2 and s_3. Any implementation of Ω must ensure that p and q eventually agree on the same leader —a non-trivial 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. Previously known implementations of Ω in partially synchronous systems require stronger reliability or synchrony assumptions [Lam98, PLL00, LFA00, ADGFT01]. In fact, these implementations assume systems that are strong enough to support the implementation of the *eventually perfect failure detector* $\diamond\mathcal{P}$.[1] In contrast, it is easy to see that S is too weak for implementing $\diamond\mathcal{P}$.

Our algorithm that implements Ω in system S, however, has a serious drawback: *all* the processes periodically send messages forever. This communication overhead is undesirable, and a natural question is whether it can be avoided. Intuitively, after a process becomes the common leader,[2] it must periodically send messages forever (because if it crashes, processes must be able to notice this failure and choose a new leader); but thereafter 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 motivates the following definition and leads us to 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 algorithms for 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 system 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 for algorithms that tolerate only one process crash (and even if we assume that all the processes in S are synchronous). We conclude that there is no communication-efficient algorithm for Ω in S that tolerates one process crash.

We next consider how to strengthen system S so that communication efficiency can be achieved. Specifically, since our complexity lower bounds are based on the lack of reliable communication in S, we make 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

[1] 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.

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

System	Properties
S^-	*Links* can be arbitrarily slow and lossy *Processes* can be arbitrarily slow and can crash, but they have a maximum execution speed
S	S^- with at least one eventually timely source (i.e., a timely correct process whose *output* links are eventually timely)
S^+	S^- with at least one eventually timely source *and* at least one fair hub (i.e., a correct process whose *input and output* links are fair)
S^{++}	S^- with at least one eventually timely source *and* such that all the links are fair

Table 1: Systems considered in this paper (in increasing order of strength).

System	Ω algorithm	Communication-efficient Ω algorithm
S^-	No	No
S	Yes	No
S^+, S^{++}	Yes	Yes

Table 2: Existence of algorithms and communication-efficient algorithms for Ω in different systems.

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 [ADGFT]. A correct process whose input and output links are fair is called a *fair hub*. Note that a fair hub need not be a timely process: it can be arbitrarily slow. We denote by S^+ a system S with at least one fair hub (whose identity is not known).[3]

S^+ is a weak type of partially synchronous system because it does not ensure timely communication between every pair of processes. In fact, in S^+ only messages sent *from* the eventually timely source(s) are guaranteed to be eventually timely. All other messages, including all those sent *to* the eventually timely sources, can be arbitrarily delayed (thus, processes cannot use eventually timely sources as intermediate nodes to communicate with each other in a timely way). This is in contrast to the partially synchronous systems defined in [DLS88, CT96] in which every pair of processes is connected by a link that is eventually timely in *both* directions.

Our next result is a *communication-efficient* algorithm for Ω in system S^+. We derive this algorithm in two stages: we first give a simpler algorithm that works in a system denoted S^{++} that is stronger than S^+, and then modify it so that it works in S^+. System S^{++} is a system S where *all* the links are fair. Tables 1 and 2 summarize our results on the existence and communication efficiency of algorithms for Ω in systems S^-, S, S^+, and S^{++}.

In summary, we investigate the feasibility and cost of implementations of Ω —a fundamental failure detector at the core of many algorithms —in systems with weak reliability and synchrony assumptions. Our contributions are the following:

1. We give the first algorithm that implements Ω in a weak partially synchronous system where only one unknown correct process needs to be timely (all other processes can be arbitrarily slow) and only the

[3] So S^+ is a system S^- with at least one eventually timely source *and* at least one fair hub, whose identities are not known. Note that the eventually timely source and the fair hub could be the same process.

links from that process need to be eventually reliable and timely (all other links can be arbitrarily slow and lossy). Previous algorithms for Ω required stronger reliability or synchrony assumptions.

2. We show that algorithms for Ω in this weak system are inherently expensive: all correct processes (except possibly one) must send messages forever; moreover, a quadratic number of links must carry messages forever. This holds even for algorithms for Ω that tolerate at most one process crash.

3. We then show that with a small additional assumption —the existence of some unknown correct process whose links can be arbitrarily slow and lossy but fair —there are efficient algorithms for Ω such that eventually only one process (the elected leader) sends messages.

It is worth noting that the results of this paper partially answer some questions questions posed by Keidar and Rajsbaum in their 2002 PODC tutorial [KR02] (this is explained in Section 7).

As a final remark, two of the algorithms presented in this paper (namely, the algorithms in Sections 4 and 6.1) were implemented and evaluated in a dynamically-changing system, where application processes may join, leave, crash or recover, and communication links may lose messages or disconnect for extended periods of time [ST07]. Experimental results presented in [ST07] indicate that the algorithms work well in practice: they are quite robust and inexpensive to run even in dynamic systems with high processor and link failure rates.

The rest of the paper is organized as follows. We first describe related work (Section 2). We next give an informal model of systems S^-, S, S^+, and S^{++} (Section 3). We then describe an algorithm for Ω in S (Section 4), and show that algorithms for Ω in S cannot be communication efficient (Section 5). We next give a communication-efficient algorithm for Ω in a system S^{++} (Section 6.1). Finally, we modify this algorithm so that it works in a system S^+ (Section 6.2). A brief discussion concludes the paper (Section 7).

2 Related work

Related work concerns the *use of* Ω to solve agreement problems and the *implementation of* Ω in various types of partially synchronous systems. Our paper is also related to the seminal work in [DDS87, DLS88] that identifies (weak) partial synchrony assumptions under which one can solve consensus. In [DDS87, DLS88], however, partial synchrony assumptions are uniform (i.e., they apply to all processes and/or all links) and message-efficiency is not a concern.

As we mentioned earlier, Ω is necessary to solve consensus and atomic broadcast [CT96, CHT96, DGFG03, EHT07] and it is used in several consensus algorithms [Lam98, MR01, LFA01, Lam01, DG02, CL02, GL03, MOZ05]. It is also a component of the weakest failure detector for the non-blocking atomic commit problem [DGFG$^+$04]. .

A simple implementation of Ω consists of implementing $\diamond\mathcal{P}$ first and outputting the smallest process currently not suspected by $\diamond\mathcal{P}$ [Lam98, PLL00]. However, 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 [LAF99] implements the failure detector $\diamond\mathcal{S}$ [4] in a way that only n links carry messages forever. However, this algorithm requires very strong system properties, namely, that no message is ever lost, all links are eventually timely in both directions, and all correct processes are timely. [LFA00] has an algorithm for Ω, but the paper assumes some strong system properties: all links are eventually reliable and timely, and all correct processes are timely.

Another communication-efficient implementation of Ω was given in [ADGFT01]. In that implementation, all correct processes need to be timely, but only the links to and from some (unknown) correct process need to be eventually timely, all other links can be arbitrarily slow and lossy. This system assumption is weaker than the ones in [LAF99, LFA00]. 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^+).

[MMR03] gives an implementation of Ω that works under an assumption on the ordering of message replies. More precisely, the implementation uses a query-response mechanism, with which a process broadcasts a query message and then waits for responses. Links are reliable and the implementation works provided that the query-response mechanism satisfies the following property: there exist a correct process p, a set S of $f+1$ processes (where f is a bound on the number of faulty processes), and a time after which, if a process $q \in S$ broadcasts a query, then q receives a reply from p among the first $n-f$ replies that q receives.

The results in our paper first appeared in an extended abstract [ADGFT03]. Since then, several papers have proposed implementations of Ω that work under weak synchrony assumptions [ADGFT04, MOZ05, HMSZ05, MRT06, FR06], as we now briefly explain.

In [ADGFT04], all links are fair and the algorithm for Ω works with the following synchrony assumption: there is some correct process p with f outgoing links that are eventually timely, where f is a bound on the number of faulty processes (such a process is called an *eventual f-source*).

In [MOZ05], all links are reliable and the Ω implementation uses query-response mechanism with the following synchrony assumption: there exist δ, a correct process p and a time after which, if p broadcasts a query then p receives replies from at least f other processes within δ time. Note that the f processes that reply to p in a timely fashion can vary over time.

In [HMSZ05], all links are fair and the Ω implementation uses a send-to-all primitive with the following synchrony assumption: there exist δ, a correct process p and a time after which, if p sends a message to all then at least f recipients receive the message within δ time. Note that the f recipients may change from message to message of p.

In [MRT06], all links are reliable and the Ω implementation is based on the query-response mechanism of [MMR03]. The implementation works under the conditions in [MMR03] *or* the system has an eventual f-source.

In [FR06], all links are reliable and all the correct processes regularly broadcast an $\text{ALIVE}(r)$ message, where r is an increasing integer (a "round number"). The synchrony assumption is defined in terms of the $\text{ALIVE}(r)$ messages: there exist a δ, a correct process p, and a suitable subset R of integers such that, for each $r \in R$, there is a set $S(r)$ of f processes such that $p \notin S(r)$ and for each process $q \in S(r)$, either (1) q has crashed, or (2) the $\text{ALIVE}(r)$ message sent by p is received by q within δ time, or (3) the $\text{ALIVE}(r)$ message sent by p is received by q among the first $n-f$ $\text{ALIVE}(r)$ messages received by q.

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

All the implementations of Ω in the above papers assume that every pair of correct processes can communicate via reliable or fair links. In contrast, the first algorithm that we present in this paper (for system S) works even if most processes cannot communicate with each other.

As a final remark, note that one can implement Ω in a given system by first implementing $\diamond\mathcal{S}$ in that system, and then transforming $\diamond\mathcal{S}$ to Ω using the algorithm in [Chu98]. This approach, however, cannot be used to implement Ω in system S: this is because the transformation algorithm in [Chu98] requires all processes to reliably communicate with each other (which may not be possible in S). Furthermore, this approach does not seem to help deriving a communication-efficient algorithm for Ω in system S^+: to use it, one must first derive a communication-efficient algorithm for $\diamond\mathcal{S}$ in S^+, and it is not clear that this algorithm would be significantly simpler than our algorithm for implementing Ω in S^+.

3 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 *directed* links. In our model, time values are taken from the set $\mathbf{R}^+$ of positive real numbers; time interval $(t_1, t_2]$ is the set of times $\{t \in \mathbf{R}^+ : t_1 < t \leq t_2\}$.

Processes. Processes are (finite or infinite) deterministic automata that execute by taking steps. In each step, a process p can do one of the following three things (according to p's state transition function): (1) p tries to receive a message from another process (as explained below) and then changes state, or (2) p sends a message to another process and then changes state, or (3) p just changes state.[5] A step need not be instantaneous, but we assume that each step takes effect at some instantaneous moment during the execution of the step.

A process p is *correct* if it executes infinitely many steps. If p executes only a finite number of steps, we say that p *crashes*.

We assume that processes have a maximum speed, i.e., there is an upper bound on the rate of execution of *every* process. More precisely, in every run every process p satisfies the following property:

- *[Maximum Rate of Execution]:* There exists $M_1 > 0$ such that for every time t, p executes at most one complete step during time interval $(t, t + M_1]$.

There may be a lower bound on the rate of execution of *some* processes. More precisely, we say that a process p is *timely (in a run)* if it satisfies the following property (in that run):

- *[Minimum Rate of Execution]:* There exists $M_2 > 0$ such that for every time t, p executes at least one complete step during time interval $(t, t + M_2]$.

Note that a timely process takes an infinite number of steps, and hence it must be correct. If a process is not timely, it may be intermittently or arbitrarily slow, or it may crash. Also note that M_1 and M_2 can vary per run and are not known to processes.

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

Links. Processes can send messages over a set of directed links. The network is fully connected, that is, there is a directed link from each process to every other process. The directed link from process p to process q, denoted $p \to q$, is an *output link* of p and an *input link* of q.

A message m carries a *type* T in addition to its *data* D: $m = (\mathrm{T}, \mathrm{D}) \in \{0,1\}^* \times \{0,1\}^*$. For each input link $q \to p$ of process p and each type T, p has a message buffer, denoted $\mathit{buffer}_p[q, \mathrm{T}]$, that can hold a *single* message of type T. Initially, $\mathit{buffer}_p[q, \mathrm{T}]$ is empty, denoted $\mathit{buffer}_p[q, \mathrm{T}] = \perp$. If q sends a message m of type T to p, and the link $q \to p$ does not lose m, then eventually $\mathit{buffer}_p[q, \mathrm{T}]$ is set to m. When this happens, we say that *message m is delivered to p from q*. If $\mathit{buffer}_p[q, \mathrm{T}]$ was already set to some previous message from q, that message is overwritten by m.

When a process p takes a step, it may choose a process q and a type T to read the contents of $\mathit{buffer}_p[q, \mathrm{T}]$. If $\mathit{buffer}_p[q, \mathrm{T}]$ has a message $m \neq \perp$ then we say that *p receives message m from q*, and $\mathit{buffer}_p[q, \mathrm{T}]$ is automatically reset to $\perp$. Otherwise p does not receive any message at that step. In either case, p may change its state to reflect the outcome.

Note that even if a message m of type T is delivered to p from q, there is no guarantee that p will eventually receive m. First, it is possible that p never chooses to check $\mathit{buffer}_p[q, \mathrm{T}]$. Second, it is also possible that $\mathit{buffer}_p[q, \mathrm{T}]$ is overwritten by a subsequent message from q of type T before p checks $\mathit{buffer}_p[q, \mathrm{T}]$.

To define link properties, it is convenient to assume that messages are unique (this can be achieved by associating a sequence number and sender id to each message).

Every link $p \to q$ satisfies the following property in every run:

- *[Integrity]:* A message m is delivered to q from p at most once, and only if p previously sent m to q.

Some links may satisfy additional properties which are described below.

We say that a link $p \to q$ is *eventually timely (in a run)* if it satisfies the following property (in that run):

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

The maximum message delay δ and the time t above can vary per run and are not known to processes.

A link that is not eventually timely can be arbitrarily slow and/or it can lose messages. A lossy link may satisfy the following fairness property: 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.[6]

More precisely, we say that a link $p \to q$ is *fair (in a run)* if it satisfies the following property (in that run):

- *[Type Fairness]*: For every type T, if p sends infinitely many messages of type T to q, then infinitely many messages of type T are delivered to q from p.

Eventually timely sources and fair hubs. A process p is an *eventually timely source* in a run if in that run (1) p is timely, and (2) the output links of p are eventually timely. Only the output links need to be eventually

[6] This kind of fairness property of links, which we call "type fairness", is new and is further discussed in [ADGFT].

timely, hence the word "source". A process p is *fair hub* in a run if in that run (1) p is correct, and (2) the input and output links of p are fair. Note that a fair hub and its input and output links can be arbitrarily slow.

Systems. We consider four systems, denoted S^-, S, S^+ and S^{++}, which differ on the properties of their processes and links. All these systems have the following properties: in every run, every process satisfies the Maximum Rate of Execution property and every link satisfies the Integrity property. System S^- has no other requirements. In system S, in every run, there is at least one eventually timely source. In system S^+, in every run, there is at least one eventually timely source and at least one fair hub. In system S^{++}, in every run, there is at least one eventually timely source and all the links are fair.

3.1 Failure detector Ω

The formal definition of failure detector Ω is given in [CT96, CHT96]. Informally, Ω outputs, at each process p, a single process denoted $leader_p$, such that the following property holds:[7]

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

Note that, at any given time, processes do not know if there is a commonly agreed leader; they only know that eventually there will be a common leader.

3.2 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.

4 Implementing Ω in system S

We now describe an algorithm that implements Ω in S. This algorithm, shown in Figure 2, 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 arbitrarily slow and lossy, except for the *output* links of some timely process whose identity is unknown).

