Distrib. Comput. (2010) 22:303–334
DOI 10.1007/s00446-010-0106-4

Adaptive progress: a gracefully-degrading liveness property

Marcos K. Aguilera · Sam Toueg

Received: 13 October 2008 / Accepted: 9 May 2010 / Published online: 25 June 2010

Abstract We introduce a simple liveness property for shared object implementations that is gracefully degrading depending on the degree of synchrony in each run. This property, called *adaptive progress*, provides a gradual bridge between obstruction-freedom and wait-freedom in partially-synchronous systems. We show that adaptive progress can be achieved using very weak shared objects. More precisely, every object has an implementation that ensures adaptive progress and uses only abortable registers (which are weaker than safe registers). As part of this work, we present a new leader election abstraction that processes can use to dynamically compete for leadership such that if there is at least one timely process among the current candidates for leadership, then a timely leader is eventually elected among the candidates. We also show that this abstraction can be implemented using abortable registers.

1 Introduction

1.1 A new progress condition

Three liveness properties have been extensively studied in the context of shared object implementations, namely, in order of increasing strength, *obstruction-freedom*, *lock-freedom*,[1] and *wait-freedom* [10,11].

In this paper, we first propose a new liveness property, called *adaptive progress* (or briefly *AP*), that provides a natural bridge between the above well-known progress properties in partially-synchronous systems.[2] The strength of the liveness guarantee provided by adaptive progress depends on the degree of synchrony, that is the number of partially-synchronous processes, in each run. As the degree of synchrony "increases", the liveness guarantee gets stronger: roughly speaking, it goes from obstruction-freedom to lock-freedom, and then continues *gradually* all the way to wait-freedom. In other words, the new liveness property *adapts its progress guarantees* to the degree of synchrony in a run, thereby providing graceful degradation. This feature is attractive for the following reason.

Many systems are synchronous most of the time. During those times, it is natural to require strong liveness guarantees, but when synchrony degrades we may be willing to gradually sacrifice some liveness. Ideally, this sacrifice should be "fair", namely, processes that fail to meet some minimal synchrony condition may fail to make progress, but not others. With adaptive progress, processes that are *timely*, namely, processes that satisfy some reasonable synchrony condition, are guaranteed to make progress. Processes that are not timely may fail to make progress, but even if they are unboundedly slow or unstable (e.g., they repeatedly oscillate between being timely and very slow) they cannot prevent the progress of timely processes. We now explain the adaptive progress property in more detail.

Research supported in part by the National Science and Engineering Research Council of Canada.

M. K. Aguilera (✉)
Microsoft Research Silicon Valley,
1065 La Avenida-bldg SVC6,
Mountain View, CA 94043, USA

S. Toueg
University of Toronto, 10 King's College Road,
Toronto, ON M5S 3G4, Canada

[1] An implementation that is lock-free is also called "non-blocking".

[2] Adaptive progress was called *timeliness-based wait-freedom* in an earlier version of this work [3].

Intuitively, adaptive progress requires that every process p that is timely in a run R be *wait-free in* R, i.e., p completes each operation that it executes in R in a finite number of steps. Timeliness is defined here as *relative* to the speed of the processes in the system, as in the seminal work on partial synchrony of [6]. More precisely, a correct process p is *timely in a run* R if there is an integer $i \geq 1$ (which is unknown and may depend on R) such that for every i consecutive process steps in R, there is at least one step of p.

We now relate adaptive progress to obstruction-freedom, lock-freedom and wait-freedom.

We first note that any implementation that satisfies the adaptive progress property (i.e., an *AP implementation*) is necessarily obstruction-free. To see this, consider an AP implementation of some arbitrary object, and suppose that there is a time after which some process p runs solo in a run R of this implementation. Obstruction-freedom requires that p completes every operation that it executes in R. Note that, by definition, p is timely in run R (even if p is extremely slow with respect to real time!). This is because: (a) "timely" is defined relative to the speed of the system's processes in R, and (b) there is a time after which p is the only process taking steps in R. (Intuitively, when p runs solo it is not slow relative to other processes, so p is timely.) Since p is timely in R, adaptive progress requires p to be wait-free in R, i.e., p must complete every operation that it executes in R—exactly as required by obstruction-freedom. Thus, adaptive progress implies obstruction-freedom.

Now consider an AP implementation of an arbitrary object O in a system with n processes. As we observed above, this implementation is obstruction-free. Consider a run R of this implementation such that every process has an infinite sequence of operations that it wishes to apply on O (so all processes continuously compete to access O). Since no process runs solo in R, an obstruction-free implementation of O does *not* guarantee any progress for any process. If there is some synchrony in R, however, then the AP implementation of O still guarantees some progress, and the amount of progress depends on the degree of synchrony in R. If some process p is timely in R, then the adaptive progress property guarantees that some process (namely p) completes all its operations in R. So if a process is timely in R then, in some precise sense, the AP implementation of O is "lock-free in R". More generally, if k processes are timely in R, these k processes are guaranteed to complete all their operations in R. In the limit, if all the processes in R are timely, then all processes complete all their operations in R, so the AP implementation of O is "wait-free in R". Thus, as the number of timely processes increases from 1 to n, the progress guarantee of an AP implementation goes from lock-freedom incrementally all the way to wait-freedom.

1.2 Achieving adaptive progress

We next consider the problem of implementing objects that satisfy the adaptive progress property. It is well-known that any object has a wait-free implementation (and *a fortiori* an AP implementation), provided one is allowed to use some strong synchronization objects like *compare-and-swap* [10]. But such objects can be slow in practice compared to weaker ones such as *registers*.

A natural question is therefore: what is the "weakest" object that one can use to achieve AP implementations? We show here the surprising result that such implementations can be achieved using objects that are strictly weaker than *safe registers*. More precisely, we give a universal AP implementation that uses only *abortable registers* [2]. Roughly speaking, an abortable register behaves like an atomic register except that, when it is accessed concurrently, some of the concurrent read or write operations may *abort* (by returning the special value $\bot$). A write operation that aborts may or may not take effect and, since the writer gets back $\bot$ in either case, it does not know whether its write operation succeeded or not.[3]

To get AP implementations using abortable registers, we proceed as follows:

1. We first introduce a dynamic leader election abstraction, denoted Ω_Δ, that processes can use to dynamically compete for leadership such that *if there is at least one timely process among the current candidates for leadership*, then a timely leader is eventually elected among the leader candidates.
2. We then describe how to implement Ω_Δ in a system with registers. We give two such implementations: The first one, which is relatively simple and efficient, uses *atomic registers*; the second one, which is significantly more complex, uses *abortable registers* only.
3. Finally, we show how Ω_Δ can be used to obtain an AP implementation of an object O of *any* type T using abortable registers. This is done in two steps:
    (a) Given any type T, we first use the universal construction described in [2] to get a *wait-free* implementation of an object O_{QA} of type T_{QA}—the *query-abortable* counterpart of T.[4]

[3] In contrast, a write operation on a *safe* register always succeeds, i.e., it always takes effect, even if it is concurrent with other read or write operations.

[4] Intuitively, an object of type T_{QA} behaves like one of type T except that: (i) concurrent operations may abort; an operation that aborts returns $\bot$ and it may or may not take effect; (ii) there is an additional operation, denoted QUERY, that any process can use to determine whether the last (non-QUERY) operation that it applied on the object took effect, and if it did, the corresponding reply; the QUERY operation may itself abort.

This construction can be done using abortable registers only.

(b) We then use Ω_Δ to transform the wait-free implementation of O_{QA} of type T_{QA} into an AP implementation of an object O of type T. Roughly speaking, *timely* processes use Ω_Δ to successively access O_{QA} *in a fair way* among themselves. This transformation does not use any shared objects.

The above approach to obtain AP implementations is similar to the "boosting" of obstruction-free implementations into wait-free implementations using synchrony [7,15] or failure detectors (which in turn can be implemented using synchrony) [9]. In contrast to AP implementations, however, the wait-free implementations obtained by boosting in [7,9,15] are not gracefully degrading: the boosting algorithms assume that all the correct processes are (eventually) timely,[5] and it is not difficult to construct runs where a partial loss of synchrony causes a total loss of liveness. In other words, if some processes are not timely, they can prevent the progress of all the correct processes, even the timely ones. It is also worth noting that these boosting algorithms use objects that are stronger than abortable registers: the algorithms in [7,9] and [15] use atomic registers and compare-and-swap, respectively. A more detailed discussion of these algorithms and other related work is given in Sect. 7.

As a final remark, the implementation of Ω_Δ using abortable registers (given in Sect. 5) implies that one can implement Ω—a failure detector which is sufficient to solve consensus [4]—in a system with abortable registers and only one timely process. Thus, in shared memory systems with limited synchrony, some powerful failure detectors can be implemented with objects that are weaker than safe registers.