In all the algorithms described in this paper, process uses some *local timers*. In particular, each process p uses a local timer denoted *SendAliveTimer* to periodically send ALIVE messages to other processes. Moreover, for each process $q \neq p$, p uses a local timer denoted *timer*$[q]$ to determine whether it has "recently" received an ALIVE message from process q. Process p implements its local timers as simple count-down counters as follows. Process p can "turn on" a local timer T by setting it to any non-negative integer k, that is, by executing the statement $T \leftarrow k$, where $k \geq 0$ is the "timeout" constant. As long as $T > 0$, process p periodically decrements T by one, and it does so at p's own pace. So, unless p first resets T, the value of T eventually reaches 0. When this occurs, we say that *timer T expires*.

[7] Henceforth, when we say that there is a time after which some property C holds, we mean that there is a time t such that for every time $t' \geq t$, property C holds at time t'.

A naïve attempt at implementing Ω is as follows. Each process periodically (a) sends ALIVE messages to the other processes, (b) computes the set of currently "alive" processes, as the set of processes from which it directly received an ALIVE message recently, and (c) selects as its leader the process with the smallest id in this set. But this algorithm does not work: in system S almost all links may suffer from arbitrary delays and/or losses, and this gives rise to several problems. In particular, (1) different processes may have different views of which processes are currently alive, and the different views may never converge, (2) a process with a small id may repeatedly alternate between appearing to be alive and crashed, and continue to do so forever. Such problems complicate the task of selecting a common and permanent leader: problem (1) may cause different processes to have different leaders (forever), and problem (2) may cause a process to repeatedly change its leader forever.

To overcome these and other similar difficulties, we use the following ideas. First, instead of selecting the leader according to the smallest process id, processes keep track of (roughly) how many times each process was previously suspected of having crashed, and they select as their leader the process with the fewest number of suspicions so far (among a set of alive processes). Second, the set of alive processes from which each process selects its leader is constructed in two stages. In the first stage, every process p periodically: (1) sends an ALIVE message to the other processes, (2) recomputes the set processes from which it directly received an ALIVE message recently (this set is denoted *active*), and (3) selects its "local" leader, denoted *localLeader*$[p]$, among the processes in its *active* set. In the second stage, every process p periodically: (1) sends its current *localLeader*$[p]$ to the other processes, (2) recomputes the set *localLeaders* of the local leaders of the processes in its *active* set, and (3) selects its (global) leader among the processes in *localLeaders*. These two stages are actually done concurrently. We now explain the algorithm in more detail.

The algorithm, shown in Figure 2, is structured as a **repeat forever** loop. In this loop, p first executes the *updateLeader* procedure to recompute its local leader and its (global) leader as described above. More precisely, p maintains a vector of "accusation" counters, denoted *counter*, where *counter*$[q]$ is p's rough estimate of how many times q's was previously suspected of having crashed. In the *updateLeader* procedure, p first selects its local leader as the process r with the smallest $(\textit{counter}[r], r)$ tuple, in lexicographical order, among the processes in its *active* set. Then p forms the set *localLeaders* consisting of all the local leaders of the processes in its *active* set. Finally, p selects its (global) leader as the process ℓ with the smallest $(\textit{counter}[\ell], \ell)$ tuple among the processes in its *localLeaders* set.

After updating its local and global leaders, p checks whether its *SendAliveTimer* has expired, i.e., whether *SendAliveTimer* $= 0$. If it has expired, then (a) p sends an ALIVE message to every process $q \neq p$ (each such message contains p's current local leader, the counter of this local leader according to p, and p's own counter), and (b) p resets its *SendAliveTimer* to some constant integer $\eta \geq 1$. Constant η is a "message efficiency" parameter that controls the rate at which p sends its ALIVE messages: p sends them once every η iterations of its repeat forever loop.

Then, for each process q, process p checks whether an ALIVE message was delivered from q, i.e., whether the corresponding buffer from q is non-empty. If so, p receives this message, it adds q to its *active* set, and it stores the local leader of q in the variable *localLeader*$[q]$. Process p also updates the counters of q and of the local leader of q. Finally, p resets *timer*$[q]$ by setting it to *timeout*$[q]$ (intuitively, p expects to receive the next ALIVE message from q within *timeout*$[q]$ iterations of its repeat forever loop).

If *timer*$[q]$ expires (before p receives another ALIVE message from q), then p removes q from its *active* set, and it sends an ACCUSATION message to q to tell q that it suspects q of having crashed. Process p also

Variable	Intuitive description
active	set of processes that p considers to be currently alive
counter$[q]$	p's estimate of q's accusation counter (the number of times processes previously timed out on q)
SendAliveTimer	count-down timer used to send an ALIVE message every η iterations of the repeat forever loop
timer$[q]$	count-down timer used to determine whether q is currently alive (*timer*$[q]$ is initialized to *timeout*$[q]$ and it is decremented by one in each iteration of the repeat forever loop; if/when $timer[q]$ reaches 0, it is reset to *timeout*$[q]$)
timeout$[q]$	length of p's timeout on q
localLeader$[q]$	p's estimate of q's local leader (q chooses its local leader to be the process r with the smallest tuple $(\textit{counter}[r], r)$ among all the processes in q's *active* set)
localLeaders	p's estimate of the set of local leaders of all the processes in p's *active* set
leader	the leader of p (p chooses its leader to be the process ℓ with the smallest tuple $(\textit{counter}[\ell], \ell)$ among all the processes in p's *localLeaders* set)

Table 3: Local variables of process p in the algorithm of Figure 2.

increments *timeout*$[q]$, and it restarts *timer*$[q]$ with this larger timeout. Intuitively, p increases the timeout on q because it does not know the speed of the eventually timely sources and the delay of their output links.

Then p checks whether an ACCUSATION message was delivered. If so, p receives it, and p increases its own accusation counter *counter*$[p]$. Finally, at the end of the repeat forever loop, p decrements by one every timer that it uses, namely, *SendAliveTimer* and *timer*$[q]$ for every $q \neq p$.

Note that this algorithm uses only two message types: ALIVE and ACCUSATION.

Figure 2 describes the algorithm by giving the pseudo-code of an (arbitrary) process p, and Table 3 describes the local variables of p. Recall that in our model, p is a deterministic automaton that takes steps, but it is easy to translate the pseudo-code of p given here into such an automaton. Without loss of generality, we can assume that: (1) for some integer b, each iteration of the repeat forever loop (lines 8–29) takes at most b automaton steps (this is because there are no infinite loops, waiting statements, or similar constructs in lines 9–29), and (2) each iteration of the repeat forever loop takes at least two complete automaton steps.

We now show that the algorithm in Figure 2 implements Ω in system S. Henceforth, we consider an arbitrary run of this algorithm in system S, and s is an eventually timely source in this run.

In the following, the local variable var of a process p is denoted by var_p. The value of var_p at time t is denoted by var_p^t.[8]

Lemma 1 *For every correct process p and every process $q \neq p$, the following holds:*

(a) If $q \in \textit{active}_p$ holds infinitely often[9] *then p receives* ALIVE *messages from q infinitely often.*

(b) If $q \in \textit{localLeaders}_p$ holds infinitely often then p receives ALIVE *messages from q infinitely often, or p receives $(\text{ALIVE}, q, -, -)$ messages infinitely often.*

[8] If a step of p takes effect at time t, then var_p^t is the value of var_p just after this step.

[9] A condition C holds infinitely often if for every time t, there is a time $t' > t$ such that C holds at time t'. Note that "C holds infinitely often" is the opposite of "there is a time after which C does not hold".

CODE FOR EACH PROCESS p:

```
procedure updateLeader()
    localLeader[p] ← r such that (counter[r], r) = min{(counter[q], q) : q ∈ active}
    localLeaders ← {localLeader[q] : q ∈ active}
    leader ← ℓ such that (counter[ℓ], ℓ) = min{(counter[q], q) : q ∈ localLeaders}

main code
    { Initialization }
    for each q ∈ Π do counter[q] ← 0 ; localLeader[q] ← {q}
    for each q ∈ Π \ {p} do timeout[q] ← η + 1; timer[q] ← timeout[q]
    active ← {p}
    SendAliveTimer ← 0                    { p sets SendAliveTimer = 0 to start sending ALIVE messages }
    repeat forever
        updateLeader()
        if SendAliveTimer = 0 then
            send (ALIVE, localLeader[p], counter[localLeader[p]], counter[p]) to every process except p
            SendAliveTimer ← η
        for each q ∈ Π \ {p} do
            if receive (ALIVE, r, rcntr, qcntr) from q then
                active ← active ∪ {q}
                localLeader[q] ← r
                counter[q] ← max{counter[q], qcntr}
                counter[r] ← max{counter[r], rcntr}
                timer[q] ← timeout[q]
            if timer[q] = 0 then
                send ACCUSATION to q
                active ← active − {q}
                timeout[q] ← timeout[q] + 1
                timer[q] ← timeout[q]
            if receive ACCUSATION from q then
                counter[p] ← counter[p] + 1
        if SendAliveTimer > 0 then SendAliveTimer ← SendAliveTimer − 1
        for each q ∈ Π \ {p} do
            if timer[q] > 0 then timer[q] ← timer[q] − 1
```

Figure 2: Implementation of Ω for system S.

PROOF. Consider two processes p and q such that p is correct and $q \neq p$.

(a) Assume that $q \in \mathit{active}_p$ holds infinitely often. Since $q \neq p$, p receives at least one ALIVE from q that causes p to first insert q in active_p. If there is a time after which p does not receive ALIVE from q, then eventually $\mathit{timer}_p[q]$ expires (i.e., $\mathit{timer}_p[q]$ reaches 0), p removes q from active_p, and p never inserts q back into this set again —a contradiction that shows part (a).

(b) Assume that $q \in \mathit{localLeaders}_p$ holds infinitely often. Since p resets $\mathit{localLeaders}_p$ to $\{\mathit{localLeader}_p[u] : u \in \mathit{active}_p\}$ infinitely often (in the *updateLeader* procedure that p executes in line 9), there must be at least one process r such that $\mathit{localLeader}_p[r] = q$ and $r \in \mathit{active}_p$ infinitely often. There are two possible cases:

(1) $r = p$. In this case, $\mathit{localLeader}_p[p] = q$ infinitely often. Since p resets $\mathit{localLeader}_p[p]$ infinitely often to a process in active_p, then $q \in \mathit{active}_p$ infinitely often. By part (a) of the lemma, p receives ALIVE messages from q infinitely often.

(2) $r \neq p$. Suppose, for contradiction, that there is a time t after which p does not receive $(\text{ALIVE}, q, -, -)$ messages. Since $r \in \mathit{active}_p$ infinitely often and $r \neq p$, by part (a) of the lemma, p receives ALIVE messages from r infinitely often. After time t, none of these messages are $(\text{ALIVE}, q, -, -)$ message. So there is a time after which $\mathit{localLeader}_p[r] \neq q$ —a contradiction. Thus, p receives $(\text{ALIVE}, q, -, -)$ messages infinitely often.

□

Observation 2 *For all processes p and q, $\mathit{counter}_p[q]$ is monotonically nondecreasing with time.*

Lemma 3 *For every two processes $p \neq q$, if*

(a) p receives ALIVE *messages from q infinitely often, or*

(b) p receives $(\text{ALIVE}, q, -, -)$ messages infinitely often

then (c) q is correct, and for every time t, there is a time after which $\mathit{counter}_p[q] \geq \mathit{counter}_q^t[q]$.

PROOF.

Part 1: (a) $\Rightarrow$ (c). Consider two processes $p \neq q$, and suppose that p receives ALIVE messages from q infinitely often. Then q sends such messages infinitely often, and so q is correct. Consider any time t. Eventually p receives a message $m = (\text{ALIVE}, -, -, \mathit{qcntr})$ that is sent by q after time t. Since q sends m after time t and $\mathit{counter}_q[q]$ is monotonically nondecreasing, $\mathit{qcntr} \geq \mathit{counter}_q^t[q]$. So, when p receives m from q, p sets $\mathit{counter}_p[q]$ to a value $v \geq \mathit{qcntr} \geq \mathit{counter}_q^t[q]$. Thereafter, $\mathit{counter}_p[q] \geq \mathit{counter}_q^t[q]$ (because $\mathit{counter}_p[q]$ is monotonically nondecreasing).

Part 2: (b) $\Rightarrow$ (c). Consider two processes $p \neq q$, and suppose that p receives $(\text{ALIVE}, q, -, -)$ messages infinitely often. Then, for some process r, p receives $(\text{ALIVE}, q, -, -)$ from r infinitely often. If $r = q$ then condition (c) holds by part 1 of this proof, and we are done.

Now assume $r \neq q$. Consider any time t, and let $C = \mathit{counter}_q^t[q]$. Note that r sends $(\text{ALIVE}, q, -, -)$ to p infinitely often. Each time r sends such a message in line 11, $\mathit{localLeader}_r[r] = q$, and so $q \in \mathit{active}_r$ at that time (this is because r resets $\mathit{localLeader}_r[r]$ to a process in active_r just before r sends $(\text{ALIVE}, q, -, -)$). Thus, $q \in \mathit{active}_r$ holds infinitely often. Since $r \neq q$, then by Lemma 1 part (a), r receives ALIVE messages from q infinitely often. By part 1 of this proof, q is correct and there is a time after which process r has $\mathit{counter}_r[q] \geq C$. So p eventually receives a message $m = (\text{ALIVE}, q, \mathit{qcntr}, -)$ from r such that $\mathit{qcntr} \geq C$. When p receives m, p sets $\mathit{counter}_p[q]$ to a value $v \geq \mathit{qcntr} \geq C$. Thereafter, $\mathit{counter}_p[q] \geq \mathit{counter}_q^t[q]$ (because $\mathit{counter}_p[q]$ is monotonically nondecreasing).

□

Lemma 4 *For every correct process p and every process q, if*

(a) $q \in \mathit{active}_p$ holds infinitely often, or

(b) $q \in \mathit{localLeaders}_p$ holds infinitely often

then (c) q is correct, and for every time t, there is a time after which $\mathit{counter}_p[q] \geq \mathit{counter}_q^t[q]$.

PROOF. If $p = q$, condition (c) holds because p is correct and $\mathit{counter}_p[p]$ is monotonically nondecreasing.

Now assume that $p \neq q$. If (a) or (b) holds, then by Lemma 1, p receives ALIVE messages from q infinitely often, or p receives $(\text{ALIVE}, q, -, -)$ messages infinitely often. By Lemma 3, condition (c) holds. □

Recall that s is an eventually timely source in the run under consideration.

Lemma 5 *There is a constant $\alpha > 0$ such that, for all $k \geq 0$ and every time t, process s executes at least k complete iterations of its repeat forever loop during time interval $(t, t + k\alpha]$.*

PROOF. The lemma follows directly from two facts: (1) there is an integer b such that each complete iteration of the repeat forever loop of s takes at most b automaton steps, and (2) s satisfies the Minimum Rate of Execution property (because s is a timely process). □

Definition 6 *Let $\alpha > 0$ be a constant that satisfies Lemma 5.*

Recall that $\eta \geq 1$ is the "timeout" value of *SendAliveTimer* (see line 12).

Definition 7 *Let $\Delta' = (\eta + 1)\alpha$.*

Lemma 8 *For every process $p \neq s$, if s sends an ALIVE message to p at some time t, then s sends another ALIVE message to p during time interval $(t, t + \Delta']$.*

PROOF. Suppose s sends an ALIVE message to $p \neq s$ at some time t (this occurs in line 11). Then, when s executes line 12 (in the same iteration of its repeat forever loop) s sets *SendAliveTimer* to $\eta \geq 1$. Since s decrements *SendAliveTimer* by one in each iteration of its repeat forever loop (in line 27), s sets *SendAliveTimer* to 0 by the time it completes η such iterations. By Lemma 5, this takes at most $\eta\alpha$ units of time. So by time $t + \eta\alpha$, s sets *SendAliveTimer* to 0. Thus, by the time s completes one more iteration of the repeat forever loop, i.e., by time $t + \eta\alpha + \alpha = t + \Delta'$, s executes line 10 with *SendAliveTimer* $= 0$ and sends another ALIVE message to p. □

Lemma 9 *For every process $p \neq s$, there is a time t' such that for every $t \geq t'$, s sends an* ALIVE *message to p during time interval $(t, t + \Delta']$.*

PROOF. Let $p \neq s$. When s executes its initialization code (lines 4-7), s sets its *SendAliveTimer* to 0. Thus, in its first execution of the repeat forever loop (lines 8-29), s executes line 10 with *SendAliveTimer* $= 0$ and sends an ALIVE message to p at some time t_1. By Lemma 8, s sends another ALIVE message to p at time $(t_1, t_1 + \Delta']$. The lemma follows by repeated applications of Lemma 8. □

Lemma 10 *There is a constant Δ and a time t_Δ such that, for all processes p, if s sends a message m to p at some time $t \geq t_\Delta$, then m is delivered to p from s by time $t + \Delta$.*

PROOF. This follows immediately from the fact that s is an eventually timely source, and therefore all its output links are eventually timely. □

Definition 11 *Let Δ be a constant that satisfies Lemma 10.*

Lemma 12 *For every process $p \neq s$, there is a time t' such that for every $t \geq t'$, there is an* ALIVE *message delivered to p from s during time interval $(t, t + \Delta' + \Delta]$.*

PROOF. Follows directly from Lemmas 9 and 10. □

Lemma 13 *There is a constant $\epsilon > 0$ such that, for every $k \geq 1$ and every process p, p takes at least $k\epsilon$ time to execute k complete iterations of its repeat forever loop.*

PROOF. The lemma follows from the following facts: (1) each complete iteration of p's repeat forever loop takes at least two complete automaton steps, and (2) p satisfies the Maximum Rate of Execution property. We now explain this proof in more detail.

Let $k \geq 1$ and consider some process p . To execute a complete iteration of the repeat forever loop, p takes at least two complete automaton steps. Thus, to execute k complete iterations of the loop, p takes at least $2k$ complete steps. By the Maximum Rate of Execution property, there exists a constant $M_1 > 0$ such that for every time t, p executes at most one complete step during time interval $(t, t + M_1]$. Thus, for every time t and every $k \geq 1$, p executes at most $2k - 1$ complete steps during time interval $(t, t + kM_1]$. Let $\epsilon = M_1$. We conclude that p takes at least $kM_1 = k\epsilon$ time to execute k complete iterations of the repeat forever loop. □

Definition 14 *Let ϵ be a constant that satisfies Lemma 13.*

Note that, by Lemma 13, p takes at least $\Delta' + \Delta$ time to execute $\lceil(\Delta' + \Delta)/\epsilon\rceil$ complete iterations of the repeat forever loop.

Definition 15 *Let $\zeta = \lceil(\Delta' + \Delta)/\epsilon\rceil + 2$.*

Lemma 16 *For every correct process $p \neq s$, there is a time after which p receives an* ALIVE *message from s at least once every ζ consecutive iterations of p's repeat forever loop.*

PROOF. Consider a correct process $p \neq s$. By Lemma 12, there is a time t' such that for every $t \geq t'$, there is an ALIVE message delivered to p from s during time interval $(t, t + \Delta' + \Delta]$.

Thus, there are infinitely many ALIVE messages that are delivered to p from s. Since p is correct, it executes its repeat forever loop infinitely often. In each iteration of this loop, p tries to receive an ALIVE message from every process $q \neq p$ (including s), so p receives ALIVE messages from s infinitely often.

Suppose p receives an ALIVE message from s at some time $t > t'$. From Lemma 12, another ALIVE message is delivered from s during the period $(t, t + \Delta' + \Delta]$. Thus, by Lemma 13, this ALIVE message is delivered to p before p completes $\lceil(\Delta' + \Delta)/\epsilon\rceil + 1$ consecutive iterations of its repeat forever loop. So p receives this ALIVE message by the time it completes $\lceil(\Delta' + \Delta)/\epsilon\rceil + 2$ iterations of the loop.

We conclude that there is a time after which p receives an ALIVE message from s at least once every $\zeta = \lceil(\Delta' + \Delta)/\epsilon\rceil + 2$ consecutive iterations of its repeat forever loop. □

Observation 17 *For every correct process p, there is a time after which $p \in$ active$_p$.*

PROOF. When p executes its initialization code, it sets *active*$_p$ to $\{p\}$. Thereafter, p never removes itself from *active*$_p$. □

Lemma 18 *For every correct process p, there is a time after which $s \in$ active$_p$.*

PROOF. Let p be any correct process. If $p = s$ then, by Observation 17, there is a time after which $s \in$ *active*$_p$. Now assume that $p \neq s$. By Lemma 16, there is a time t_1 after which p receives an ALIVE message from s at least once every ζ consecutive iterations of its repeat forever loop. Each time p receives such a message, p adds s to *active*$_p$. We claim that p removes s from *active*$_p$ only a finite number of times, which concludes the proof. Suppose, for contradiction, that p removes s from *active*$_p$ infinitely often (line 22). Then, p increments *timeout*$_p[s]$ infinitely often (line 23), and so there is a time t_2 after which *timeout*$_p[s] > \zeta$. We now consider p's execution after time $t = \max(t_1, t_2)$.

After time t, each time p receives an ALIVE message from s, p resets *timer*$_p[s]$ to *timeout*$_p[s] > \zeta$. After each iteration where *timer*$_p[s]$ is reset this way, *timer*$_p[s]$ can decrease to 0 only if p completes at least ζ consecutive iterations of its repeat forever loop without receiving any ALIVE message from s (in each such iteration p decreases *timer*$_p[s]$ by one). But after time t process p receives an ALIVE message from s at least once every ζ consecutive iterations of its repeat forever loop. So there is a time after which *timer*$_p[s] \neq 0$. Note that p removes s from *active*$_p$ only if it executes line 20 with *timer*$_p[s] = 0$. Thus there is a time after which p does not remove s from *active*$_p$ —a contradiction. □

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

PROOF. Consider any correct process $p \neq s$. Each time p sends an ACCUSATION message to s, p removes s from $active_p$. By Lemma 18, there is a time after which p does not remove s from $active_p$. So there is a time after which p does not send any ACCUSATION messages to s. Moreover, s never sends ACCUSATION messages to itself. Thus there is a time after which no process (whether correct or faulty) sends ACCUSATION messages to s. Since s increases $counter_s[s]$ only when it receives such messages, $counter_s[s]$ is bounded. □

Definition 20 *For every process p, let c_p be the largest value of $counter_p[p]$ in the run that we consider ($c_p = \infty$ if $counter_p[p]$ is unbounded). Let ℓ be the process such that $(c_\ell, \ell) = \min\{(c_p, p) : p$ is a correct process$\}$.*

By definition, ℓ is a correct process. Furthermore, by Lemma 19, $counter_s[s]$ is bounded, i.e., $c_s < \infty$. Thus, $c_\ell < \infty$, i.e., $counter_\ell[\ell]$ is bounded.

Lemma 21 *For every correct process p,*

(a) if there is a time after which $\ell \in active_p$ then there is a time after which $localLeader_p[p] = \ell$, and

(b) if there is a time after which $\ell \in localLeaders_p$ then there is a time after which $leader_p = \ell$.

PROOF.

(a) Let p be any correct process, and suppose that there is a time after which $\ell \in active_p$. We claim that for every $q \neq \ell$, (i) there is a time after which $q \notin active_p$, or (ii) there is a time after which $(counter_p[\ell], \ell) < (counter_p[q], q)$. From the way p sets $localLeader_p[p]$ in the *updateLeader* procedure, this claim implies there is a time after which $localLeader_p[p] = \ell$.

To show the claim, consider any process $q \neq \ell$, and suppose that condition (i) does not hold, i.e., suppose that $q \in active_p$ holds infinitely often. We now show that condition (ii) is satisfied. By Lemma 4 part (a), q is correct, and for every time t, there is a time after which $counter_p[q] \geq counter_q^t[q]$. There are two cases:

(1) *$counter_q[q]$ is bounded.* In this case, $c_q < \infty$, and so there is a time t when $counter_q^t[q] = c_q$. So there is a time after which $counter_p[q] \geq c_q$. Recall that q is correct and $q \neq \ell$, and so by the definition of ℓ, we have $(c_\ell, \ell) < (c_q, q)$. Since $counter_p[\ell] \leq c_\ell$ (always), there is a time after which $(counter_p[\ell], \ell) \leq (c_\ell, \ell) < (c_q, q) \leq (counter_p[q], q)$.

(2) *$counter_q[q]$ is unbounded.* In this case, $counter_p[q]$ is also unbounded. So there is a time after which $counter_p[\ell] \leq c_\ell < counter_p[q]$.

So, in both cases, there is a time after which $(counter_p[\ell], \ell) < (counter_p[q], q)$, i.e., condition (ii) holds.

(b) (Similar to the proof of part (a).)

Let p be any correct process, and suppose that there is a time after which $\ell \in localLeaders_p$. We claim that for every $q \neq \ell$, (i) there is a time after which $q \notin localLeaders_p$, or (ii) there is a time after which

$(\mathit{counter}_p[\ell], \ell) < (\mathit{counter}_p[q], q)$. From the way p sets $leader_p$ in the *updateLeader* procedure, this claim implies there is a time after which $leader_p = \ell$.

To show the claim, consider any process $q \neq \ell$, and suppose that condition (i) does not hold, i.e., suppose that $q \in \mathit{localLeaders}_p$ holds infinitely often. We now show that condition (ii) is satisfied. By Lemma 4 part (b), q is correct, and for every time t, there is a time after which $\mathit{counter}_p[q] \geq \mathit{counter}_q^t[q]$. The rest of the proof now proceeds identically to cases (1) and (2) of part (a) above.

□

We now proceed to show that for every correct process p there is a time after which $\ell \in \mathit{localLeaders}_p$ (and hence, by the above lemma, there is a time after which $leader_p = \ell$).

Lemma 22 *There is a time after which $\ell \in \mathit{active}_s$.*

PROOF. If $\ell = s$ then, by Observation 17, there is a time after which $\ell \in \mathit{active}_s$. Now suppose $\ell \neq s$. There are three possible cases: (1) there is a time after which $\ell \in \mathit{active}_s$, (2) ℓ is added to and removed from active_s infinitely often, or (3) there is a time after which $\ell \notin \mathit{active}_s$. We now show that cases (2) or (3) cannot occur. In case (2), every time s removes ℓ from active_s, s sends an ACCUSATION message to ℓ, and so s sends ACCUSATION messages to ℓ infinitely often. In case (3), there is a time after which s does not receive ALIVE messages from ℓ. Thus, $\mathit{timer}_s[\ell]$ expires infinitely often at s (this is because s initially sets $\mathit{timer}_s[\ell]$ to some positive value, and each time this timer expires, s resets it to a positive value). Therefore, s sends ACCUSATION messages to ℓ infinitely often. So, in both cases (2) and (3), s sends ACCUSATION messages to ℓ infinitely often. Since the output links of s are eventually timely, and ℓ tries to receive an ACCUSATION message from s infinitely often (specifically once in each iteration of its repeat forever loop), ℓ receives ACCUSATION messages from s infinitely often. Thus, ℓ increments $\mathit{counter}_\ell[\ell]$ infinitely often, and so $\mathit{counter}_\ell[\ell]$ is not bounded —a contradiction. Thus, only case (1) is possible. □

Lemma 23 *There is a time after which $\mathit{localLeader}_s[s] = \ell$.*

PROOF. By Lemma 22, there is a time after which $\ell \in \mathit{active}_s$. So, by Lemma 21 part (a), there is a time after which $\mathit{localLeader}_s[s] = \ell$. □

Lemma 24 *For every correct process p, there is a time after which $\mathit{localLeader}_p[s] = \ell$.*

PROOF. Consider any correct process p. If $p = s$ then the result is immediate from Lemma 23 . Now assume that $p \neq s$. In this case, from Lemma 16, p receives ALIVE messages from s infinitely often. By Lemma 23, there is a time t after which $\mathit{localLeader}_s[s] = \ell$. So after time t, all the ALIVE messages that s sends to p are of the form $(\text{ALIVE}, \ell, -, -)$. Thus, there is a time after which all the ALIVE messages that p receives from s are of the form $(\text{ALIVE}, \ell, -, -)$. So there is a time after which $\mathit{localLeader}_p[s] = \ell$. □

Corollary 25 *For every correct process p, there is a time after which $\ell \in \mathit{localLeaders}_p$.*

PROOF. From Lemmas 18 and 24, there is a time after which $s \in \mathit{active}_p$ and $\mathit{localLeader}_p[s] = \ell$. Since p repeatedly sets $\mathit{localLeaders}_p$ to $\{\mathit{localLeader}_p[q] : q \in \mathit{active}_p\}$, there is a time after which $\ell \in \mathit{localLeaders}_p$. □

Lemma 26 *For every correct process p, there is a time after which $\mathit{leader}_p = \ell$.*

PROOF. Immediate from Lemma 21 part (b) and Corollary 25. □

From Lemma 26 and the fact that ℓ is a correct process, we have

Theorem 27 *The algorithm in Figure 2 implements Ω in system S.*

5 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 lower bounds that we derive here hold even if we assume that all processes in S are synchronous (i.e., all processes have the same, constant execution speed) and *at most one process may crash*.

Theorem 28 *Consider any algorithm $\mathcal{A}$ for Ω in a system S with $n \geq 2$ processes such that all processes are synchronous and at most one process may crash.*

1. *In* every *run of $\mathcal{A}$, all correct processes, except possibly one, send messages forever.*
2. *In* some *run of $\mathcal{A}$, at least $\lfloor \frac{n^2}{4} \rfloor$ links carry messages forever.*

PROOF. Henceforth we consider an algorithm $\mathcal{A}$ that implements Ω in a system S with $n \geq 2$ processes such that all processes are synchronous and at most one them may crash. We first observe the following:

Lemma 29 *For any run of $\mathcal{A}$ and any correct process p, if there is a time after which p does not receive any message from other processes, then there is a time after which the leader of p is p.*

To prove this lemma, suppose there is a run R of $\mathcal{A}$, a correct process p, and a time t after which p does not receive any message. Without loss of generality, we can assume that no process crashes in R. This is because if some process f crashes at some time t' (i.e., f stops taking steps after time t') in R, we can modify R to get a similar run where f never crashes, but all its output links crash permanently at time t' (i.e., they lose all the messages that f sends after time t'); this modified run is indistinguishable from R to all processes, except for process f who is now correct.

Since R is a run of an algorithm that implements Ω, and process p is correct, in run R there is a correct process q and a time after which the leader of p is q. We claim that $q = p$ (which proves the lemma).

Suppose, for contradiction, that $q \neq p$. Let R' be a run of $\mathcal{A}$ that is identical to R up to time t, and such that after time t: (a) process q crashes, and (b) all the *input* links of p crash permanently, while the *output* links of p become timely and stop losing messages (p is the eventually timely source in run R'). Since process p receives exactly the same messages at the same times in R and R', p cannot distinguish between R and R', and so it behaves exactly the same way in R and R'.[10]

Thus, in run R' of $\mathcal{A}$ there is a time after which the leader of p is q, even though q crashes —a contradiction that concludes the proof of Lemma 29.