1.3 Dynamic activity monitors

In this paper, we also introduce a new abstraction, called a *(dynamic) activity monitor*, that serves as a building block for dynamic applications in shared memory systems. Intuitively, for every ordered pair of processes p and q, the activity monitor denoted $\mathcal{A}(p, q)$ is an abstraction that helps p determine whether q is currently *active* or *inactive*, and whether q is timely (with respect to p). This activity monitor is fully dynamic: both p and q can independently turn the monitoring mechanism on or off at any time they want. We use activity monitors to implement Ω_Δ in Sect. 4.2 in a modular way that shields the implementation from low-level synchrony mechanisms, such as timers and timeouts.

[5] It is easy to see that the concepts of "timely" and "eventually timely", which is seemingly weaker, are actually the same when the timeliness bounds are not known and depend on each run (as assumed in [7,9,15] and here).

Summary of contributions

- We introduce a new liveness property, called adaptive progress, for shared object implementations. This liveness property is simple and fair: every process that is timely is guaranteed to be wait-free. It is also gracefully degrading: when synchrony increases, the liveness guarantee also increases gradually from obstruction-freedom (when there are no synchrony assumptions) all the way to wait-freedom (when all processes are timely).
- We give two universal constructions that satisfy the adaptive progress property: a simple one that uses plain (atomic) registers, and a more complex one that uses only abortable registers. The second construction implies that adaptive progress can be achieved with registers that are weaker than safe.
- We specify a new leader election abstraction, denoted Ω_Δ, that allows processes to dynamically compete for leadership. In contrast to previously defined dynamic leader election abstractions, the specification of Ω_Δ refers to the synchrony of the processes that participate in the election: roughly speaking, if there is at least one timely process among the processes that currently wish to be elected, then a timely process is eventually elected.
- We show how to implement Ω_Δ in systems with registers, and also in systems with abortable registers. This shows that it is possible to implement the powerful failure detector Ω using only abortable registers, provided at least one process in the system is timely.
- We introduce the concept of a dynamic activity monitor, denoted $\mathcal{A}(p, q)$, that can help a process p determine the current "status" of another process q. With $\mathcal{A}(p, q)$, each of p and q can independently stop or resume its participation in this monitoring whenever it wants. We believe that both Ω_Δ and $\mathcal{A}(p, q)$ are useful building blocks for dynamic applications in shared memory systems.

Road map

The paper is organized as follows. In Sect. 2, we explain our shared-memory model and define the adaptive progress property. We define the dynamic leader elector Ω_Δ in Sect. 3. In Sect. 4, we implement Ω_Δ using registers, in two steps. First, we define activity monitors and implement them using registers in Sect. 4.1. Then, we use activity monitors and registers to implement Ω_Δ in Sect. 4.2. In Sect. 5, we implement Ω_Δ using abortable registers. In Sect. 6, we show how to use Ω_Δ to achieve an AP implementation of an arbitrary type. We conclude the paper with a discussion of related work in Sect. 7.

2 Model

We consider shared-memory systems with $n \geq 2$ processes $\Pi = \{0, \ldots, n-1\}$ that can communicate with each other via shared registers. We consider two types of shared registers, atomic registers [13,14] and abortable registers [2].[6] In our model, time values are taken from the set $\mathbb{N}$ of positive integers.

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 invokes an operation on a shared register and changes state, (2) p receives a response from an operation and changes state, or (3) p just changes state. If p invokes an operation in a step, p's next step is to receive a response from that operation (and change state). For convenience, we assume that each step occurs instantaneously and there is at most one step per time unit.

A process may fail by crashing, in which case the process's state changes to a crash state and the process stops taking steps forever. A process p is *correct* if p does not crash. A correct process takes infinitely many steps (a process can take "do-nothing" steps if it has nothing to do). We now define what it means for a process p to be timely with respect to another process q in a run:

Definition 1 We say that p is q-*timely* (in a run) if p is correct and there is an integer $i \geq 1$ such that every time interval containing i steps of q has at least one step of p (in this run).

Note that the timeliness bound i above is not known (it depends on each run and on each pair of processes p and q).

Definition 2 We say that p *is timely* (in a run) if p is q-timely for every process $q \in \Pi$ (in this run).

We consider the following liveness property for object implementations in shared memory systems:

Definition 3 (*Adaptive progress*) An object implementation satisfies the adaptive progress property, if, for every run R of the implementation, every process that is timely in R completes its operations on the object in a finite number of its own steps.

Throughout the paper, if C is some property, we say that *there is a time after which C holds* if there is a time t such that for every time $t' \geq t$, property C holds at time t'. Similarly, we say that C *holds infinitely often* if for every time t, there is a time $t' > t$ such that C holds at time t'. Finally, we say that a variable v *increases without bound* if for every $k \in \mathbb{N}$ there is a time after which $v > k$.

Properties of timely and non-timely processes

We now state and prove some basic properties of timely and non-timely processes. The following lemmas are with respect to an arbitrary run R.

Observation 1 (a) If p is correct then p is p-timely. (b) If p is correct and q crashes then p is q-timely.

Proof Trivial from the definitions.

Lemma 1 *If a process p is timely then there is an integer $i \geq 1$ such that every time interval containing i process steps has at least one step of p.*

Proof Suppose that p is a timely process. Since p is timely, for every process q, p is q-timely, and so there is an integer $i_{pq} \geq 1$ such that every time interval containing i_{pq} steps of q has at least one step of p. Let $i = 1 + \sum_{q \in \Pi}(i_{pq} - 1)$. Note that $i \geq 1$. Moreover, every time interval containing i process steps must have at least i_{pq} steps of q for some process q. Such a time interval has at least one step of p.

Lemma 2 *If p is a correct process then p is timely if and only if there is an integer $i \geq 1$ such that every time interval containing i process steps has at least one step of p.*

Proof Let p be a correct process. If p is timely then, by Lemma 1, there is an integer $i \geq 1$ such that every time interval containing i process steps has at least one step of p.

If p is not timely then there is a process q such that p is not q-timely. Thus, since p is correct, for every integer $i \geq 1$, there is a time interval containing i process steps (those of q) but no steps of p.

Lemma 3 *For all processes p, q, and r, if p is q-timely and q is r-timely then p is r-timely.*

Proof Let p, q, and r be processes such that p is q-timely and q is r-timely. So p and q are correct, and there are integers $i_{pq} \geq 1$ and $i_{qr} \geq 1$ such that (*) every time interval containing i_{pq} steps of q has at least one step of p and (**) every time interval containing i_{qr} steps of r has at least one step of q.

If r crashes, then p is r-timely by Observation 1(b). Now assume that r is correct. Let $i_{pr} = i_{pq}(i_{qr} - 1) + 1$. Note that $i_{pr} \geq 1$. Consider any time interval containing i_{pr} steps of r. By (**), such a time interval has at least i_{pq} steps of q. By (*), this time interval has at least one step of p. Thus, since p is correct, p is r-timely.

Corollary 1 *For all processes p and q, if p is q-timely and q is timely then p is timely.*

[6] With both types of registers, read and write operations are not instantaneous, each such operation spans an interval of time; but their behavior is *linearizable* [12].

Proof Let p and q be processes such that p is q-timely and q is timely. By definition, q is r-timely for every process r. By Lemma 3, p is r-timely for every process r. Thus, p is timely.

Corollary 2 *For all processes p and q, if p is not timely and q is timely then p is not q-timely.*

3 The dynamic leader elector Ω_Δ

Intuitively, Ω_Δ is a dynamic leader election abstraction that allows processes to dynamically compete for leadership such that if there is at least one timely process among the candidates for leadership, then a timely leader is eventually elected.

Each process p interacts with Ω_Δ via *input* and *output* variables, denoted CANDIDATE_p and LEADER_p, respectively; these variables are local to p. Process p uses the input variable CANDIDATE_p to tell Ω_Δ whether it currently wants to compete for leadership: if p wants to do so it writes *true* to CANDIDATE_p, otherwise it writes *false* to CANDIDATE_p.

At each process p, Ω_Δ writes the output variable LEADER_p to tell p who the current leader is. More precisely, Ω_Δ sets LEADER_p to q if it thinks that q is the current leader, and Ω_Δ sets LEADER_p to the special value "?" when it does not give p any information about who may be the current leader (this can occur when Ω_Δ is still in the process of computing a leader or when p is not competing for leadership).

Note that some processes may repeatedly switch between competing and not competing for leadership, forever. Others may crash, or fail to be timely. Processes that are not timely may "flicker" forever: their execution speed may fluctuate so that sometimes they appear to be crashed or very slow, and sometimes they appear to be alive and timely. Ω_Δ ensures that if there are some timely processes that "permanently" compete for leadership, then a timely leader is eventually elected. This is guaranteed even if several processes that compete for leadership flicker forever.