We now prove part (1) of the theorem. Let R be an arbitrary run of algorithm $\mathcal{A}$, and $correct(R)$ be the number of correct processes in R. To prove Part (1) of the theorem, we must show that at least $correct(R)-1$ correct processes send messages forever (*). To do so, consider the following two cases:

(a) $correct(R) \leq 1$. In this case, (*) trivially holds.

(b) $correct(R) \geq 2$. Suppose, for contradiction, that (*) does not hold, i.e., at most $correct(R) - 2$ correct processes send messages forever. Thus in R there are at least two distinct correct processes that do *not* send messages forever. In other words, in R there are two distinct correct processes p and q and a time t such that p and q do not send any message after 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 eventually timely sources in R), and (b) no process crashes (the argument is as before: we can "replace" the crash of a process, by the simultaneous and permanent crash of all its output 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 of $\mathcal{A}$ that is identical to R except that p crashes in R' after time t. Note that, except for p, processes cannot distinguish between runs R and R', and so they behave the same in 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 of $\mathcal{A}$ that is identical to R, except that in R'' after time t, (1) all the output links of p crash permanently, and (2) all the input links of p crash permanently, except for the link from q to p which, as in run R, is eventually timely (so q is the eventually timely source of run R''). Note that, except for p, processes cannot distinguish between runs R and R'', and so they behave the same in 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 Lemma 29, there is a time after which the leader of p is p. Thus, in run R'' of $\mathcal{A}$ correct processes p and q do not reach agreement on a common leader —a contradiction. So (*) holds, and this concludes the proof of part (1) 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 of $\mathcal{A}$ such that: (a) all the n processes are correct, (b) all the links between processes in A are eventually timely, (c) A has an eventually timely 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 crashed, and (e) all the output links of every process in B are permanently crashed. So in run R, any process $p \in B$ can receive messages only from process s: all messages sent by other processes to p are lost.

[10] Note that even if the algorithm $\mathcal{A}$ that p executes is non-deterministic, we can chose run R' such that p behaves the same in R' and in R.

Note that in run R, for every process $q \in A$ and every process $p \in B$, there is a time after which the leader of q is not p. Intuitively, this is because p may eventually crash, and since p's output links are permanently crashed, q would not be able to notice p's 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, p can receive messages only from q ($= s$). Since in R there is a time after which q does not send messages to p, then eventually p stops receiving messages. So, by Lemma 29, 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 of $\mathcal{A}$ which is similar to R, except that the eventually timely source is q rather than s. More precisely, R' is like R, except that all the links from s to processes in B are permanently crashed, and all the links from q to processes in B are eventually timely. Since no process in B can communicate with anyone (their output links are permanently crashed in both R and R'), processes in A cannot distinguish between runs R and R', and so they behave the same in 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 crashed in R', then there is a time after which p does not receive any message in R'. So, by Lemma 29, in R' there is a time after which the leader of p is p. Thus, in run R' of $\mathcal{A}$ 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 $\lceil \frac{n}{2} \rceil \cdot \lfloor \frac{n}{2} \rfloor = \lfloor \frac{n^2}{4} \rfloor$ links carry messages forever in run R. $\square$

From Theorem 28 part (1), we immediately get the following result:

Corollary 30 *There is no communication-efficient algorithm for Ω in a system S with $n \geq 3$ processes, even if we assume that all processes are synchronous and at most one process may crash.*

6 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 n). In order to achieve this, Theorem 28 implies that we must strengthen the system model S. In this section, we first give a communication-efficient algorithm for Ω for system S^{++} (i.e., a system S where all links are fair), and then we modify this algorithm so that it works in system S^{+} (i.e., a system S where only the links to and from some unknown timely process are fair).

6.1 Implementing Ω in system S^{++}

We now give a communication-efficient algorithm for Ω in system S^{++}. Recall that in S^{++} there is an eventually timely source and all the 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 *active*, from which it received an ALIVE message recently (an adaptive timeout is used to determine the current set of active processes), and (c) chooses as leader the process with smallest id in its set *active*.[11] Such a simple algorithm would work in a system where *all* correct processes are eventually timely sources. But in system S^{++}, it would fail: for example, if S^{++} has *only one* eventually timely source and this process happens to have a large id, the leadership could forever oscillate among the correct processes that have a smaller id.

To fix this problem, we use a similar technique as in our previous algorithm (in Figure 2): a process uses accusation counters, not process ids, to select the leader among processes in its *active* set. More precisely, each process keeps a counter of the number of times it was previously suspected of having crashed, 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 processes in its *active* set (using the process ids to break ties). If a process p times out on a process q in $active_p$, p removes q from $active_p$ and it sends an "accusation" message to q, which causes q to increment its own accusation counter. The hope here is that, as with the previous algorithm, the counter of each eventually timely source remains bounded (because it is timely and all its output links are eventually timely), and so the correct process with the smallest bounded counter is eventually selected as the leader by all.

The above algorithm, however, does not work in system S^{++}: this is because the accusation counter of all correct processes may keep increasing forever, causing the leadership to keep oscillating forever. To see this, consider the following scenario in a system with $n = 2$ processes, namely, p and s. (We can extend this scenario to any number of processes.) Process s is the eventually timely source, while process p is correct but its output links are not always timely. Suppose that the accusation counters of p and s are 1 and 3, respectively, but, because s has not received a recent message from p, s considers itself to be the leader. Then, s receives an ALIVE message from p, and so p joins s's *active* set. Since p's accusation counter is smaller than the counter of s, the leader of s becomes p. When s gives up the leadership, it stops sending ALIVE messages (for communication efficiency). Unfortunately, this triggers p to time out on s and so p sends an ACCUSATION message that causes s to increment its accusation counter to 4. Now p's ALIVE messages become slow, causing the following chain of events: (a) s times out on p, (b) s sends an ACCUSATION to p, causing p to increment its accusation counter to 2, (c) s removes p from its *active* set, causing s to consider itself to be the leader again. Now, the accusation counters of p and s are 2 and 4, respectively, and this scenario can repeat itself forever. This results in a run where the accusation counters of p and s keep increasing and the leader of s keeps oscillating between p and s.

To fix this problem, a process p 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 the ALIVE messages that it previously sent (and not by the fact that p 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 the leadership in the past —this is its current *phase number* —and it includes this number in each ALIVE message that it

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

Variable	Intuitive description
active	set of processes that p considers to be currently competing for leadership
counter$[q]$	p's estimate of q's accusation counter (the number of times processes previously timed out on q)
phase$[q]$	p's estimate of the number of times that q voluntarily relinquished the leadership
SendAliveTimer	count-down timer used to periodically send ALIVE messages (if it is set to -1 it is deactivated)
timer$[q]$	count-down timer used to determine whether q is currently active (if it is set to -1 it is deactivated)
timeout$[q]$	length of p's timeout on q
leader	the leader of p (p chooses its leader to be the process ℓ with the smallest tuple $(\mathit{counter}[\ell], \ell)$ among all the processes in p's *active* set)
newleader	temporary variable for storing a newly computed leader of p

Table 4: Local variables of process p in the algorithm of Figure 3.

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 message that it sends to p; p considers this ACCUSATION message to be legitimate only if the phase number that it contains matches its own. Furthermore, whenever p gives up the leadership and stops sending ALIVE messages voluntarily, p increments its own phase number (and it does not communicate this new phase number to any process): this effectively causes p to ignore all the spurious ACCUSATION messages that may result if/when p voluntarily stops sending ALIVE messages.

As we mentioned above, as long as a process p considers itself to be the leader, p periodically sends an ALIVE message to every process except itself. If p considers that some other process is the leader, it does not send any ALIVE messages. This is done using a timer, denoted *SendAliveTimer*, as follows. Whenever p changes its *active* set or the accusation counter of a process, p recomputes its leader by executing the *updateLeader*$()$ procedure. If the leader of p changes, p checks whether it has just gained or lost the leadership.

1. If p gained the leadership, p turns on its *SendAliveTimer* by setting it to 0 (in line 4). Note that p periodically checks whether *SendAliveTimer* $= 0$ (line 15). If it is, then p sends an ALIVE message to every process $q \neq p$, and it resets *SendAliveTimer* to η to schedule its next sending of ALIVE messages (lines 16-17).

2. If p lost the leadership, p increases its phase number and p turns off its *SendAliveTimer* by setting it to -1 (line 7) —this causes p to stop sending ALIVE messages.

Figure 3 describes the algorithm by giving the pseudo-code of an (arbitrary) process p, and Table 4 describes the local variables of p. It is easy to translate the pseudo-code of p into an automaton for p. Without loss of generality, we can assume that: (1) for some integer b, each iteration of the repeat forever loop (lines 13–34) takes at most b automaton steps (this is because there are no infinite loops, waiting statements, or similar constructs in lines 14–34), and (2) each iteration of the repeat forever loop takes at least two complete automaton steps.

We now show that the algorithm in Figure 3 implements Ω in system S^{++} and that it is communication-efficient. Henceforth, we consider an arbitrary run of this algorithm in system S^{++}, and s is an eventually

```
CODE FOR EACH PROCESS p:

procedure updateLeader()
    newleader ← ℓ such that (counter[ℓ], ℓ) = min{(counter[q], q) : q ∈ active}
    if newleader ≠ leader then          { if the leader of p changes then }
        if newleader = p then           { if p gains the leadership then }
            SendAliveTimer ← 0              { p sets SendAliveTimer = 0 to start sending ALIVE messages }
        if leader = p then              { if p loses the leadership then }
            phase[p] ← phase[p] + 1         { p increases its phase number and }
            SendAliveTimer ← −1             { p sets SendAliveTimer = −1 to stop sending ALIVE messages }
        leader ← newleader              { p updates its leader variable }

main code
    { Initialization }
    for each q ∈ Π do counter[q] ← 0; phase[q] ← 0
    for each q ∈ Π \ {p} do timeout[q] ← η + 1; timer[q] ← −1
    active ← {p}
    leader ← ⊥

    repeat forever
        updateLeader()

        if SendAliveTimer = 0 then
            send (ALIVE, counter[p], phase[p]) to every process except p
            SendAliveTimer ← η

        for each q ∈ Π \ {p} do
            if receive (ALIVE, qcntr, qph) from q then
                active ← active ∪ {q}
                counter[q] ← max{counter[q], qcntr}
                phase[q] ← max{phase[q], qph}
                timer[q] ← timeout[q]

            if timer[q] = 0 then
                send (ACCUSATION, phase[q]) to q
                active ← active − {q}
                timeout[q] ← timeout[q] + 1
                timer[q] ← −1

            if receive (ACCUSATION, ph) from q then
                if ph = phase[p] then
                    counter[p] ← counter[p] + 1

        if SendAliveTimer > 0 then SendAliveTimer ← SendAliveTimer − 1
        for each q ∈ Π \ {p} do
            if timer[q] > 0 then timer[q] ← timer[q] − 1
```

Figure 3: Communication-efficient implementation of Ω for a system S where all links are fair.

timely source in this run.

Lemma 31 *For every correct process p and every process $q \neq p$, if $q \in$ active$_p$ holds infinitely often then p receives* ALIVE *messages from q infinitely often.*

PROOF. Identical to part (a) of the proof of Lemma 1. □

Observation 32 *For all processes p and q, counter$_p[q]$ and phase$_p[q]$ are monotonically nondecreasing with time.*

Lemma 33 *For every two processes $p \neq q$, if p receives* ALIVE *messages from q infinitely often then q is correct, and for every time t, there is a time after which counter$_p[q] \geq$ counter$^t_q[q]$ and phase$_p[q] \geq$ phase$^t_q[q]$.*

PROOF. (Similar to the proof of Lemma 3 part 1.) Consider two processes $p \neq q$, and suppose that p receives ALIVE messages from q infinitely often. Then q sends such messages infinitely often, and so q is correct. Consider any time t. Eventually p receives a message $m = (\text{ALIVE}, qcntr, qph)$ that is sent by q after time t. Note that *counter*$_q[q]$ and *phase*$_q[q]$ are monotonically nondecreasing. Since q sends m after time t, *qcntr* $\geq$ *counter*$^t_q[q]$ and *qph* $\geq$ *phase*$^t_q[q]$. When p receives m from q, p sets *counter*$_p[q]$ to a value $v \geq$ *qcntr* $\geq$ *counter*$^t_q[q]$, and p sets *phase*$_p[q]$ to a value $v' \geq$ *qph* $\geq$ *phase*$^t_q[q]$. Thereafter, *counter*$_p[q] \geq$ *counter*$^t_q[q]$ and *phase*$_p[q] \geq$ *phase*$^t_q[q]$ (because *counter*$_p[q]$ and *phase*$_p[q]$ are monotonically nondecreasing). □

Lemma 34 *For every correct process p and every process q, if (a) $q \in$ active$_p$ holds infinitely often then (b) q is correct, and for every time t, there is a time after which counter$_p[q] \geq$ counter$^t_q[q]$ and phase$_p[q] \geq$ phase$^t_q[q]$.*

PROOF. (Similar to the proof of Lemma 4.) If $p = q$, condition (b) holds because p is correct, and *counter*$_p[p]$ and *phase*$_p[p]$ are monotonically nondecreasing. Now assume that $p \neq q$ and $q \in$ *active*$_p$ holds infinitely often. By Lemma 31, p receives ALIVE messages from q infinitely often. By Lemma 33, condition (b) holds. □

Lemma 35 *For every distinct correct processes p and q, if p sends a message of type* T *to q infinitely often, then q receives a message of type* T *from p infinitely often.*

PROOF. Let p and q be distinct correct processes, and suppose that p sends a message of type T to q infinitely often. Since the link $p \rightarrow q$ is fair, a message of type T is delivered to q from p infinitely often. Since q is correct, q executes an infinite number of iterations of its repeat forever loop. In each such iteration, q tries to receive one message of each type from every process other than q, including p. Therefore, q receives a message of type T from p infinitely often. □

Recall that s is an eventually timely source in the run under consideration.

Lemma 36 *There is a constant $\alpha > 0$ such that, for all $k \geq 0$ and every time t, process s executes at least k complete iterations of its repeat forever loop during time interval $(t, t + k\alpha]$.*

PROOF. Identical to the proof of Lemma 5. □

Definition 37 *Let $\alpha > 0$ be a constant that satisfies Lemma 36.*

Recall that $\eta \geq 1$ is the "timeout" value of *SendAliveTimer* (see line 17).

Definition 38 *Let $\Delta' = (\eta + 1)\alpha$.*

Lemma 39 *For every process $p \neq s$ and every $k \geq 0$, if s sends an $(\text{ALIVE}, -, k)$ message to p at some time t then s sends another $(\text{ALIVE}, -, k)$ message to p during time interval $(t, t + \Delta']$, or $phase_s[s] > k$ holds at time $t + \Delta'$.*

PROOF. After s executes its initialization code (lines 9-12), s starts its first execution of the repeat forever loop (lines 13-34). Suppose that s sends an $(\text{ALIVE}, -, k)$ message to a process $p \neq s$ at some time t (line 16). Note that $phase_s[s] = k$ at time t, and that in line 17 of the same iteration of its repeat forever loop, s sets $SendAliveTimer_s$ to $\eta \geq 1$.

Consider the first $(\eta+1)$ iterations of the repeat forever loop that s finishes to execute after time t (including the iteration that s is executing at time t). Let t' be the time when s completes the last one of these iterations. By Lemma 36, for every time t, s executes at least $(\eta + 1)$ complete iterations of its repeat forever loop during time interval $(t, t + (\eta + 1)\alpha]$. And so, we have $t' \leq t + (\eta + 1)\alpha$, i.e., $t' \leq t + \Delta'$. Now consider time interval $[t, t']$. There are two possible cases:

1. *During $[t, t']$, s does not set $SendAliveTimer_s$ to -1 in line 7 in the updateLeader procedure.* In this case, it is clear that s does not modify its $phase_s[s]$ during $[t, t']$ (this is because s modifies $phase_s[s]$ only in line 6 in the *updateLeader* procedure), and so $phase_s[s] = k$ during the entire time interval $[t, t']$.