To define Ω_Δ precisely, we first partition the set of correct processes according to how frequently they compete for leadership, as follows:

Definition 4 For each run R of Ω_Δ, we partition the set of processes that are correct in R as follows:

- *Ncandidates* $= \{q : q$ is correct and there is a time after which $\text{CANDIDATE}_q = \mathit{false}\}$.
- *Pcandidates* $= \{q : q$ is correct and there is a time after which $\text{CANDIDATE}_q = \mathit{true}\}$.
- *Rcandidates* $= \{q : q$ is correct and $\text{CANDIDATE}_q = \mathit{true}$ infinitely often and $\text{CANDIDATE}_q = \mathit{false}$ infinitely often$\}$.

Intuitively, the letters N, P, and R in the above definitions stand for Not candidate, Permanent candidate, and Repeated candidate, respectively.

Ω_Δ is defined as follows:

Definition 5 In every run R of Ω_Δ, the following properties hold:

1. If there is a timely process in *Pcandidates* then there is a timely process ℓ in *Pcandidates* or in *Rcandidates* such that
   (a) There is a time after which $\text{LEADER}_\ell = \ell$.
   (b) For every process $p \in$ *Pcandidates*, there is a time after which $\text{LEADER}_p = \ell$.
   (c) For every process $p \in$ *Rcandidates*, there is a time after which $\text{LEADER}_p \in \{?, \ell\}$.
2. For every process $p \in$ *Ncandidates*, there is a time after which $\text{LEADER}_p = ?$

Achieving stronger leader election properties: canonical use of Ω_Δ

Note that with the above specification of Ω_Δ, the elected leader ℓ can be in *Rcandidates*. In other words, Ω_Δ may elect as the permanent leader a process ℓ that repeatedly joins and then leaves the competition for leadership, forever. Since a process that leaves the competition for leadership is usually not interested (or willing) to be the leader, this "feature" of Ω_Δ can be undesirable. We can make this problem disappear if Ω_Δ is used in a particular way, which we call "canonical".

Suppose that a process p with $\text{CANDIDATE}_p = \mathit{false}$ wishes to set CANDIDATE_p to *true* (to compete for leadership). The use of Ω_Δ is canonical if p first waits until $\text{LEADER}_p \neq p$ before it sets CANDIDATE_p to *true*. Intuitively, if p stops being a candidate, p must wait until it stops being the leader (if it was the leader) before p is allowed to become a candidate again. This prevents a process in *Rcandidates* from being the leader forever.

More precisely, we define canonical use as follows:

Definition 6 The use of Ω_Δ is *canonical* (in a run R) if, for every correct process p, after p sets CANDIDATE_p to *false*, p waits until $\text{LEADER}_p \neq p$ before p sets CANDIDATE_p to *true*.

We first show that using Ω_Δ in the canonical way is not harmful, i.e., p's waiting for $\text{LEADER}_p \neq p$ does not cause p to block.

Lemma 4 *If a correct process p waits for $\text{LEADER}_p \neq p$ when $\text{CANDIDATE}_p = \mathit{false}$ then p does not wait forever.*

Proof Let p be a correct process and suppose, by contradiction, that p waits forever for $\text{LEADER}_p \neq p$ when $\text{CANDIDATE}_p = \mathit{false}$. Then there is a time after which

CANDIDATE$_p$ $= \mathit{false}$, and so $p \in \mathit{Ncandidates}$. By Property (2) of Ω_Δ, there is a time after which LEADER$_p = ?$, and so p does not wait forever for LEADER$_p \neq p$, a contradiction.

We now state and prove the main property obtained when Ω_Δ is used in the canonical way, namely, the leader ℓ elected by Ω_Δ is a timely process in *Pcandidates*, that is, a timely process that competes for leadership "forever":

Theorem 2 *With a canonical use of Ω_Δ, the following properties hold (in every run R)*:

1. *If there is a timely process in Pcandidates then there is a timely process ℓ in Pcandidates such that*
   (a) *There is a time after which* LEADER$_\ell = \ell$.
   (b) *For every process $p \in$ Pcandidates, there is a time after which* LEADER$_p = \ell$.
   (c) *For every process $p \in$ Rcandidates, there is a time after which* LEADER$_p \in \{?, \ell\}$.
2. *For every process $p \in$ Ncandidates, there is a time after which* LEADER$_p = ?$

Proof We first note that, by definition, Ω_Δ ensures Property (2). To show Property (1) above, assume that there is a timely process in *Pcandidates*. By the definition of Ω_Δ, there is a timely process $\ell \in \mathit{Pcandidates} \cup \mathit{Rcandidates}$, that satisfies Properties (a), (b), (c). It suffices to show that, when Ω_Δ is used in a canonical way, $\ell \notin \mathit{Rcandidates}$.

Suppose, by contradiction, that $\ell \in \mathit{Rcandidates}$. By definition, ℓ sets CANDIDATE to *true* and CANDIDATE to *false* infinitely often. With a canonical use of Ω_Δ, after ℓ changes the value of CANDIDATE to *false*, ℓ waits until LEADER$_\ell \neq \ell$, and only after this wait is over ℓ can change CANDIDATE from *false* to *true*. Thus, LEADER$_\ell \neq \ell$ infinitely often. This contradicts Property (a). So $\ell \notin \mathit{Rcandidates}$.

It is sometimes sufficient to have a leader election abstraction that provides the following simple property: (a) the process elected as the leader knows that it is the leader, and (b) the other processes know that they are not the leader. The following corollary to Theorem 2 states that Ω_Δ provides this simple property.

Corollary 3 *With a canonical use of Ω_Δ, the following properties hold (in every run R):*

If there is a timely process in Pcandidates then there is a timely process ℓ in Pcandidates such that

(a) *There is a time after which* LEADER$_\ell = \ell$.
(b) *For every correct process $p \neq \ell$, there is a time after which* LEADER$_p \neq p$.

4 Implementing Ω_Δ using registers

In this section, we show that Ω_Δ can be implemented using (atomic) registers. To do so, we first define activity monitors and explain how to implement them using registers (Sect. 4.1). We then use activity monitors and registers to implement Ω_Δ (Sect. 4.2).

4.1 Definition and implementation of activity monitors

For any two processes p and q, a *(dynamic) activity monitor* $\mathcal{A}(p, q)$ is an abstraction that can be used by p to determine whether q is currently *active* or *inactive*, and whether q is timely with respect to p (i.e., whether q is p-timely). This activity monitor is fully dynamic: both p and q can independently turn the monitoring mechanism on or off at any time they want, say for efficiency reasons.

Process p tells $\mathcal{A}(p, q)$ to turn the monitoring of q on or off by writing *on* or *off* to a variable MONITORING$_p[q]$ (which is local to p and is periodically read by $\mathcal{A}(p, q)$).

Similarly, q tells $\mathcal{A}(p, q)$ whether q is active for p or not by writing *on* or *off* to a variable ACTIVE-FOR$_q[p]$ (which is local to q and is periodically read by $\mathcal{A}(p, q)$). If q is alive and ACTIVE-FOR$_q[p] = \mathit{on}$ at time t, we say that q is *active for p at time t*. Otherwise, we say that q is *inactive for p at time t*.

The activity monitor $\mathcal{A}(p, q)$ tells p two things: (a) what it thinks the current status of q is, and (b) how many times it has so far suspected that q is not p-timely. To do so, $\mathcal{A}(p, q)$ writes two output variables, denoted STATUS$_p[q]$ and FAULTCNTR$_p[q]$, which are local to process p.

Intuitively, STATUS$_p[q] = \mathit{active}$, *inactive* or ?, if $\mathcal{A}(p, q)$ estimates that q is currently active for p, inactive for p, or $\mathcal{A}(p, q)$ has no estimate on the status of q, respectively; and FAULTCNTR$_p[q]$ is the number of times $\mathcal{A}(p, q)$ has suspected that q is not p-timely. Figure 1 summarizes the meaning of the input and output variables of $\mathcal{A}(p, q)$.

Note that there are nine possibilities for the input of $\mathcal{A}(p, q)$: each of MONITORING$_p[q]$ and ACTIVE-FOR$_q[p]$ can be (1) eventually always on, (2) eventually always off, or (3) oscillating between on and off, forever. Furthermore, there are many possibilities for the behaviors of p and q: (1) p may crash or not, (2) q may crash or not, and (3) q may be p-timely or not. To define $\mathcal{A}(p, q)$, we must specify its output in all the above cases. This is done as follows:

Definition 7 In every run R of $\mathcal{A}(p, q)$, if p is correct in R then the following properties hold:

– STATUS$_p[q]$ properties

1. If there is a time after which MONITORING$_p[q]$=*off* then there is a time after which STATUS$_p[q]$=?

The input of $\mathcal{A}(p,q)$ consists of two process-local variables:

1. $\text{MONITORING}_p[q] \in \{on, off\}$ at p — used by p to indicate whether it wants to monitor q.
2. $\text{ACTIVE-FOR}_q[p] \in \{on, off\}$ at q — used by q to indicate whether it is active for p.

The output of $\mathcal{A}(p,q)$ consists of two process-local variables:

1. $\text{STATUS}_p[q] \in \{active, inactive, ?\}$ — estimate of q's current status; "?" means "I don't know".
2. $\text{FAULTCNTR}_p[q] \in \mathbb{N}$ — number of times q was suspected of not being p-timely.

Fig. 1 Input and output variables of activity monitor $\mathcal{A}(p,q)$

2. If there is a time after which $\text{MONITORING}_p[q]{=}on$ then there is a time after which $\text{STATUS}_p[q]{\neq}?$.
3. If q crashes or there is a time after which $\text{ACTIVE-FOR}_q[p]{=}off$ then there is a time after which $\text{STATUS}_p[q]{\neq}active$.
4. If q is p-timely and there is a time after which $\text{ACTIVE-FOR}_q[p]{=}on$ then there is a time after which $\text{STATUS}_p[q]{\neq}inactive$.

– $\text{FAULTCNTR}_p[q]$ properties

5. $\text{FAULTCNTR}_p[q]$ is bounded if *any* of the following conditions hold:
   (a) q is p-timely
   (b) q crashes
   (c) there is a time after which $\text{ACTIVE-FOR}_q[p] = off$
   (d) there is a time after which $\text{MONITORING}_p[q] = off$
6. $\text{FAULTCNTR}_p[q]$ increases without bound if *all* of the following conditions hold:
   (a) q is not p-timely
   (b) q is correct
   (c) there is a time after which $\text{ACTIVE-FOR}_q[p] = on$
   (d) there is a time after which $\text{MONITORING}_p[q] = on$

Intuitively, Properties 1 and 2 indicate how $\text{STATUS}_p[q]$ depends on $\text{MONITORING}_p[q]$, while Properties 3 and 4 indicate how it depends on $\text{ACTIVE-FOR}_q[p]$ and the scheduling of q. For example, if q crashes then, by Property 3, there is a time after which $\text{STATUS}_p[q] = inactive$ or $\text{STATUS}_p[q] = ?$. If, in addition, there is a time after which $\text{MONITORING}_p[q]{=}on$ then Property 2 implies that there is a time after which $\text{STATUS}_p[q] = inactive$.

Properties 5 and 6 specify the behavior $\text{FAULTCNTR}_p[q]$. Note the Property 6 is not the converse of Property 5 (e.g., the negation of "there is a time after which X" is *not* "there is a time after which not X").

It is easy to implement an activity monitor $\mathcal{A}(p,q)$ using an atomic register R. If $p = q$ the implementation is trivial. If $p \neq q$, the detailed algorithm code is given in Fig. 2 and its key ideas are the following. When q is active for p, q periodically writes an increasing counter to R. If q wants to indicate it is no longer active for p, q writes a special value -1 to R, to indicate it is stopping willingly (instead of crashing). When p does not monitor q, p sets $\text{STATUS}_p[q]$ to "?". When p monitors q, p checks if R increases periodically and, if so, p sets $\text{STATUS}_p[q]$ to *active*. Otherwise, p times out on R (we use adaptive timeouts that increase over time). When a timeout happens, p sets $\text{STATUS}_p[q]$ to *inactive* and p may or may not increment $\text{FAULTCNTR}_p[q]$: p increments $\text{FAULTCNTR}_p[q]$ if (a) $R \neq -1$ *and* (b) R increased since the last time p incremented $\text{FAULTCNTR}_p[q]$. Condition (a) prevents $\text{FAULTCNTR}_p[q]$ from increasing forever if q stops being active for p, which is necessary to ensure part (c) of Property 5 above. Condition (b) prevents $\text{FAULTCNTR}_p[q]$ from increasing forever if q crashes, which is necessary to ensure part (b) of Property 5 above.

In the appendix, we show the following:

Theorem 3 *For any pair of processes $p \neq q$, the algorithm in Fig. 2 implements an activity monitor $\mathcal{A}(p,q)$ using registers.*

4.2 Implementing Ω_Δ using activity monitors and registers

We now give an algorithm for Ω_Δ in a system with registers where every pair of processes (p,q) is equipped with an activity monitor $\mathcal{A}(p,q)$. This algorithm does not have any synchrony mechanisms, such as timers and timeouts, because synchrony has been completely incorporated into the activity monitors.

The algorithm, shown in Fig. 3, uses a shared register *CounterRegister*$[p]$ for each process p; this register counts roughly how many times p has been considered "bad" for leadership. When a process p is a candidate for leadership, p periodically queries $\mathcal{A}(p,q)$ for each process q. Recall that $\mathcal{A}(p,q)$ outputs a counter $\text{FAULTCNTR}_p[q]$ and a status $\text{STATUS}_p[q]$. Process p uses $\text{FAULTCNTR}_p[q]$ to detect "bad" processes: if p sees that $\text{FAULTCNTR}_p[q]$ increases then p increments *CounterRegister*$[q]$ to "punish" q. Process p uses the vector STATUS_p to determine the set $activeSet_p$ of

Fig. 2 Implementation of $\mathcal{A}(p, q)$ using registers. The *top* shows code for the monitored process q, while the *bottom* shows code for the monitoring process p

```
CODE FOR MONITORED PROCESS q:
{ A(p,q)-Input : ACTIVE-FOR[p] }
{ Initial state }
    HbRegister[q,p] = −1  { shared register written by q and read by p. 'Hb' stands for heartbeat }
    hbCounter = 0                                                              { local variable }
{ Main code }
    repeat forever
        WRITE(HbRegister[q,p], −1)
        while ACTIVE-FOR[p] = off do skip
        while ACTIVE-FOR[p] = on do
            hbCounter ← hbCounter + 1
            WRITE(HbRegister[q,p], hbCounter)
```

```
CODE FOR MONITORING PROCESS p:
{ A(p,q)-Input : MONITORING[q] }
{ A(p,q)-Output : ⟨STATUS[q], FAULTCNTR[q]⟩ }
{ Initial state }
    STATUS[q] = ?
    FAULTCNTR[q] = 0
    HbRegister[q,p] = −1                 { shared register written by q and read by p }
    hbTimeout = 1                                                    { local variable }
    hbTimer = 1                                                      { local variable }
    hbCounter = 0                                                    { local variable }
    prevHbCounter = 0                                                { local variable }
    allow_increment = true                                           { local variable }
{ Main code }
    repeat forever
        STATUS[q] ← ?
        while MONITORING[q] = off do skip
        hbTimer ← hbTimeout
        while MONITORING[q] = on do
            if hbTimer ≥ 1 then hbTimer ← hbTimer − 1
            if hbTimer = 0 then
                hbTimer ← hbTimeout
                prevHbCounter ← hbCounter
                hbCounter ← READ(HbRegister[q,p])
                if hbCounter < 0 then STATUS[q] ← inactive
                if hbCounter ≥ 0 and hbCounter > prevHbCounter then
                    STATUS[q] ← active
                    allow_increment ← true
                if hbCounter ≥ 0 and hbCounter ≤ prevHbCounter then
                    STATUS[q] ← inactive
                    if allow_increment then
                        FAULTCNTR[q] ← FAULTCNTR[q] + 1
                        hbTimeout ← hbTimeout + 1
                        allow_increment ← false
```

processes q with $\text{STATUS}_p[q] = \textit{active}$; p also includes itself in $\textit{activeSet}_p$. Process p picks its leader as the process ℓ in $\textit{activeSet}_p$ with smallest *CounterRegister*$[\ell]$. If p picks itself as leader then p sets $\mathcal{A}(p, q)$'s $\text{ACTIVE-FOR}_p[q]$ to *on* (for every process q). Otherwise, p sets $\text{ACTIVE-FOR}_p[q]$ to *off*. Intuitively, a process is perceived to be active only if it considers itself to be the leader.

Every time p stops and starts being a candidate for leadership, p increments its own *CounterRegister*$[p]$ as a "self-punishment". This ensures that a process r that stops and starts being a candidate infinitely often has an unbounded *CounterRegister*$[r]$, which is necessary to ensure that eventually r is not chosen as leader. Without this self-punishment, it is easy to find a scenario where r has the smallest *CounterRegister*$[-]$ and leadership oscillates forever between r and another process.