   We claim that by the end of the η-th iteration of the $(\eta + 1)$ iterations that we are considering, s sets $SendAliveTimer_s \leftarrow 0$. In fact, either s does this by executing line 4 of the *updateLeader* procedure in one of the first η iterations, or s decrements its $SendAliveTimer_s$ from η by 1 (in line 32) in each one of the first η iterations. In either case, by the end of the η-th iteration, s sets $SendAliveTimer_s \leftarrow 0$.

   Thus, by the end of the $(\eta + 1)$-th iteration, s finds that $SendAliveTimer_s = 0$ (in line 15), and it sends an $(\text{ALIVE}, -, k)$ message to p (in line 16). This sending must occur at least one step after s sends the $(\text{ALIVE}, -, k)$ message to p at time t, so, by the Maximum Rate of Execution property, it must occur after time t. Moreover, this sending occurs by time $t' \leq t + \Delta'$. So s sends an $(\text{ALIVE}, -, k)$ message to p during interval $(t, t + \Delta']$.

2. *During $[t, t']$, s sets $SendAliveTimer_s$ to -1 in line 7 in the updateLeader procedure.* Note that during the execution of this procedure, s increments $phase_s[s]$ in line 6. This increment must occur at least one step after s sends the $(\text{ALIVE}, -, k)$ message to p at time t (because after sending and before incrementing, s executes steps to try to receive ALIVE and ACCUSATION messages). Thus, by the Maximum Rate of Execution property, the incrementing must occur after time t. Moreover, this

increment must occur by time t', so it happens during time interval $(t, t']$, which is contained in interval $(t, t + \Delta']$. Since *phase*$_s[s] = k$ at time t, *phase*$_s[s]$ is incremented during interval $(t, t + \Delta']$, and it is monotonically nondecreasing, we have $phase_s[s] > k$ at time $t + \Delta'$.

From the above, we conclude that s sends an $(\text{ALIVE}, -, k)$ message to p during interval $(t, t + \Delta']$, or $phase_s[s] > k$ holds at time $t + \Delta'$.

□

Lemma 40 *There is a constant Δ and a time t_Δ such that, for all processes p, if s sends a message m to p at some time $t \geq t_\Delta$, then m is delivered to p from s by time $t + \Delta$.*

PROOF. This follows immediately from the fact that s is an eventually timely source, and therefore all its output links are eventually timely. □

Lemma 41 *There is a constant $\epsilon > 0$ such that, for every $k \geq 1$ and every process p, p takes at least $k\epsilon$ time to execute k complete iterations of its repeat forever loop.*

PROOF. Identical to the proof of Lemma 13. □

Definition 42 *Let Δ, t_Δ and ϵ be constants that satisfy 40 and 41, respectively.*

Definition 43 *Let $\zeta = \lceil (\Delta' + \Delta)/\epsilon \rceil + 3$.*

We now show that at the eventually timely source s, *counter*$_s[s]$ is bounded. To prove this, (1) we note that s increments *counter*$_s[s]$ only if a process times out on s, (2) we distinguish two types of such timeouts on s, which we call "'proper" and "improper", (3) we prove that proper timeouts on s do not affect *counter*$_s[s]$ (so only improper timeouts on s can cause s to increment *counter*$_s[s]$), and (4) we show that the number of improper timeouts on s is finite. We now proceed with this proof (Lemmas 45-48).

Suppose that a process p times out on s. If this timeout was started after time t_Δ and its value was at least ζ, we say that it is "proper"; otherwise we say it is "improper". More precisely,

Definition 44 *Suppose that*

(1) a process p executes line 24 with $q = s$ and timer$_p[s] = 0$ at some time t_e,

(2) p sets timer$_p[s]$ to timeout$_p[s]$ in line 23 at some time $t_s \leq t_e$, and

(3) p does not set timer$_p[s]$ in line 23 during time interval $(t_s, t_e]$.

We say this timeout of p on s is proper *if and only if (a) $t_s \geq t_\Delta$ and (b) timeout$_p[s] \geq \zeta$ at time t_s. A timeout that is not proper is* improper.

Lemma 45 *For every process p, the number of improper timeouts of p on s is finite.*

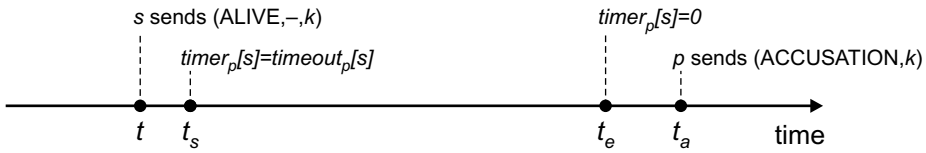

Figure 4: Timeline of events in proof of Lemma 47.

PROOF. Let p be any process. If p times out on s only finitely often, the lemma trivially holds. Now suppose p times out on s infinitely often, i.e., p executes line 24 with *timer*$_p[s] = 0$ infinitely many times. Note that each time this occurs, p increases *timeout*$_p[s]$ (in line 27). So there is a time after which *timeout*$_p[s] > \zeta$. Thus, there is a time after which every timeout of p on s is proper. □

Definition 46 *An* $(\text{ACCUSATION}, ph)$ *message that is sent to* s *is outdated if* $ph <$ *phase*$_s[s]$ *at the time this message is sent.*

Note that any outdated $(\text{ACCUSATION}, ph)$ message that s receives does not affect *counter*$_s[s]$. In fact, if s receives an $(\text{ACCUSATION}, ph)$ message that is outdated, then *phase*$_s[s] > ph$ at the time t this message was sent to s, so *phase*$_s[s] > ph$ also holds at the time when s executes line 30 of its code (because *phase*$_s[s]$ is monotonically nondecreasing). Thus, s does *not* execute line 31, i.e., it does not modify *counter*$_s[s]$.

Lemma 47 *Suppose a process* p *times out on* s *(in line 24). If this timeout is proper, then the* $(\text{ACCUSATION}, -)$ *message that* p *sends to* s *as a consequence of this timeout (in line 25) is outdated.*

PROOF. Suppose some process p times out on s, and that this timeout is proper. More precisely, suppose that

(1) p executes line 24 with $q = s$ and *timer*$_p[s] = 0$ at some time t_e,

(2) p sets *timer*$_p[s]$ to *timeout*$_p[s]$ in line 23 at some time $t_s \leq t_e$,

(3) p does not set *timer*$_p[s]$ in line 23 during time interval $(t_s, t_e]$, and

(4) $t_s \geq t_\Delta$ and *timeout*$_p[s] \geq \zeta$ at time t_s.

Suppose that the above timeout causes p to send some $(\text{ACCUSATION}, k)$ message to s, and let $t_a \geq t_e$ be the time when this occurs (in line 25). Figure 4 shows a timeline with times t_s, t_e, and t_a. We must prove that this $(\text{ACCUSATION}, k)$ is outdated, that is, we must show that *phase*$_s[s] > k$ at time t_a. Suppose, for contradiction, that *phase*$_s[s] \leq k$ at time t_a. Since *phase*$_s[s]$ is monotonically nondecreasing and $t_e \leq t_a$, *phase*$_s[s] \leq k$ also holds at time t_e.

We first note that p executes at least $\zeta - 1$ complete iterations of its repeat forever loop during time interval $[t_s, t_e]$. This follows from assumptions (1), (2), (3) and (4) above, and the fact that p decreases *timer*$_p[s]$ by exactly 1 in each repeat forever loop iteration (in line 34).

By Lemma 41, p takes at least $\epsilon(\zeta - 1)$ time to execute $(\zeta - 1)$ complete iterations of its repeat forever loop. Thus, from the above, $t_e \geq t_s + \epsilon(\zeta - 1)$. Since $\zeta = \lceil(\Delta' + \Delta)/\epsilon\rceil + 3$, we have $t_e \geq t_s + \Delta' + \Delta + 2\epsilon$.

CLAIM 1: *p does not receive any* $(\text{ALIVE}, -, -)$ *messages from s during time interval* $(t_s, t_e]$. To see this, note that such a receipt would cause p to set *timer*$_p[s]$ in line 23, and this would happen during $(t_s, t_e]$ since, at time t_e, p executes line 24. This would violate assumption (3).

Since p sends an $(\text{ACCUSATION}, k)$ message to s at time t_a in line 25, *phase*$_p[s] = k$ at time t_a. So *phase*$_p[s] = k$ also holds at time t_e when p executes line 24.

CLAIM 2: *p receives at least one* $(\text{ALIVE}, -, k)$ *message from s by time t_s*. Indeed, if $k > 0$ then the only way for p to have *phase*$_p[s] = k$ at time t_e is by receiving $(\text{ALIVE}, -, k)$ from s by time t_e. By Claim 1, p must receive such a message by time t_s. For $k = 0$, note that by time t_s, p must receive some $(\text{ALIVE}, -, k')$ message from s that causes p to set *timer*$_p[s]$ in line 23 at time t_s. Moreover, k' cannot be greater than 0 otherwise *phase*$_p[s] > 0$ at time t_s, so *phase*$_p[s] > 0$ at time t_e (since *phase*$_p[s]$ is monotonically nondecreasing), contradicting that *phase*$_p[s] = k = 0$ at time t_e. Thus $k' = 0$. This proves Claim 2.

From Claim 2, s sends an $(\text{ALIVE}, -, k)$ message to p at some time $t \leq t_s$. This implies that *phase*$_s[s] = k$ at time t. Since *phase*$_s[s] \leq k$ at time t_e (where $t_e > t_s \geq t$), and *phase*$_s[s]$ is monotonically nondecreasing, we conclude that *phase*$_s[s] = k$ during the entire time interval $[t, t_e]$.

Thus, by repeated applications of Lemma 39 starting at time t, it is clear that from time t and up to time t_e, s sends an $(\text{ALIVE}, -, k)$ message to p at least once every Δ' time; more precisely, s sends at least one $(\text{ALIVE}, -, k)$ message to p during each time interval $(\tau, \tau + \Delta']$ contained in interval $[t, t_e]$.

Since time interval $(t_s, t_s + \Delta']$ is contained in interval $[t, t_e]$ (because $t \leq t_s$ and $t_s + \Delta' \leq t_e$), s sends an $(\text{ALIVE}, -, k)$ message to p during $(t_s, t_s + \Delta']$. By assumption (4), $t_s \geq t_\Delta$. Thus, by Lemma 40 and the definitions of t_Δ and Δ, this $(\text{ALIVE}, -, k)$ message is delivered to p from s during time interval $(t_s, t_s + \Delta' + \Delta]$.

CLAIM 3: *p executes at least one complete iteration of its repeat forever loop during time interval* $[t_s + \Delta' + \Delta, t_e]$. To see this, recall that p executes *at least* $\zeta - 1$ complete iterations of its repeat forever loop during time interval $[t_s, t_e]$. Moreover, during time interval $[t_s, t_s + \Delta' + \Delta]$, p executes *at most* $\lceil (\Delta' + \Delta)/\epsilon \rceil = \zeta - 3$ complete iterations of its repeat forever loop (this follows from the definition of ϵ). This implies Claim 3.

Since an $(\text{ALIVE}, -, k)$ message is delivered to p from s during time interval $(t_s, t_s + \Delta' + \Delta]$, and p executes at least one complete iteration of its repeat forever loop during time interval $[t_s + \Delta' + \Delta, t_e]$, we conclude that p receives some $(\text{ALIVE}, -, -)$ message from s during interval $(t_s, t_e]$ —a contradiction to Claim 1. □

Lemma 48 *counter*$_s[s]$ *is bounded.*

PROOF. Note that s increases its *counter*$_s[s]$ only if it receives an $(\text{ACCUSATION}, -)$ message (lines 29-31). There are two kinds of such $(\text{ACCUSATION}, -)$ messages: (a) those that are sent to s as a consequence of a *proper* timeout on s, and (b) those that are sent to s as a consequence of an *improper* timeout on s. By Lemma 47, all the $(\text{ACCUSATION}, -)$ messages of kind (a) are outdated. As we previously observed, such messages do not affect *counter*$_s[s]$. Thus only those messages of kind (b) may cause s to increment $counter_s[s]$. By Lemma 45, the number of improper timeouts on s is finite. Since each timeout on s causes at most one $(\text{ACCUSATION}, -)$ message to be sent to s, the number of $(\text{ACCUSATION}, -)$ messages of kind (b) is finite. Therefore *counter*$_s[s]$ is bounded. □

Definition 49 *For every process p, let c_p be the largest value of counter$_p[p]$ in the run that we consider ($c_p = \infty$ if counter$_p[p]$ is unbounded). Let ℓ be the process such that $(c_\ell, \ell) = \min\{(c_p, p) : p$ is a correct process$\}$.*

By definition, ℓ is a correct process. Furthermore, by Lemma 48, *counter*$_s[s]$ is bounded, i.e., $c_s < \infty$. Thus, $c_\ell < \infty$, i.e., *counter*$_\ell[\ell]$ is bounded.

Lemma 50 *For every correct process p, if there is a time after which $\ell \in$ active$_p$, then there is a time after which leader$_p = \ell$.*

PROOF. (Similar to the proof of Lemma 21.) Let p be any correct process, and suppose that there is a time after which $\ell \in$ *active*$_p$. We claim that for every $q \neq \ell$, (i) there is a time after which $q \notin$ *active*$_p$, or (ii) there is a time after which $(\textit{counter}_p[\ell], \ell) < (\textit{counter}_p[q], q)$. From the way p sets *leader*$_q$ in the *updateLeader* procedure, this claim implies there is a time after which *leader*$_p = \ell$.

To show the claim, consider any process $q \neq \ell$, and suppose that condition (i) does not hold, i.e., suppose that $q \in$ *active*$_p$ holds infinitely often. We now show that condition (ii) is satisfied. By Lemma 34, q is correct, and for every time t, there is a time after which $\textit{counter}_p[q] \geq \textit{counter}_q^t[q]$. There are two cases:

(1) *counter*$_q[q]$ *is bounded.* In this case, $c_q < \infty$, and so there is a time t when $\textit{counter}_q^t[q] = c_q$. So there is a time after which $\textit{counter}_p[q] \geq c_q$. Recall that q is correct and $q \neq \ell$, and so by the definition of ℓ, we have $(c_\ell, \ell) < (c_q, q)$. Since $\textit{counter}_p[\ell] \leq c_\ell$ (always), there is a time after which $(\textit{counter}_p[\ell], \ell) \leq (c_\ell, \ell) < (c_q, q) \leq (\textit{counter}_p[q], q)$.

(2) *counter*$_q[q]$ *is unbounded.* In this case, *counter*$_p[q]$ is also unbounded. So there is a time after which $\textit{counter}_p[\ell] \leq c_\ell < \textit{counter}_p[q]$.