Figure 3 shows the code in detail. Initially, p sets $\textit{leader}_p$ to ?, $\text{MONITORING}_p[q]$ to *off* and $\text{ACTIVE-FOR}_p[q]$ to *off* for every process q. While $\text{CANDIDATE}_p = \textit{false}$, p does nothing. When p finds that $\text{CANDIDATE}_p = \textit{true}$, p

```
CODE FOR PROCESS p:

{ Ω_Δ-Input : CANDIDATE }
{ Ω_Δ-Output : LEADER }
{ A(p, q)-Input : MONITORING[q] }
{ A(p, q)-Output : ⟨STATUS[q], FAULTCNTR[q]⟩ }
{ A(q, p)-Input : ACTIVE-FOR[q] }

{ Initial state }
    LEADER = ?
    ∀q ∈ Π : MONITORING[q] = off ∧ ACTIVE-FOR[q] = off
    ∀q ∈ Π : faultCntr[q] = 0 ∧ maxFaultCntr[q] = 0          { local variables }
    ∀q ∈ Π : counter[q] = 0                                   { local variables }
    activeSet = {p}                                           { local variable }
    CounterRegister[p] = 0                                    { shared register }

{ Main code }
1  repeat forever
2      LEADER ← ?
3      for each q ∈ Π do MONITORING[q] ← off
4      for each q ∈ Π do ACTIVE-FOR[q] ← off
5      while CANDIDATE = false do skip
6      for each q ∈ Π do MONITORING[q] ← on
7      counter[p] ← READ(CounterRegister[p])
8      WRITE(CounterRegister[p], counter[p] + 1)
9      while CANDIDATE = true do
10         for each q ∈ Π do
               { consult activity monitor A(p, q) about status of q }
11             repeat ⟨status[q], faultCntr[q]⟩ ← ⟨STATUS[q], FAULTCNTR[q]⟩
               until status[q] ≠ ?
12         activeSet ← {q : q ∈ Π ∧ status[q] = active} ∪ {p}
13         for each q ∈ Π do counter[q] ← READ(CounterRegister[q])
14         LEADER ← ℓ such that (counter[ℓ], ℓ) = min{(counter[q], q) : q ∈ activeSet}
15         if LEADER = p then
16             for each q ∈ Π do ACTIVE-FOR[q] ← on
17         else for each q ∈ Π do ACTIVE-FOR[q] ← off
18         for each q ∈ Π do
19             if faultCntr[q] > maxFaultCntr[q] then
20                 WRITE(CounterRegister[q], counter[q] + 1)
21                 maxFaultCntr[q] ← faultCntr[q]
```

Fig. 3 Implementation of Ω_Δ using activity monitors and registers

sets $\text{MONITORING}_p[q]$ to *on* for every process q, to indicate it wants $\mathcal{A}(p, q)$ to monitor q. Then, p increments *CounterRegister*$[p]$. While p finds that $\text{CANDIDATE}_p = \textit{true}$, p repeats the following actions. First, p queries its activity monitors $\mathcal{A}(p, q)$ until it gets a non-? status from each process q. Then, p sets $\textit{activeSet}_p$ to contain itself and every process q that is considered active by $\mathcal{A}(p, q)$. Next, p picks its leader as the process ℓ in $\textit{activeSet}_p$ with smallest *CounterRegister*$[\ell]$. If p picks itself, it sets $\text{ACTIVE-FOR}_p[q]$ to *on* otherwise it sets it to *off*, for every process q. Next, if p finds that $\text{FAULTCNTR}_p[q]$ increased then p increments *CounterRegister*$[q]$.

Correctness of this algorithm is given by the following:

Theorem 4 *The algorithm in Fig. 3 implements* Ω_Δ *in a system with registers where every pair of processes* (p, q) *is equipped with an activity monitor* $\mathcal{A}(p, q)$.

We now proceed to show this theorem. Henceforth, we consider an arbitrary run R of the algorithm.

We first show that no correct process gets stuck forever during the execution of an iteration of the loop in lines 9–21.

Lemma 5 *Every correct process completes every iteration of the while loop in lines* 9–21 *that it starts.*

Proof Suppose, by contradiction, that some correct process p gets stuck forever during the execution of an iteration of the loop in lines 9–21. It is easy to see that the only place where p can get stuck is in the repeat-until loop of line 11. Let q' be the value of variable q of p while p is executing this loop. Before entering the loop in lines 9–21, p sets $\text{MONITORING}_p[q']$ to *on* in line 6, and $\text{MONITORING}_p[q']$ is still equal to *on* when p gets stuck in the loop of line 11. Thus, there is a time after which $\text{MONITORING}_p[q'] = \textit{on}$. By Property (2) of $\mathcal{A}(p, q')$, there is a time after which $\text{STATUS}_p[q'] \neq$?. Thus, p does not get stuck forever executing the loop of line 11 with $q = q'$—a contradiction.

We classify correct processes into the following three subsets (according to their behavior in run R):

Definition 8

- *ncandidates* is the set of correct processes that execute the body of the while loop in lines 9–21 finitely many times.
- *infcandidates* is the set of correct processes that execute the body of the while loop in lines 9–21 infinitely many times.
- *pcandidates* is the set of correct processes that execute the body of the while loop in lines 9–21 infinitely many times *and* eventually execute forever in this loop.

Note that *infcandidates* and *ncandidates* form a partition of the set of correct processes, and *pcandidates* is a subset of *infcandidates*.

To prove that the algorithm satisfies the properties of Ω_Δ, we first relate the sets *pcandidates*, *ncandidates*, and *infcandidates* (which we will use to prove properties of the algorithm) to the sets *Pcandidates*, *Ncandidates*, and *Rcandidates* (which are used to specify Ω_Δ).

Lemma 6 *Pcandidates $\subseteq$ pcandidates, Ncandidates $\subseteq$ ncandidates, and Pcandidates $\cup$ Rcandidates $\supseteq$ infcandidates.*

Proof Let $p \in$ *Pcandidates*. By definition, p is correct and there is a time after which $\text{CANDIDATE}_p = \mathit{true}$. Thus, from the code of the algorithm, it is clear that p eventually executes forever in the loop in lines 9–21. By Lemma 5, p executes this loop infinitely many times. Therefore, by definition, $p \in$ *pcandidates*.

Let $p \in$ *Ncandidates*. By definition, p is correct and there is a time after which $\text{CANDIDATE}_p = \mathit{false}$. Thus, from the code of the algorithm, it is clear that p executes the body of the loop in lines 9–21 finitely many times. Therefore, by definition, $p \in$ *ncandidates*.

Let $p \in$ *infcandidates*. Thus, p is correct and $p \notin$ *ncandidates*. By the above, $p \notin$ *Ncandidates*. Thus, $p \in$ *Pcandidates* $\cup$ *Rcandidates*.

Lemma 7 *For every process $p \in$ ncandidates, there is a time after which (a) $\text{LEADER}_p = ?$ and (b) for every process $q \in \Pi$, $\text{MONITORING}_p[q] = \mathit{off}$ and $\text{ACTIVE-FOR}_p[q] = \mathit{off}$.*

Proof Let $p \in$ *ncandidates*. By definition of *ncandidates* and Lemma 5, it is clear that p eventually executes forever in the empty loop of line 5. Note that just before entering this loop, p sets LEADER_p to ? in line 2 and, for every process $q \in \Pi$, p sets $\text{MONITORING}_p[q]$ to *off* in line 3 and $\text{ACTIVE-FOR}_p[q]$ to *off* in line 4.

Corollary 4 *For every process $p \in$ Ncandidates, there is a time after which $\text{LEADER}_p = ?$.*

Proof By Lemma 6, *Ncandidates* $\subseteq$ *ncandidates*. The corollary is now immediate from Part (a) of Lemma 7.

By the above corollary, Property (2) of Ω_Δ (Definition 5) is satisfied in run R of the algorithm. We now proceed to show that Property (1) of Ω_Δ is also satisfied in run R. Roughly speaking, the proof will proceed as follows. Assume that there is a timely process in *Pcandidates*. We show that if p is one such process then *CounterRegister*$[p]$ eventually stops changing—intuitively, processes stop "punishing" p. Then, for each process p, we define c_p to be the final value of *CounterRegister*$[p]$ if it stops changing or $c_p = \infty$ otherwise. We let ℓ to be the process p with smallest c_p, breaking ties by process id. We then show that eventually ℓ picks itself as leader forever, that is, there is a time after which $\text{LEADER}_\ell = \ell$. This proves part (a) of Property (1) of Ω_Δ. Because ℓ sets $\text{ACTIVE-FOR}_\ell[p]$ to *on* exactly when $\text{LEADER}_\ell = \ell$, it follows that, for every process p, there is a time after which $\text{ACTIVE-FOR}_\ell[p] = \mathit{on}$. We then show that, for every process $q \neq \ell$, $\text{LEADER}_q \neq q$. Thus, for every $q \neq \ell$, there is a time after which $\text{ACTIVE-FOR}_q[p] = \mathit{off}$ for every process p. If there is a time after which $\text{ACTIVE-FOR}_q[p] = \mathit{off}$ and $p \neq q$, we argue that there is a time after which p does not pick q as its leader. Thus, for every process p, there is a time after which $\text{LEADER}_p \in \{p, \ell, ?\}$. However, when $p \neq \ell$, we showed that $\text{LEADER}_p \neq p$. Thus, there is a time after which $\text{LEADER}_p \in \{\ell, ?\}$. This proves part (c) of Property (1) of Ω_Δ. Finally, we argue that for every process p in *pcandidates*, $\text{LEADER}_p \neq ?$. Since *Pcandidates* $\subseteq$ *pcandidates*, this now proves part (b) of Property (1) of Ω_Δ.

We now proceed with the detailed proof.

Definition 9 Let *Timely* $= \{q : q$ is timely in run $R\}$.

If *Pcandidates* $\cap$ *Timely* $= \emptyset$, then Property (1) of Ω_Δ is trivially satisfied. Henceforth (from Lemmas 8 to 24) we assume that

Assumption 5 *Pcandidates $\cap$ Timely $\neq \emptyset$*

and show that Property (1) of Ω_Δ is also satisfied in this case.

Lemma 8 *pcandidates $\cap$ Timely $\neq \emptyset$.*

Proof By Lemma 6, *Pcandidates* $\subseteq$ *pcandidates*. By Assumption 5, *Pcandidates* $\cap$ *Timely* $\neq \emptyset$. Thus, *pcandidates* $\cap$ *Timely* $\neq \emptyset$.

Lemma 9 *For every process p, if some process writes to CounterRegister$[p]$ infinitely many times then CounterRegister$[p]$ increases without bound.*[7]

[7] Recall that we say v increases without bound if for every $k \in \mathbb{N}$ there is a time after which $v > k$.

Proof Let p be some process, and suppose that some process q writes to *CounterRegister*$[p]$ infinitely many times. First note that (*) *CounterRegister*$[p]$ is written only in lines 8 and 20, using 1 plus a value read from *CounterRegister*$[p]$ in lines 7 and 13, respectively.

We claim that for every integer $i \geq 0$, there is a time after which *CounterRegister*$[p] \geq i$. This claim proves the lemma.

We show the claim by induction on i. For the base case ($i = 0$), note that initially *CounterRegister*$[p] = 0$. Moreover, from (*), *CounterRegister*$[p] \geq 0$ always holds. This shows the base case.

Now suppose the claim holds for i, that is, there is a time t_i after which *CounterRegister*$[p] \geq i$. We show that there is a time t_{i+1} after which *CounterRegister*$[p] \geq i + 1$. From (*), there is a time $t_i' > t_i$ after which, if *CounterRegister*$[p]$ is written, then it is written with 1 plus a value read from *CounterRegister*$[p]$ after time t_i. By assumption, such a value read from *CounterRegister*$[p]$ is at least i. Thus, if *CounterRegister*$[p]$ is written after time t_i' then forever after *CounterRegister*$[p] \geq i + 1$. Since q writes to *CounterRegister*$[p]$ infinitely many times, q writes to *CounterRegister*$[p]$ after time t_i'. After that, *CounterRegister*$[p] \geq i + 1$. This shows the claim.

Corollary 5 *For every process* p, *CounterRegister*$[p]$ *increases without bound or it stops changing.*

Proof Let p be a process. If *CounterRegister*$[p]$ never stops changing then some process writes to *CounterRegister*$[p]$ infinitely many times. By Lemma 9, *CounterRegister*$[p]$ increases without bound.

Lemma 10 *Let p and q be processes such that $p \in$ infcandidates. Then* FAULTCNTR$_p[q]$ *increases without bound if and only if p writes to CounterRegister$[q]$ infinitely many times in line* 20.

Proof Let p and q be processes such that $p \in$ *infcandidates*.

First, suppose that FAULTCNTR$_p[q]$ increases without bound. Since $p \in$ *infcandidates*, p executes line 11 infinitely often, and so *faultCntr*$_p[q]$ also increases without bound. Also, p executes the test *faultCntr*$_p[q] >$ *maxFaultCntr*$_p[q]$ in line 19 infinitely many times. From the way p sets *maxFaultCntr*$_p[q]$ in line 21, it is clear that p writes to *CounterRegister*$[q]$ infinitely many times in line 20.

Now, suppose that p writes to *CounterRegister*$[q]$ infinitely many times in line 20. Thus, p finds that *faultCntr*$_p[q] >$ *maxFaultCntr*$_p[q]$ infinitely many times in line 19. So, *faultCntr*$_p[q]$ increases without bound. Variable *faultCntr*$_p[q]$ is set to FAULTCNTR$_p[q]$ in line 11, and so FAULTCNTR$_p[q]$ also increases without bound.

Lemma 11 *For every process $q \in$ pcandidates $\cap$ Timely, CounterRegister$[q]$ stops changing.*

Proof Assume, by contradiction, that for some process $q \in$ *pcandidates* $\cap$ *Timely*, *CounterRegister*$[q]$ changes infinitely many times. There are only two lines where *CounterRegister*$[q]$ can be changed: (1) in line 8, *CounterRegister*$[q]$ is written by q, and (2) in line 20, *CounterRegister*$[q]$ is written by some process. However, q executes line 8 only finitely many times (since $q \in$ *pcandidates*). Therefore, processes write to *CounterRegister*$[q]$ infinitely many times in line 20. Since there are only finitely many processes, some process p writes to *CounterRegister*$[q]$ infinitely many times in line 20. Thus, $p \in$ *infcandidates* and so, by Lemma 10, FAULTCNTR$_p[q]$ increases without bound. But, $q \in$ *Timely*, so q is p-timely, and thus, by Property (5) of $\mathcal{A}(p, q)$, FAULTCNTR$_p[q]$ is bounded—a contradiction.

Lemma 12 *For every process $p \in$ infcandidates $-$ pcandidates, CounterRegister$[p]$ increases without bound.*

Proof Let $p \in$ *infcandidates* $-$ *pcandidates*. By definition of *infcandidates* and *pcandidates*, it is clear that p enters and exits the body of the loop in lines 9–21 infinitely many times. Each time it enters this loop, p first writes to *CounterRegister*$[p]$ in line 8. Thus, by Lemma 9, *CounterRegister*$[p]$ increases without bound.

Definition 10 For every process p, we define c_p as follows. If *CounterRegister*$[p]$ stops changing then c_p is the final value of *CounterRegister*$[p]$; otherwise, $c_p = \infty$.

We now define ℓ as the process in *pcandidates* with smallest c_p, breaking ties using the process id. Note that ℓ is well defined because, by Lemma 8, the set *pcandidates* is not empty.

Definition 11 Let ℓ be the process such that $(c_\ell, \ell) = \min\{(c_p, p) : p \in \textit{pcandidates}\}$.

Lemma 13 *There is a time after which CounterRegister$[\ell]$= $c_\ell < \infty$.*

Proof By Lemmas 8 and 11, there is a process $k \in$ *pcandidates* such that *CounterRegister*$[k]$ stops changing. Thus, by the definition of c_k, $c_k < \infty$. By the definition of ℓ, $(c_\ell, \ell) \leq (c_k, k)$, and so $c_\ell < \infty$. By the definition of c_ℓ, there is a time after which *CounterRegister*$[\ell] = c_\ell$.

Lemma 14 *For every process $p \neq \ell$ such that $p \in$ activeSet$_\ell$ infinitely often, there is a time after which* $(\textit{CounterRegister}[\ell], \ell) < (\textit{CounterRegister}[p], p)$.

Proof Suppose $p \in$ *activeSet*$_\ell$ infinitely often and $p \neq \ell$. Since $\ell \in$ *pcandidates*, ℓ executes line 12 infinitely many times. By the way ℓ sets *activeSet*$_\ell$ in line 12, it is clear that STATUS$_\ell[p] =$ *active* infinitely often. By the contrapositive of Property (3) of $\mathcal{A}(\ell, p)$, we have (*) p is correct and ACTIVE-FOR$_p[\ell] =$ *on* infinitely often.

By Lemma 13, there is a time after which $\mathit{CounterRegister}[\ell] = c_\ell < \infty$. By Corollary 5, there are two possible cases:

Case 1 *CounterRegister*$[p]$ *increases without bound.* Thus, there is a time after which $c_\ell < \mathit{CounterRegister}[p]$. Since there is a time after which $\mathit{CounterRegister}[\ell] = c_\ell$, there is a time after which $(\mathit{CounterRegister}[\ell], \ell) < (\mathit{CounterRegister}[p], p)$.