So, in both cases, there is a time after which $(\textit{counter}_p[\ell], \ell) < (\textit{counter}_p[q], q)$, i.e., condition (ii) holds. □

Observation 51 *For every correct process p, there is a time after which $p \in$ active$_p$.*

PROOF. When p executes its initialization code, it sets *active*$_p$ to $\{p\}$. Thereafter, p never removes itself from *active*$_p$. □

Corollary 52 *There is a time after which leader$_\ell = \ell$.*

PROOF. By Observation 51, there is a time after which $\ell \in$ *active*$_\ell$. The result now follows from Lemma 50. □

Corollary 53 *There is a time after which phase$_\ell[\ell]$ stops changing.*

PROOF. Note that ℓ changes *phase*$_\ell[\ell]$ only when it considers that it lost the leadership (in lines 5-6), and each time this occurs ℓ sets *leader*$_\ell \neq \ell$ (in line 8). By Corollary 52, this can happen only a finite number of times. □

Definition 54 *Let ℓphase be the final value of phase$_\ell[\ell]$.*

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

Lemma 55 *For every correct process q, there is a time after which $\ell \in$ active$_q$.*

PROOF. Let q be any correct process. If $q = \ell$ then, by Corollary 52, there is a time after which $\ell \in$ *active*$_\ell$. Now suppose $q \neq \ell$. By Corollary 52 and the definitions of ℓ*phase* and ℓ, ℓ sends messages of form $(\text{ALIVE}, -, \ell\textit{phase})$ to q infinitely often, and these are the only messages of type ALIVE that ℓ sends to q infinitely often. By Lemma 35, q receives messages of type ALIVE from ℓ infinitely often. Thus, q receives messages of form $(\text{ALIVE}, -, \ell\textit{phase})$ from ℓ infinitely often. Therefore, (*) there is a time after which q has *phase*$_q[\ell] = \ell$*phase*. Moreover, q adds ℓ to *active*$_q$ infinitely often. We claim that q removes ℓ from *active*$_q$ only finitely often, and so the lemma follows. Suppose, for contradiction, that q removes ℓ from *active*$_q$ infinitely often. Then, q sends $(\text{ACCUSATION}, -)$ messages to ℓ infinitely often. By Lemma 35, ℓ receives $(\text{ACCUSATION}, -)$ messages from q infinitely often. By (*), there is a time after which the only $(\text{ACCUSATION}, -)$ messages that q sends are $(\text{ACCUSATION}, \ell\textit{phase})$ messages. Thus, ℓ receives $(\text{ACCUSATION}, \ell\textit{phase})$ messages from q infinitely often. So, ℓ eventually increments *counter*$_\ell[\ell]$ to a value greater than c_ℓ —a contradiction to the definition of c_ℓ. □

By Lemmas 50 and 55, we have

Lemma 56 *For every correct process q, there is a time after which leader$_q = \ell$.*

Lemma 57 *There is a time after which only ℓ sends messages.*

PROOF. There are only two types of messages: ALIVE and ACCUSATION. When a process p considers that it lost the leadership, it stops sending ALIVE messages (by setting its *SendAliveTimer* to -1 in line 7). Furthermore, p resumes sending messages only if it considers itself to be the leader again (lines 3-4) and it sets *leader*$_p = p$ (in line 8). So, by Lemma 56, there is a time after which only ℓ sends ALIVE messages.

We claim that only a finite number of ACCUSATION messages are sent. To see this, note that when a process p sends an ACCUSATION message to a process q (in line 25), p "turns off" *timer*$_p[q]$ by setting it to -1 (in line 28). After this occurs, p can send another ACCUSATION message to q only if p "turns on" *timer*$_p[q]$ again (in line 23), and this happens only if p receives an ALIVE message from q (in line 19). Thus, p can send an infinite number of ACCUSATION messages to q only if p receives an infinite number of ALIVE messages from q. Since there is a time after which only ℓ sends ALIVE messages, p can send an infinite number of ACCUSATION messages only to ℓ. But p sends only a finite number of ACCUSATION messages to ℓ: This is because each time p sends an ACCUSATION message to ℓ, p removes ℓ from *active*$_p$, and from Lemma 55, this can happen only a finite number of times. Thus each process p sends only a finite number of ACCUSATION messages to every process. □

From Lemmas 56 and 57, we get the following result:

Theorem 58 *The algorithm in Figure 3 implements Ω in system S^{++}, and it is communication-efficient.*

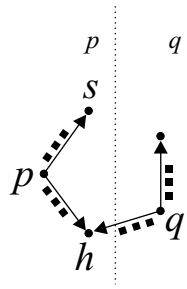

Figure 5: Partitioning that may occur if we run the algorithm of Figure 3 in system S^+.

6.2 Implementing Ω in system S^+

We now describe a communication-efficient algorithm for Ω for system S^+. Recall that in S^+ there is an eventually timely source and a correct process whose input and output links are fair.

Our starting point is the algorithm for system S^{++} that we gave in the previous section (Figure 3). We first note that this algorithm does not work in system S^+ because in S^+ some links can experience *arbitrary* message losses (in contrast to S^{++} where all the 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, because the link $p \rightarrow q$ may have crashed. 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 scenario, illustrated in Figure 5, can occur as follows: (a) the eventually timely 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 chooses 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 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.

The algorithm that incorporates the above ideas is shown in Figure 6. In this algorithm, there are $n + 2$ message types: ALIVE, CHECK, and ACCUSATION-p for each process p.

Figure 6 describes the algorithm by giving the pseudo-code of an arbitrary process p, and Table 4 describes the local variables of p (this algorithm has the same variables with the same meaning in as in the previous algorithm). It is easy to translate the pseudo-code of p into an automaton for p. Without loss of generality, we can assume that: (1) for some integer b, each iteration of the repeat forever loop (lines 13–43) takes at most b automaton steps (this is because there are no infinite loops, waiting statements, or similar constructs in lines 14–43), and (2) each iteration of the repeat forever loop takes at least two complete automaton steps.

We now show that the algorithm in Figure 6 implements Ω in system S^+, and that it is communication-efficient. Henceforth, we consider an arbitrary run of this algorithm in system S^+. Let s be an eventually timely source and h be a fair hub, in this run.

Lemma 59 *For every correct process p and every process $q \neq p$, if $q \in active_p$ holds infinitely often then p receives* ALIVE *messages from q infinitely often.*

PROOF. Identical to part (a) of the proof of Lemma 1. □

Observation 60 *For all processes p and q, $counter_p[q]$ and $phase_p[q]$ are monotonically nondecreasing with time.*

Lemma 61 *For every two processes $p \neq q$, if p receives* ALIVE *messages from q infinitely often then q is correct, and for every time t, there is a time after which $counter_p[q] \geq counter_q^t[q]$ and $phase_p[q] \geq phase_q^t[q]$.*

PROOF. Identical to the proof of Lemma 33. □

Lemma 62 *For every correct process p and every process q, if (a) $q \in active_p$ holds infinitely often then (b) q is correct, and for every time t, there is a time after which $counter_p[q] \geq counter_q^t[q]$ and $phase_p[q] \geq phase_q^t[q]$.*

PROOF. (Similar to the proof of Lemma 34.) If $p = q$, condition (b) holds because p is correct, and $counter_p[p]$ and $phase_p[p]$ are monotonically nondecreasing. Now assume that $p \neq q$ and $q \in active_p$ holds infinitely often. By Lemma 59, p receives ALIVE messages from q infinitely often. By Lemma 61, condition (b) holds. □

Lemma 63 *For every correct process $p \neq h$, (1) if p sends a message of type* T *to h infinitely often, then h receives a message of type* T *from p infinitely often, and (2) if h sends a message of type* T *to p infinitely often, then p receives a message of type* T *from h infinitely often.*

CODE FOR EACH PROCESS p:

```
procedure updateLeader()
    newleader ← ℓ such that (counter[ℓ], ℓ) = min{(counter[q], q) : q ∈ active}
    if newleader ≠ leader then          { if the leader of p changes then }
        if newleader = p then           { if p gains the leadership then }
            SendAliveTimer ← 0              { p sets SendAliveTimer = 0 to start sending ALIVE messages }
        if leader = p then              { if p loses the leadership then }
            phase[p] ← phase[p] + 1         { p increases its phase number and }
            SendAliveTimer ← −1             { p sets SendAliveTimer = −1 to stop sending ALIVE messages }
        leader ← newleader              { p updates its leader variable }

main code
    { Initialization }
    for each q ∈ Π do counter[q] ← 0; phase[q] ← 0
    for each q ∈ Π \ {p} do timeout[q] ← η + 1; timer[q] ← −1
    active ← {p}
    leader ← ⊥

    repeat forever
        updateLeader()

        if SendAliveTimer = 0 then
            send (ALIVE, counter[p], phase[p]) to every process except p
            SendAliveTimer ← η

        for each q ∈ Π \ {p} do
            if receive (ALIVE, qcntr, qph) from q then
                active ← active ∪ {q}
                counter[q] ← max{counter[q], qcntr}
                phase[q] ← max{phase[q], qph}
                timer[q] ← timeout[q]
                if q ≠ leader and p ≠ leader then
                    send (CHECK, leader, phase[leader]) to q

            if receive (CHECK, r, rph) from q then
                if timer[r] = −1 then
                    phase[r] ← max{phase[r], rph}
                    timer[r] ← timeout[r]

            if timer[q] = 0 then
                send (ACCUSATION-q, phase[q]) to every process except p
                active ← active − {q}
                timeout[q] ← timeout[q] + 1
                timer[q] ← −1

            for each r ∈ Π do
                if receive (ACCUSATION-r, ph) from q then
                    if r = p then
                        if ph = phase[p] then
                            counter[p] ← counter[p] + 1
                    else send (ACCUSATION-r, ph) to r

        if SendAliveTimer > 0 then SendAliveTimer ← SendAliveTimer − 1
        for each q ∈ Π \ {p} do
            if timer[q] > 0 then timer[q] ← timer[q] − 1
```

Figure 6: Communication-efficient implementation of Ω for system S^+.

PROOF. (Similar to the proof of Lemma 35.) Let p be a correct process such that $p \neq h$. (1) First, suppose that p sends a message of type T to h infinitely often. Since h is fair hub, h is correct and link $p \rightarrow h$ is fair. Thus, a message of type T is delivered to h from p infinitely often. Since h is correct, h executes an infinite number of iterations of its repeat forever loop. In each such iteration, h tries to receive one message of each type from every process other than h, including p. Therefore, h receives a message of type T from p infinitely often.

(2) Now suppose that h sends a message of type T to p infinitely often. This case is identical to case (1) except that we exchange the roles of p and h. □

Recall that s is an eventually timely source in the run under consideration.

Lemma 64 *There is a constant $\alpha > 0$ such that, for all $k \geq 0$ and every time t, process s executes at least k complete iterations of its repeat forever loop during time interval $(t, t + k\alpha]$.*

PROOF. Identical to the proof of Lemma 5. □

Definition 65 *Let $\alpha > 0$ be a constant that satisfies Lemma 64.*

Recall that $\eta \geq 1$ is the "timeout" value of *SendAliveTimer* (see line 17).

Definition 66 *Let $\Delta' = (\eta + 1)\alpha$.*

Lemma 67 *For every process $p \neq s$ and every $k \geq 0$, if s sends an $(\text{ALIVE}, -, k)$ message to p at some time t then s sends another $(\text{ALIVE}, -, k)$ message to p during time interval $(t, t + \Delta']$, or $phase_s[s] > k$ holds at time $t + \Delta'$.*

PROOF. (Similar to the proof of Lemma 39 noting that, in case 1 of that proof, s cannot modify $phase_s[s]$ in line 28 because no process ever sends $(\text{CHECK}, s, -)$ to s.)

After s executes its initialization code (lines 9-12), s starts its first execution of the repeat forever loop (lines 13-43). Suppose that s sends an $(\text{ALIVE}, -, k)$ message to a process $p \neq s$ at some time t (line 16). Note that $phase_s[s] = k$ at time t, and that in line 17 of the same iteration of its repeat forever loop, s sets $SendAliveTimer_s$ to $\eta \geq 1$.

Consider the first $(\eta+1)$ iterations of the repeat forever loop that s finishes to execute after time t (including the iteration that s is executing at time t). Let t' be the time when s completes the last one of these iterations. By Lemma 64, for every time t, s executes at least $(\eta + 1)$ complete iterations of its repeat forever loop during time interval $(t, t + (\eta + 1)\alpha]$. And so, we have $t' \leq t + (\eta + 1)\alpha$, i.e., $t' \leq t + \Delta'$. Now consider time interval $[t, t']$. There are two possible cases:

1. *During $[t, t']$, s does not set $SendAliveTimer_s$ to -1 in line 7 in the updateLeader procedure.* In this case, s does not modify its $phase_s[s]$ during $[t, t']$: the only places where s could possibly modify $phase_s[s]$ is in lines 6 or 28, but s does not execute line 6 during $[t, t']$ since s does not execute line 7 by assumption, and s does not modify $phase_s[s]$ in line 28 because no process ever sends $(\text{CHECK}, s, -)$ to s due to the check in line 24. Therefore, $phase_s[s] = k$ during the entire time interval $[t, t']$.

We claim that by the end of the η-th iteration of the $(\eta + 1)$ iterations that we are considering, s sets *SendAliveTimer*$_s \leftarrow 0$. In fact, either s does this by executing line 4 of the *updateLeader* procedure in one of the first η iterations, or s decrements its *SendAliveTimer*$_s$ from η by 1 (in line 41) in each one of the first η iterations. In either case, by the end of the η-th iteration, s sets *SendAliveTimer*$_s \leftarrow 0$.

Thus, by the end of the $(\eta+1)$-th iteration, s finds that *SendAliveTimer*$_s = 0$ (in line 15), and it sends an $(\text{ALIVE}, -, k)$ message to p (in line 16). This sending must occur at least one step after s sends the $(\text{ALIVE}, -, k)$ message to p at time t, so, by the Maximum Rate of Execution property, it must occur after time t. Moreover, this sending occurs by time $t' \leq t + \Delta'$. So s sends an $(\text{ALIVE}, -, k)$ message to p during interval $(t, t + \Delta']$.

2. *During $[t, t']$, s sets SendAliveTimer$_s$ to -1 in line 7 in the updateLeader procedure.* Note that during the execution of this procedure, s increments *phase*$_s[s]$ in line 6. This increment must occur at least one step after s sends the $(\text{ALIVE}, -, k)$ message to p at time t (because after sending and before incrementing, s executes steps to try to receive ALIVE and ACCUSATION messages). Thus, by the Maximum Rate of Execution property, the incrementing must occur after time t. Moreover, this increment must occur by time t', so it happens during time interval $(t, t']$, which is contained in interval $(t, t + \Delta']$. Since *phase*$_s[s] = k$ at time t, *phase*$_s[s]$ is incremented during interval $(t, t + \Delta']$, and it is monotonically nondecreasing, we have $phase_s[s] > k$ at time $t + \Delta'$.

From the above, we conclude that s sends an $(\text{ALIVE}, -, k)$ message to p during interval $(t, t + \Delta']$, or $phase_s[s] > k$ holds at time $t + \Delta'$. □

Lemma 68 *There is a constant Δ and a time t_Δ such that, for all processes p, if s sends a message m to p at some time $t \geq t_\Delta$, then m is delivered to p from s by time $t + \Delta$.*

PROOF. This follows immediately from the fact that s is an eventually timely source, and therefore all its output links are eventually timely. □

Lemma 69 *There is a constant $\epsilon > 0$ such that, for every $k \geq 1$ and every process p, p takes at least $k\epsilon$ time to execute k complete iterations of its repeat forever loop.*

PROOF. Identical to the proof of Lemma 13. □

Definition 70 *Let Δ, t_Δ and ϵ be constants that satisfy 68 and 69 respectively.*

Definition 71 *Let $\zeta = \lceil (\Delta' + \Delta)/\epsilon \rceil + 3$.*

Lemma 72 *For all processes p and r and every $k \geq 0$, if p receives a (CHECK, r, k) message at some time t then r sends an $(\text{ALIVE}, -, k)$ message by time t.*

PROOF. Let p and r be processes and $k \geq 0$. Suppose that p receives a (CHECK, r, k) message at some time t. For contradiction, suppose r does not send an $(\text{ALIVE}, -, k)$ message by time t. Let r' be the process to first send a (CHECK, r, k) message, and let t' be the time when this happens. Note that $t' \leq t$ and, at time t', *phase*$_{r'}[r] = k$. Then $r' \neq r$ since a process does not send a CHECK message for itself due to the check in line 24. There are now two possibilities.

- If $k > 0$ then, at time t', *phase*$_{r'}[r] = k \geq 1$. There are only two places where r' can set *phase*$_{r'}[r]$ to k: line 22 or 28. In the first case, r' previously receives $(\text{ALIVE}, -, k)$ from r, which contradicts the assumption that r does not send an $(\text{ALIVE}, -, k)$ message by time t. In the second case, r' previously receives (CHECK, r, k), which means some process sends (CHECK, r, k) before time t', which contradicts the choice of r'.

- If $k = 0$ then, at time t', *leader*$_{r'} = r$, and so r' previously set *leader*$_{r'}$ to r. When this happened, $r \in$ *active*$_{r'}$ (because the leader is picked among processes in *active*). Since $r' \neq r$, r' previously added r to *active*, and so r' previously received a $(\text{ALIVE}, -, k')$ message from r for some k'. Then, $k' = 0$ (otherwise upon receiving such a message r' sets *phase*$[r] > 0$, and so at time t', *phase*$_{r'}[r] > 0$, contradicting the fact that at time t', *phase*$_{r'}[r] = k = 0$). Thus, r' receives a $(\text{ALIVE}, -, k)$ message from r by time t, which contradicts the fact that r does not send an $(\text{ALIVE}, -, k)$ message by time t.

□

Definition 73 *Suppose that*

(1) a process p executes line 30 with $q = s$ and timer$_p[s] = 0$ at some time t_e,

(2) p sets timer$_p[s]$ to timeout$_p[s]$ in line 23 or 29 at some time $t_s \leq t_e$, and

(3) p does not set timer$_p[s]$ in line 23 or 29 during time interval $(t_s, t_e]$.

We say this timeout of p on s is proper *if and only if (a) $t_s \geq t_\Delta$ and (b) timeout$_p[s] \geq \zeta$ at time t_s. A timeout that is not proper is* improper.

Lemma 74 *For every process p, the number of improper timeouts of p on s is finite.*

PROOF. Identical to the proof of Lemma 45. □

Definition 75 *An $(\text{ACCUSATION-}s, ph)$ message is outdated if ph $<$ phase$_s[s]$ at the time this message is sent.*

Note that any outdated $(\text{ACCUSATION-}s, ph)$ message that s receives does not affect *counter*$_s[s]$. In fact, if s receives an $(\text{ACCUSATION-}s, ph)$ message that is outdated, then *phase*$_s[s] >$ *ph* at the time t this message was sent to s, so *phase*$_s[s] >$ *ph* also holds at the time when s executes line 38 of its code (because *phase*$_s[s]$ is monotonically non-decreasing). Thus, s does not execute line 39, i.e., it does not modify *counter*$_s[s]$.

Lemma 76 *Suppose a process p times out on s (in line 30). If this timeout is proper, then every $(\text{ACCUSATION-}s, -)$ message that p sends in line 31 as a consequence of this timeout is outdated.*

PROOF. Suppose some process p times out on s, and that this timeout is proper. More precisely, suppose that

(1) p executes line 30 with $q = s$ and *timer*$_p[s] = 0$ at some time t_e,

(2) p sets *timer*$_p[s]$ to *timeout*$_p[s]$ in line 23 or 29 at some time $t_s \leq t_e$,

(3) p does not set *timer*$_p[s]$ in line 23 or 29 during time interval $(t_s, t_e]$, and

(4) $t_s \geq t_\Delta$ and *timeout*$_p[s] \geq \zeta$ at time t_s.

Suppose that this timeout causes p to send some $(\text{ACCUSATION-}s, k)$ message, and let $t_a \geq t_e$ be the time when this occurs (in line 31). We must prove that this $(\text{ACCUSATION-}s, k)$ is outdated, that is, we must show that *phase*$_s[s] > k$ at time t_a. Suppose, for contradiction, that *phase*$_s[s] \leq k$ at time t_a. Since *phase*$_s[s]$ is monotonically nondecreasing and $t_e \leq t_a$, *phase*$_s[s] \leq k$ also holds at time t_e.

We first note that p executes at least $\zeta - 1$ complete iterations of its repeat forever loop during time interval $[t_s, t_e]$. This follows from assumptions (1), (2), (3) and (4) above, and the fact that p decreases *timer*$_p[s]$ by exactly 1 in each repeat forever loop iteration (in line 43).

By Lemma 69, p takes at least $\epsilon(\zeta - 1)$ time to execute $(\zeta - 1)$ complete iterations of its repeat forever loop. Thus, from the above, $t_e \geq t_s + \epsilon(\zeta - 1)$. Since $\zeta = \lceil (\Delta' + \Delta)/\epsilon \rceil + 3$, we have $t_e \geq t_s + \Delta' + \Delta + 2\epsilon$.

CLAIM 1: *p does not receive any $(\text{ALIVE}, -, -)$ messages from s, or any $(\text{CHECK}, s, -)$ messages, during time interval $(t_s, t_e]$.* To see this, note that such a receipt would cause p to set *timer*$_p[s]$ in line 23 or 29, and this would happen during $(t_s, t_e]$ since, at time t_e, p executes line 30. This would violate assumption (3).

Since p sends an $(\text{ACCUSATION-}s, k)$ message to s at time t_a in line 31, *phase*$_p[s] = k$ at time t_a. So *phase*$_p[s] = k$ also holds at time t_e when p executes line 30.

CLAIM 2: *s sends at least one $(\text{ALIVE}, -, k)$ message at some time $t \leq t_s$.* There are two possibilities:

- If $k > 0$ then the only way for p to have *phase*$_p[s] = k$ at time t_e is by receiving $(\text{ALIVE}, -, k)$ from s, or receiving (CHECK, s, k) from some process, and this must happen by time t_e. From Claim 1, this receipt must actually happen by time t_s. If p receives $(\text{ALIVE}, -, k)$ from s by time t_s then s sends $(\text{ALIVE}, -, k)$ at some time $t \leq t_s$. If p receives (CHECK, s, k) from some process by time t_s then, by Lemma 72, s also sends $(\text{ALIVE}, -, k)$ at some time $t \leq t_s$.

- If $k = 0$ then note that initially *timer*$_p[s] = -1$ and at time t_e, *timer*$_p[s] = 0$. The only way for p to change *timer*$_p[q]$ from -1 to a nonnegative value is for p to receive $(\text{ALIVE}, -, k')$ from s or (CHECK, s, k') from some process, for some k'. This happens by time t_e, and so from Claim 1, it happens by time t_s. Moreover, $k' = 0$, otherwise upon receiving such a message, p sets *phase*$_p[s]$ to a positive value by time t_s, and so *phase*$_p[s] \neq 0$ at time t_a, a contradiction. Thus, by time t_s, p receives $(\text{ALIVE}, -, 0)$ from s or $(\text{CHECK}, s, 0)$ from some process. In the first case, s sends $(\text{ALIVE}, -, 0)$ at some time $t \leq t_s$. In the second case, by Lemma 72, s also sends $(\text{ALIVE}, -, 0)$ at some time $t \leq t_s$.

This shows Claim 2.

Claim 2 implies that *phase*$_s[s] = k$ at time t. Since *phase*$_s[s] \leq k$ at time t_e (where $t_e > t_s \geq t$), and *phase*$_s[s]$ is monotonically nondecreasing, we conclude that *phase*$_s[s] = k$ during the entire time interval $[t, t_e]$.

Thus, by repeated applications of Lemma 67 starting at time t, it is clear that from time t and up to time t_e, s sends an $(\text{ALIVE}, -, k)$ message to p at least once every Δ' time; more precisely, s sends at least one $(\text{ALIVE}, -, k)$ message to p during each time interval $(\tau, \tau + \Delta']$ contained in interval $[t, t_e]$.

Since time interval $(t_s, t_s + \Delta']$ is contained in interval $[t, t_e]$ (because $t \leq t_s$ and $t_s + \Delta' \leq t_e$), s sends an $(\text{ALIVE}, -, k)$ message to p during $(t_s, t_s + \Delta']$. By assumption (4), $t_s \geq t_\Delta$. Thus, by Lemma 68 and the definitions of t_Δ and Δ, this $(\text{ALIVE}, -, k)$ message is delivered to p from s during time interval $(t_s, t_s + \Delta' + \Delta]$.

CLAIM 3: *p executes at least one complete iteration of its repeat forever loop during time interval* $[t_s + \Delta' + \Delta, t_e]$. To see this, recall that p executes *at least* $\zeta - 1$ complete iterations of its repeat forever loop during time interval $[t_s, t_e]$. Moreover, during time interval $[t_s, t_s + \Delta' + \Delta]$, p executes *at most* $\lceil (\Delta' + \Delta)/\epsilon \rceil = \zeta - 3$ complete iterations of its repeat forever loop (this follows from the definition of ϵ). This implies Claim 3.

Since an $(\text{ALIVE}, -, k)$ message is delivered to p from s during time interval $(t_s, t_s + \Delta' + \Delta]$, and p executes at least one complete iteration of its repeat forever loop during time interval $[t_s + \Delta' + \Delta, t_e]$, we conclude that p receives some $(\text{ALIVE}, -, -)$ message from s during interval $(t_s, t_e]$ —a contradiction to Claim 1. □

The above lemma considers ACCUSATION messages sent in line 31. A process that receives such messages may forward it in line 40. The next corollary says that if a timeout is proper then any ACCUSATION that it generates (whether in line 31 or 40) is outdated.

Corollary 77 *Suppose a process p times out on s (in line 30). If this timeout is proper, then every* $(\text{ACCUSATION-}s, -)$ *message that is sent to s (in line 31 or 40) as a consequence of this timeout is outdated.*

PROOF. By Lemma 76, if a process p times out on s (in line 30) and this timeout is proper, then every $(\text{ACCUSATION-}s, -)$ message that p sends to all other processes in line 31 as a consequence of this timeout is outdated. Let $(\text{ACCUSATION-}s, ph)$ be the first such message that p sends, and let t be the time at which it is sent. Since this message is outdated, then every $(\text{ACCUSATION-}s, ph)$ that is sent at time $t' \geq t$ is also outdated (this is because $phase_s[s]$ is monotonically non-decreasing). In particular, every $(\text{ACCUSATION-}s, ph)$ message that is sent by a process to s in line 40 (after receiving one of the $(\text{ACCUSATION-}s, ph)$ messages sent earlier by p in line 31) is also outdated. □

Lemma 78 *$counter_s[s]$ is bounded.*

PROOF. Note that s increases its $counter_s[s]$ only if it receives an $(\text{ACCUSATION-}s, -)$ message (lines 35-40). There are two kinds of such $(\text{ACCUSATION-}s, -)$ messages: (a) those that are sent to s as a consequence of a *proper* timeout on s, and (b) those that are sent to s as a consequence of an *improper* timeout on s. By Corollary 77, all the $(\text{ACCUSATION-}s, -)$ messages of kind (a) are outdated. As we previously observed, such messages do not affect $counter_s[s]$. Thus only those messages of kind (b) may cause s to increment $counter_s[s]$. By Lemma 74, the number of improper timeouts on s is finite. Since each timeout on s causes at most $n - 1$ $(\text{ACCUSATION-}s, -)$ message to be sent to s, the number of $(\text{ACCUSATION-}s, -)$ messages of kind (b) is finite. Therefore $counter_s[s]$ is bounded. □

Definition 79 *For every process p, let c_p be the largest value of $counter_p[p]$ in the run that we consider ($c_p = \infty$ if $counter_p[p]$ is unbounded). Let ℓ be the process such that $(c_\ell, \ell) = \min\{(c_p, p) : p$ is a correct process$\}$.*

By definition, ℓ is a correct process. Furthermore, by Lemma 78, *counter*$_s[s]$ is bounded, i.e., $c_s < \infty$. Thus, $c_\ell < \infty$, i.e., *counter*$_\ell[\ell]$ is bounded.

Lemma 80 *For every correct process p, if there is a time after which $\ell \in$ active$_p$, then there is a time after which leader$_p = \ell$.*

PROOF. This proof is identical to the proof of Lemma 50 (except that it uses Lemma 62 instead of Lemma 34), and hence we omit it here. □

Observation 81 *For every correct process p, there is a time after which $p \in$ active$_p$.*

PROOF. When p executes its initialization code, it sets *active*$_p$ to $\{p\}$. Thereafter, p never removes itself from *active*$_p$. □

Corollary 82 *There is a time after which leader$_\ell = \ell$.*

PROOF. By Observation 81, there is a time after which $\ell \in$ *active*$_\ell$. The result now follows from Lemma 80. □

Corollary 83 *There is a time after which phase$_\ell[\ell]$ stops changing.*

PROOF. Note that ℓ changes *phase*$_\ell[\ell]$ only when it considers that it lost the leadership (in lines 5-6), and each time this occurs ℓ sets *leader*$_\ell \neq \ell$ (in line 8). By Corollary 82, this can happen only a finite number of times. □

Definition 84 *Let ℓphase be the final value of phase$_\ell[\ell]$.*

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

Lemma 85 *For every correct process p, there is a time after which if leader$_p = \ell$ then phase$_p[\ell] \geq \ell$phase.*

PROOF. Let p be a correct process. If $p = \ell$ then the lemma follows by the definition of ℓ*phase*. Now suppose $p \neq \ell$. If there is a time after which *leader*$_p \neq \ell$ then the lemma follows vacuously. So, suppose that *leader*$_p = \ell$ infinitely often. Then, by the way *leader*$_p$ is computed, $\ell \in$ *active*$_p$ at the beginning of infinitely many iterations of the repeat forever loop. Note that initially $\ell \notin$ *active*$_p$ since $\ell \neq p$, and so ℓ is added to *active*$_p$ at least once, and this happens in line 20.

We claim that ℓ is added to *active*$_p$ in line 20 infinitely often. Indeed, suppose not and consider the last time when ℓ is added to *active*$_p$. When this happens, *timer*$_p[\ell]$ is set to *timeout*$_p[\ell]$. Subsequently, each loop iteration decrements *timer*$_p[\ell]$, until it finally reaches 0. Then, the next loop iteration removes ℓ from *active*$_p$ and thereafter ℓ is never again in *active*$_p$—a contradiction that shows the claim.

By the claim, p receives $(\text{ALIVE}, -, -)$ messages from ℓ infinitely often. Note that ℓ only sends finitely many $(\text{ALIVE}, -, x)$ messages with $x <$ ℓ*phase*. Therefore, there is a time after which the only

$(\text{ALIVE}, -, y)$ messages received from ℓ are those with $y \geq \ell phase$. When p receives one such message, p sets $phase_p[\ell]$ to $y \geq \ell phase$. Then, $phase_p[\ell] \geq \ell phase$ forever after, since $phase_p[\ell]$ is monotonically nondecreasing. □

Lemma 86 *A process p can send only finitely many $(\text{ACCUSATION-}\ell, x)$ messages with $x < \ell phase$.*

PROOF. Note that (1) ℓ only sends finitely many $(\text{ALIVE}, -, x)$ messages with $x < \ell phase$. We now claim that (2) only finitely many (CHECK, ℓ, x) are sent with $x < \ell phase$. Indeed, when some correct process q sends a (CHECK, ℓ, x) message, $leader_q = \ell$ and $phase_q[\ell] = x$. By Lemma 85, there is a time after which if $leader_q = \ell$ then $phase_q[\ell] \geq \ell phase$. Thus, there is a time after which any (CHECK, ℓ, x) message that q sends has $x \geq \ell phase$. This shows the claim.