Case 2 *CounterRegister*$[p]$ *stops changing.* By definition of c_p, there is a time after which $\mathit{CounterRegister}[p] = c_p < \infty$. It now suffices to show that $(c_\ell, \ell) < (c_p, p)$. By (*) and Lemma 7, $p \notin \mathit{ncandidates}$. So $p \in \mathit{infcandidates}$. Since *CounterRegister*$[p]$ stops changing, by Lemma 12, $p \in \mathit{pcandidates}$. Thus, by the definition of ℓ and the fact that $p \neq \ell$, we have $(c_\ell, \ell) < (c_p, p)$.

Since $\mathit{activeSet}_p$ is initialized to $\{p\}$ and p never removes itself from $\mathit{activeSet}_p$, we have the following:

Observation 6 For every process p, $p \in \mathit{activeSet}_p$.

We now show that ℓ eventually picks itself as the leader.

Lemma 15 *There is a time after which* $\text{LEADER}_\ell = \ell$.

Proof Since $\ell \in \mathit{pcandidates}$, (a) there is a time after which the only place where ℓ can set LEADER_ℓ is in line 14, and (b) ℓ sets LEADER_ℓ in line 14 infinitely many times. Each time ℓ sets LEADER_ℓ in line 14, ℓ sets LEADER_ℓ to the process q in $\mathit{activeSet}_\ell$ with smallest $(\mathit{counter}_\ell[q], q)$, where the $\mathit{counter}_\ell$ vector has values read from the *CounterRegister* vector in line 13. Since ℓ is correct, by Observation 6, $\ell \in \mathit{activeSet}_\ell$. From Lemma 14, we conclude that there is a time after which $\text{LEADER}_\ell = \ell$.

Lemma 16 *For every process p, there is a time after which* $\text{ACTIVE-FOR}_\ell[p] = \mathit{on}$.

Proof Let p be any process. Since $\ell \in \mathit{pcandidates}$, (a) there is a time after which the only place where ℓ can set $\text{ACTIVE-FOR}_\ell[p]$ is inside the if-then-else statement of lines 15–17, and (b) ℓ sets $\text{ACTIVE-FOR}_\ell[p]$ in this if-then-else statement infinitely many times. By Lemma 15, there is a time after which $\text{LEADER}_\ell = \ell$. From the way ℓ sets $\text{ACTIVE-FOR}_\ell[p]$ in the if-then-else statement, it is now clear that there is a time after which $\text{ACTIVE-FOR}_\ell[p] = \mathit{on}$.

Lemma 17 $\ell \in \mathit{Timely}$.

Proof Suppose, by contradiction, that $\ell \notin \mathit{Timely}$.

By Lemma 8, there exists some process $p \in \mathit{pcandidates} \cap \mathit{Timely}$. We now show that p and ℓ meet the conditions of Property 6 of $\mathcal{A}(p, \ell)$, implying that $\text{FAULTCNTR}_p[\ell]$ increases without bound.

(a) By assumption, $\ell \notin \mathit{Timely}$. Moreover, since $p \in \mathit{pcandidates} \cap \mathit{Timely}$, $p \in \mathit{Timely}$. By Corollary 2, ℓ is not p-timely.
(b) By definition of ℓ, $\ell \in \mathit{pcandidates}$. So, ℓ is correct.
(c) By Lemma 16, there is a time after which $\text{ACTIVE-FOR}_\ell[p] = \mathit{on}$.
(d) Since $p \in \mathit{pcandidates}$, eventually p executes forever in the loop in lines 9–21. Before getting stuck in this loop, p sets $\text{MONITORING}_p[\ell]$ to *on* in line 6 and p does not set $\text{MONITORING}_p[\ell]$ to *off* afterwards. Thus, there is a time after which $\text{MONITORING}_p[\ell] = \mathit{on}$.

By Property 6 of $\mathcal{A}(p, \ell)$, $\text{FAULTCNTR}_p[\ell]$ increases without bound. By Lemma 10, p writes to *CounterRegister*$[\ell]$ infinitely many times. Thus, by Lemma 9, *CounterRegister*$[\ell]$ increases without bound. But, by Lemma 13, *CounterRegister*$[\ell]$ stops changing—a contradiction.

Lemma 18 *For every process* $p \in \mathit{infcandidates}$*, there is a time after which* $\ell \in \mathit{activeSet}_p$.

Proof Let $p \in \mathit{infcandidates}$. By Lemma 16, there is a time after which $\text{ACTIVE-FOR}_\ell[p] = \mathit{on}$. By Lemma 17, ℓ is timely, and so ℓ is p-timely. Since p is correct, by Property (4) of $\mathcal{A}(p, \ell)$, (*) there is a time after which $\text{STATUS}_p[\ell] \neq \mathit{inactive}$, i.e., $\text{STATUS}_p[\ell] \in \{?, \mathit{active}\}$.

Since $p \in \mathit{infcandidates}$, p executes lines 11 and 12 infinitely many times. In line 11, p sets $\mathit{status}_p[\ell]$ to $\text{STATUS}_p[\ell]$, and this is the only line in which p sets $\mathit{status}_p[\ell]$. Thus, from (*), there is a time after which $\mathit{status}_p[\ell] \in \{?, \mathit{active}\}$. Moreover, each time p executes line 12, $\mathit{status}_p[\ell] \neq ?$ (because of the condition of the loop in line 11). So there is a time after which, every time p executes line 12, p finds that $\mathit{status}_p[\ell] = \mathit{active}$. From the way p sets $\mathit{activeSet}_p$ in line 12, there is a time after which $\ell \in \mathit{activeSet}_p$.

The next lemma shows that, except for ℓ, all processes in *infcandidates* eventually stop considering themselves as the leader.

Lemma 19 *For every process* $p \in \mathit{infcandidates} - \{\ell\}$*, there is a time after which* $\text{LEADER}_p \neq p$.

Proof Let $p \in \mathit{infcandidates} - \{\ell\}$. By Lemma 18, there is a time t_1 after which $\ell \in \mathit{activeSet}_p$.

We claim that there is a time t_2 after which $(\mathit{CounterRegister}[\ell], \ell) < (\mathit{CounterRegister}[p], p)$. To prove this claim, first note that, by Lemma 13, there is a time after which $\mathit{CounterRegister}[\ell] = c_\ell < \infty$. By Corollary 5, *CounterRegister*$[p]$ increases without bound or it stops changing. If *CounterRegister*$[p]$ increases without bound, the claim immediately follows. Now assume that *CounterRegister*$[p]$ stops changing. By the definition of c_p, there is a time after which $\mathit{CounterRegister}[p] = c_p < \infty$.

To prove the claim it now suffices to show $(c_\ell, \ell) < (c_p, p)$. Since $p \in$ *infcandidates* and *CounterRegister*$[p]$ stops changing, by Lemma 12, $p \in$ *pcandidates*. Thus, by the definition of ℓ and the fact that $p \neq \ell$, we have $(c_\ell, \ell) < (c_p, p)$—this shows the claim.

There are only two places in the code where p can set LEADER$_p$: (1) in line 2, where p sets LEADER$_p$ to ?, and (2) in line 14, where p sets LEADER$_p$ to the process q in *activeSet*$_p$ with the smallest $(counter_p[q], q)$, where the *counter*$_p$ vector has values read from the *CounterRegister* vector in line 13. From the above, if p executes lines 13 and 14 after time $\max\{t_1, t_2\}$, it finds that (a) $\ell \in activeSet_p$ and (b) $(counter_p[\ell], \ell) < (counter_p[p], p)$. So if p executes lines 13 and 14 after time $\max\{t_1, t_2\}$, p sets LEADER$_p$ to a process different from p. Since $p \in$ *infcandidates*, p executes lines 13 and 14 infinitely many times, and so p executes lines 13 and 14 after time $\max\{t_1, t_2\}$. We conclude that there is a time after which LEADER$_p \neq p$.

Lemma 20 *For every correct process $q \neq \ell$ and every process p, there is a time after which* ACTIVE-FOR$_q[p] = $ *off*.

Proof Let $q \neq \ell$ be a correct process and p be a process. If $q \in$ *ncandidates* then by Lemma 7, there is a time after which ACTIVE-FOR$_q[p] =$ *off*. Now suppose that $q \notin$ *ncandidates*. Since q is correct, $q \in$ *infcandidates*. So, q executes the if-then-else statement of lines 15–17 infinitely many times. In this if-then-else statement, q sets ACTIVE-FOR$_q[p]$ to *off* if LEADER$_q \neq q$ and q sets ACTIVE-FOR$_q[p]$ to *on* if LEADER$_q = q$. Moreover, this is the only statement where q can set ACTIVE-FOR$_q[p]$ to *on*. By Lemma 19 there is a time after which LEADER$_q \neq q$. Therefore, there is a time after which ACTIVE-FOR$_q[p] =$ *off*.