Consider any process r. We now claim that r sends $(\text{ACCUSATION-}\ell, x)$ only finitely many times with $x < \ell phase$ in line 31. This claim immediately implies the lemma, because a process can relay an ACCUSATION message in line 40 only if another process previously sent this message in line 31. To show the claim, suppose that process r sends $(\text{ACCUSATION-}\ell, x)$ and $(\text{ACCUSATION-}\ell, x')$ in line 31 with $x, x' < \ell phase$ at two different times t_1 and t_2. Then, between times t_1 and t_2, r must set $timer_r[\ell]$ to some value different from -1. This can only happen in lines 23 and 29. Therefore, between t_1 and t_2, r must either receive $(\text{ALIVE}, -, x'')$ from ℓ or receive $(\text{CHECK}, \ell, x'')$ from some process with $x'' < \ell phase$. By (1) and (2), this can only happen finitely many times. This shows the claim. □

Lemma 87 *No process sends $(\text{ACCUSATION-}\ell, \ell phase)$ messages infinitely often in line 31.*

PROOF. Suppose, for contradiction, that some process p sends infinitely many $(\text{ACCUSATION-}\ell, \ell phase)$ messages in line 31. Note that $p \neq \ell$, because a process never sends ACCUSATION messages to itself. We claim that ℓ receives such messages infinitely often, which is a contradiction because (1) every time ℓ receives such a message, it increments $counter_\ell[\ell]$, and so (2) eventually $counter_\ell[\ell]$ becomes greater than c_ℓ.

To show the claim, first assume that $p \neq h$. Then p sends $(\text{ACCUSATION-}\ell, \ell phase)$ to h infinitely often. By Lemma 86, and the easy fact that no process sends $(\text{ACCUSATION-}\ell, y)$ with $y > \ell phase$, there is a time after which $(\text{ACCUSATION-}\ell, \ell phase)$ is the only ACCUSATION-ℓ message that p sends. This implies, by Lemma 63, that h receives $(\text{ACCUSATION-}\ell, \ell phase)$ from p infinitely often. If $h = \ell$ then the claim follows. Otherwise, every time h receives $(\text{ACCUSATION-}\ell, \ell phase)$ from p, it sends $(\text{ACCUSATION-}\ell, \ell phase)$ to ℓ. So h sends $(\text{ACCUSATION-}\ell, \ell phase)$ to ℓ infinitely often. By Lemma 86, there is a time after which these are the only ACCUSATION-ℓ messages that h sends. This implies, by Lemma 63, that ℓ receives $(\text{ACCUSATION-}\ell, \ell phase)$ from h infinitely often, which shows the claim.

The argument for the case $p = h$ is very similar. □

Lemma 88 *No process p adds and removes ℓ to and from its set $active_p$ infinitely often.*

PROOF. Suppose, for contradiction, that some process p adds and removes ℓ to and from $active_p$ infinitely often. This implies that (a) p receives $(\text{ALIVE}, -, -)$ messages from ℓ infinitely often, and (b) p sends $(\text{ACCUSATION-}\ell, -)$ messages infinitely often in line 31. From (a) and the definition of $\ell phase$, p eventually

receives an $(\text{ALIVE}, -, \ell phase)$ from ℓ. So, there is a time after which $phase_p[\ell] = \ell phase$. Thus, from (b), p sends infinitely many $(\text{ACCUSATION-}\ell, \ell phase)$ messages in line 31 —a contradiction to Lemma 87. □

Lemma 89 *There is a time after which $\ell \in active_h$ and $phase_h[\ell] = \ell phase$.*

PROOF. If $h = \ell$ the result follows by the definition of $\ell phase$ and the fact that $\ell \in active_\ell$. Now assume $h \neq \ell$. By Corollary 82 and the definition of $\ell phase$, ℓ sends an infinite number of $(\text{ALIVE}, -, \ell phase)$ messages to all processes except itself. Moreover, ℓ only sends a finite number of $(\text{ALIVE}, -, y)$ with $y \neq \ell phase$. Since $h \neq \ell$, this implies by Lemma 63 that h receives an infinite number of these $(\text{ALIVE}, -, \ell phase)$ messages from ℓ. Therefore, there is a time after which h has $phase_h[\ell] = \ell phase$. Moreover, h adds ℓ to $active_h$ infinitely often. From Lemma 88, h removes ℓ from $active_h$ only finitely often, and so the lemma follows. □

By Lemmas 80 and 89, we have

Lemma 90 *There is a time after which $leader_h = \ell$.*

Lemma 91 *There is a time after which only ℓ sends* ALIVE *messages.*

PROOF. Consider any correct process $p \neq \ell$. From Lemma 88, there are two possible cases:

1. There is a time after which $\ell \in active_p$. In this case, by Lemma 80, there is a time after which $leader_p = \ell$. After this time, p does not send ALIVE messages.

2. There is a time after which $\ell \notin active_p$. This implies that (a) there is a time after which p does not receive any ALIVE message from ℓ and (b) $p \neq h$ (by Lemma 89), and (c) $h \neq \ell$ (because if $h = \ell$ then, by Corollary 82, h sends an infinite number ALIVE messages to p, and so by Lemma 63, p receives an infinite number of ALIVE messages from h, which contradicts (a)). Now, suppose, for contradiction, that p sends ALIVE messages infinitely often. By Lemma 63, h receives ALIVE messages from p infinitely often. By Lemmas 89 and 90, there is a time after which $leader_h = \ell$ and $phase_h[\ell] = \ell phase$. After that time, each time h receives an ALIVE message from p, h sends a $(\text{CHECK}, \ell, \ell phase)$ message to p (since $p \neq \ell$ and $h \neq \ell$). Thus, h sends infinitely many $(\text{CHECK}, \ell, \ell phase)$ messages to p, and there is a time after which $(\text{CHECK}, \ell, \ell phase)$ are the only $(\text{CHECK}, -, -)$ messages that h sends to p. By Lemma 63, this implies that p receives $(\text{CHECK}, \ell, \ell phase)$ from h infinitely often. Therefore, we have the following:

   (i) There is a time after which p has $phase_p[\ell] = \ell phase$,

   (ii) p starts $timer_p[\ell]$ and times out on ℓ infinitely often (because of (a)), and

   (iii) p sends infinitely many $(\text{ACCUSATION-}\ell, \ell phase)$ messages to ℓ in line 31 —a contradiction to Lemma 87.

Thus, in both cases (1) and (2) there is a time after which p does not send ALIVE messages. □

Lemma 92 *For every correct process p, there is a time after which $leader_p = \ell$.*

PROOF. Let p be any correct process. From Lemma 88, there are two possible cases:

1. There is a time after which $\ell \in$ *active*$_p$. In this case, by Lemma 80, there is a time after which *leader*$_p = \ell$.

2. There is a time after which $\ell \notin$ *active*$_p$. Since a process $q \neq p$ can remain in *active*$_p$ only if p keeps receiving ALIVE messages from q, then, by Lemma 91 and the fact that $p \in$ *active*$_p$ (always), there is a time after which *active*$_p = \{p\}$. So there is a time after which $leader_p = p$. From this time on, p repeatedly sends ALIVE forever —a contradiction to Lemma 91.

Thus, only case (1) holds. □

Lemma 93 *There is a time after which only ℓ sends messages.*

PROOF. There are $n + 2$ types of messages: ALIVE, CHECK, and ACCUSATION-q, for each process q.

1. By Lemma 91, *there is a time after which only ℓ sends* ALIVE *messages.*

2. *Only a finite number of* CHECK *messages are sent.* To see this, note that a process p sends a CHECK message to another process q only if p receives an ALIVE message from q at a time when $leader_p \neq q$. By Lemmas 91 and 92, there is a time after which this cannot occur.

3. *For any process q, only a finite number of* ACCUSATION-q *messages are sent.* To show this, let q be a process. It is sufficient to prove that each process p sends a finite number of ACCUSATION-q messages in line 31 (this is because p *relays* an ACCUSATION-q message in line 40 only if another process previously sent this message in line 31 of its code).

   When a process p sends an $(\text{ACCUSATION-}q, -)$ message in line 31, p "turns off" *timer*$_p[q]$ by setting it to -1 in line 34. After this occurs, p can send another $(\text{ACCUSATION-}q, -)$ message in line 31 only if p "turns on" *timer*$_p[q]$ again in line 23 or line 29, and this can happen only if (a) p receives an ALIVE message from q (in line 19), or (b) p receives a $(\text{CHECK}, -)$ message (in line 26). Thus, p can send an infinite number of $(\text{ACCUSATION-}q, -)$ messages in line 31 only if (a) p receives an infinite number of ALIVE messages from q or (b) p receives an infinite number of $(\text{CHECK}, -)$ messages. From (1) and (2) above, we deduce that p can send an infinite number of $(\text{ACCUSATION-}q, -)$ messages in line 31 only for $q = \ell$. But p sends only a finite number of $(\text{ACCUSATION-}\ell, -)$ in line 31, because each time p sends such a message, p removes ℓ from *active*$_p$ (in line 32), and from Lemma 92, there is a time after which $\ell \in$ *active*$_p$. Thus each process p sends only a finite number of $(\text{ACCUSATION-}q, -)$ messages in line 31 for every process q.

□

From Lemmas 92 and 93, we get the following result:

Theorem 94 *The algorithm in Figure 6 implements Ω in system S^+, and it is communication-efficient.*

7 Final remarks

In their 2002 PODC tutorial [KR02], 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, in contrast to Ω, $\diamond\mathcal{P}$ is *not* implementable in system S. In fact, it is easy to 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 eventually timely source(s) in S. So S is an example of a partially synchronous 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 [ADGFT01].

In [KR02], Keidar and Rajsbaum also ask: "When 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^+).

Finally, it is also worth pointing out that the above results provide an alternative proof that $\diamond\mathcal{P}$ is *strictly* stronger than $\diamond\mathcal{S}$: 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.

Acknowledgement

The authors would like to thank the anonymous referees for many helpful comments.

References

[ADGFT] Marcos K. Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. Type fairness and a comparison with other link fairness properties. In preparation.

[ADGFT01] Marcos K. Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. Stable leader election (extended abstract). In *Proceedings of the 15th International Symposium on Distributed Computing*, LNCS 2180, pages 108–122. Springer-Verlag, October 2001.

[ADGFT03] Marcos K. Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. On implementing Omega with weak reliability and synchrony assumptions. In *Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing*, pages 306–314, July 2003.

[ADGFT04] Marcos K. Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. Communication-efficient leader election and consensus with limited link synchrony. In *Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing*, pages 328–337, July 2004.

[BMS02] Marin Bertier, Olivier Marin, and Pierre Sens. Implementation and performance evaluation of an adaptable failure detector. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 354–363, June 2002.

[CGR07] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective (invited talk). In *Proceedings of the 26th ACM Symposium on Principles of Distributed Computing*, pages 398–407, August 2007.

[CHT96] Tushar D. Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.

[Chu98] Francis Chu. Reducing Ω to $\diamond W$. *Information Processing Letters*, 67(6):298–293, September 1998.

[CL02] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, November 2002.

[CT96] Tushar D. Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[CTA02] Wei Chen, Sam Toueg, and Marcos K. Aguilera. On the quality of service of failure detectors. *IEEE Transactions on computers*, 51(5):561–580, May 2002.

[DDS87] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.

[DG02] Partha Dutta and Rachid Guerraoui. Fast indulgent consensus with zero degradation. In *Proceedings of the 4th European Dependable Computing Conference*, LNCS 2485, pages 191–208. Springer-Verlag, October 2002.

[DGFG03] Carole Delporte-Gallet, Hugues Fauconnier, and Rachid Guerraoui. Shared memory vs. message passing. Research Report IC/2003/77, EPFL, December 2003.

[DGFG+04] Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, Vassos Hadzilacos, Petr Kouznetsov, and Sam Toueg. The weakest failure detectors to solve certain fundamental problems in distributed computing. In *Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing*, pages 338–346, July 2004.

[DLS88] Cynthia Dwork, Nancy A. Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

[DT00] Boris Deianov and Sam Toueg. Failure detector service for dependable computing. In *Proceedings of the 2000 International Conference on Dependable Systems and Networks*, pages B14–B15, June 2000.

[EHT07] Jonathan Eisler, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector to solve nonuniform consensus. *Distributed Computing*, 19(4):335–359, March 2007.

[FR06] Antonio Fernandez and Michel Raynal. From an intermittent rotating star to a leader. Technical Report 1810, IRISA, Université de Rennes, France, August 2006.

[FRT01] Christof Fetzer, Michel Raynal, and Frederic Tronel. An adaptive failure detection protocol. In *Proceedings of the 2001 Pacific Rim International Symposium on Dependable Computing*, pages 146–153, December 2001.

[GL03] Eli Gafni and Leslie Lamport. Disk Paxos. *Distributed Computing*, 16(1):1–20, February 2003.

[HMSZ05] Martin Hutle, Dahlia Malkhi, Ulrich Schmid, and Lidong Zhou. Chasing the weakest system model for implementing Omega and consensus. Research Report 74/2005, Technische Universität Wien, Institut für Technische Informatik, 2005.

[KR02] Idit Keidar and Sergio Rajsbaum. On the cost of fault-tolerant consensus when there are no faults—a tutorial. Slides of tutorial presentation in the *21th ACM Symposium on Principles of Distributed Computing*, July 2002.

[LAF99] Mikel Larrea, Sergio Arevalo, and Antonio Fernandez. 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. Springer-Verlag, September 1999.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[Lam01] Leslie Lamport. Paxos made simple. *SIGACT News*, 32(4):18–25, December 2001.

[LFA00] Mikel Larrea, Antonio Fernandez, and Sergio Arevalo. Optimal implementation of the weakest failure detector for solving consensus. In *Proceedings of the 19th Symposium on Reliable Distributed Systems*, pages 52–59, October 2000.

[LFA01] Mikel Larrea, Antonio Fernandez, and Sergio Arevalo. Eventually consistent failure detectors. In *Proceedings of the 13th ACM Symposium on Parallel Algorithms and Architectures*, pages 326–327, July 2001.

[MMR03] Achour Mostefaoui, Eric Mourgaya, and Michel Raynal. Asynchronous implementation of failure detectors. In *Proceedings of the 2003 International Conference on Dependable Systems and Networks*, pages 351–360, June 2003.

[MOZ05] Dahlia Malkhi, Florian Oprea, and Lidong Zhou. Omega meets Paxos: leader election and stability without eventual timely links. In *Proceedings of the 19th International Conference on Distributed Computing*, LNCS 3724, pages 199–213. Springer-Verlag, September 2005.

[MR01] Achour Mostefaoui and Michel Raynal. Leader-based consensus. *Parallel Processing Letters*, 11(1):95–107, March 2001.

[MRT06] Achour Mostefaoui, Michel Raynal, and Corentin Travers. Time-free and timer-based assumptions can be combined to obtain eventual leadership. *IEEE Transactions on Parallel and Distributed Systems*, 17(7):656–666, July 2006.

[PLL00] Roberto De Prisco, Butler Lampson, and Nancy A. Lynch. Revisiting the Paxos algorithm. *Theoretical Computer Science*, 243:35–91, July 2000.

[ST07] Nicolas Schiper and Sam Toueg. A stable, robust, and lightweight Leader Election Service for dynamic systems. Research Report 2007/05, University of Lugano, May 2007.

[vRMH98] Robbert van Renesse, Yaron Minsky, and Mark Hayden. A gossip-style failure detection service. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 55–70, September 1998.