Lemma 21 *For every process $p \in$ infcandidates, there is a time after which activeSet$_p = \{p, \ell\}$.*

Proof Let $p \in$ *infcandidates*. By Lemma 18, there is a time after which $\ell \in activeSet_p$. Since $p \in$ *infcandidates*, p is correct, so by Observation 6, $p \in activeSet_p$. Therefore, there is a time after which both p and ℓ are in *activeSet*$_p$. We now prove that, for every $q \notin \{p, \ell\}$, there is a time after which $q \notin activeSet_p$. Let $q \notin \{p, \ell\}$. Either q crashes or, by Lemma 20, there is a time after which ACTIVE-FOR$_q[p] =$ *off*. Since p is correct, by Property (3) of $\mathcal{A}(p, q)$, there is a time after which STATUS$_p[q] \neq$ *active*. Since $p \in$ *infcandidates*, p sets $status_p[q]$ to STATUS$_p[q]$ in line 11 and then it sets *activeSet*$_p$ to $\{q : q \in \Pi \wedge status_p[q] = active\} \cup \{p\}$ in line 12, infinitely many times. Since there is a time after which STATUS$_p[q] \neq$ *active* and $q \neq p$, there is a time after which $q \notin activeSet_p$.

We now show that eventually, correct processes either choose ℓ or ? as their leader.

Lemma 22 *For every correct process p, there is a time after which* LEADER$_p \in \{?, \ell\}$.

Proof Let p be a correct process. If $p \in$ *ncandidates* then by Lemma 7, there is a time after which LEADER$_p$ = ?. Now assume that $p \notin$ *ncandidates*. Since p is correct, $p \in$ *infcandidates*. There are only two places in the code where p can set LEADER$_p$: (1) in line 2, where p sets LEADER$_p$ to ?, and (2) in line 14, where p sets LEADER$_p$ to a process in *activeSet*$_p$. Since $p \in$ *infcandidates*, p sets LEADER$_p$ in line 14 infinitely many times. By Lemma 21, there is a time after which $activeSet_p = \{p, \ell\}$. Therefore, there is a time after which LEADER$_p \in \{?, p, \ell\}$. If $p = \ell$ the lemma is immediate. If $p \neq \ell$, by Lemma 19, there is a time after which LEADER$_p \neq p$, and the lemma also follows.

Lemma 23 *For every process $p \in$ pcandidates, there is a time after which* LEADER$_p = \ell$.

Proof Let $p \in$ *pcandidates*. We claim that there is a time after which LEADER$_p \neq$?.

To prove this claim note that since $p \in$ *pcandidates*: (a) there is a time after which p does not execute line 2, which is the only place where LEADER$_p$ can be set to ?, and (b) p sets LEADER$_p$ in line 14 infinitely many times, and when it does so, it is clear that p sets LEADER$_p$ to a non-? value. So the claim holds.

From Lemma 22 and the above claim, there is a time after which LEADER$_p = \ell$.

Putting together the above results, we get:

Lemma 24 *$\ell \in$ (Pcandidates $\cup$ Rcandidates) $\cap$ Timely. Furthermore, the following holds:*

1. *There is a time after which* LEADER$_\ell = \ell$.
2. *For every process $p \in$ Pcandidates, there is a time after which* LEADER$_p = \ell$.
3. *For every process $p \in$ Rcandidates, there is a time after which* LEADER$_p \in \{?, \ell\}$.

Proof Since $\ell \in$ *pcandidates*, we have that $\ell \in$ *infcandidates*, and so by Lemma 6, $\ell \in$ *Pcandidates* $\cup$ *Rcandidates*. By Lemma 17, $\ell \in$ (*Pcandidates* $\cup$ *Rcandidates*) $\cap$ *Timely*. We now show that the above three properties hold:

1. This is Lemma 15.
2. Let $p \in$ *Pcandidates*. By Lemma 6, $p \in$ *pcandidates*. By Lemma 23, there is a time after which LEADER$_p = \ell$.
3. This follows immediately from Lemma 22 since every process in *Rcandidates* is correct.

Theorem 4 *The algorithm in Fig. 3 implements Ω_Δ in a system with registers where every pair of processes (p, q) is equipped with an activity monitor $\mathcal{A}(p, q)$.*

Proof Property (2) of Ω_Δ holds by Corollary 4. If $Pcandidates \cap Timely = \emptyset$, Property (1) of Ω_Δ trivially holds. If $Pcandidates \cap Timely \neq \emptyset$, Assumption 5 holds. In this case, we can apply Lemma 24 which shows that Property (1) of Ω_Δ holds.

From Theorems 3 and 4, we have

Theorem 7 *The algorithm obtained by combining the algorithms in Figs. 2 and 3 implements* Ω_Δ *in a system with registers.*

Note that this algorithm for implementing Ω_Δ with registers ensures that if $Pcandidates \cap Timely \neq \emptyset$ then there is a time after which the only processes that write to shared registers are the leader and processes in *Rcandidates*. Thus, in a precise sense, the implementation is "write efficient".

5 Implementing Ω_Δ using abortable registers

We now show how to implement Ω_Δ using (single-writer single-reader) abortable registers.[8] An abortable register is a very weak object because its read or write operations may abort if they are *concurrent*.[9] For example, suppose process p wants to communicate a value v to process q by writing v to abortable register R. Then, p needs to write v to R successfully (without aborting) at least once, and q needs to periodically read R to see if its value has changed. However, every time p writes to R it is possible that q reads R concurrently, causing both write and read to abort, and this could go on forever.

To implement Ω_Δ, we first give two communication mechanisms as building blocks: (1) a mechanism for p to send to q the final value of a variable (of p) that stops changing, provided p is q-timely (if p is not q-timely or the variable keeps changing forever, q may never see any of p's values), and (2) a mechanism for p to periodically communicate a heartbeat to q so that q can determine if p is q-timely or not (but p cannot convey any other information to q in this way). We then explain how these two weak communication mechanisms can be used to implement Ω_Δ.

Communicating the final value of a variable that eventually stops changing. Suppose p wants to communicate to q the latest content of p's local variable $msgTo[q]$. To do so, whenever p sees that $msgTo[q]$ changed to some new value v, p repeatedly writes v to R until the write is successful. At the same time q periodically reads R to check for new contents. To try to avoid concurrent execution, q slows down the rate at which it reads R if q thinks that p might be trying to write to R without success—this happens if the reads by q abort or return values that do not change. If p is q-timely, eventually q slows down (the rate at which it reads R) enough so that p executes its write solo, ensuring that eventually p's write is successful. In fact, if $msgTo[q]$ stops changing, eventually p writes successfully the final value of $msgTo[q]$ to R and stops writing to R. Thus, eventually q reads R without p writing concurrently, and q gets the final value.

Note that this mechanism may fail to communicate any information if p is not q-timely or if $msgTo[q]$ keeps changing forever. In both cases, there are runs in which *all* reads by q are concurrent with a write by p and they all abort.

The code details are shown in Fig. 4. There is a vector $MsgRegister[p, q]$ of abortable registers written by p and read by q, for every pair of distinct processes p and q. There are two procedures, *WriteMsgs*(*msgTo*) and *ReadMsgs*(), which are to be called by processes periodically. Procedure *WriteMsgs*(*msgTo*) serves for a process p to communicate the contents of $msgTo[q]$ to every process $q \neq p$. Variable $msgCurr[q]$ has the value of $msgTo[q]$ that p is currently trying to write to $MsgRegister[p, q]$ and $prevWriteDone[q]$ indicates whether the value of $msgCurr[q]$ has been written successfully to $MsgRegister[p, q]$. The procedure returns the vector *prevWriteDone*. Procedure *ReadMsgs*() serves for a process q to receive contents communicated by every process $p \neq q$. In this procedure, q reads $MsgRegister[p, q]$ for each p, every $readTimeout[p]$ invocations. If the read aborts or returns the same value as the last successful read then q increments $readTimeout[p]$. Otherwise, q resets $readTimeout[p]$ to 1 and sets $prevMsgFrom[p]$ to the value read. At the end of the procedure, q returns *prevMsgFrom*, which has the last successfully read message from every process.

Communicating a heartbeat. Suppose that a process p wants to communicate a "heartbeat signal" to q, which q can use to determine if p is q-timely or not. If processes had an atomic register $\hat{R}$, p could write an increasing counter to $\hat{R}$ and q could read $\hat{R}$ and verify that its value increases in a timely fashion. This scheme is problematic if we replace $\hat{R}$ with an abortable register R, for two reasons: (a) the writes of p to R may always abort and never take effect, and (b) the reads of q on R may always abort and so q never sees the value of R. We can avoid problem (a) by having q gradually slow the rate with which it reads R (as we did above in *ReadMsgs*), but how do we deal with problem (b)? The key idea is that if q reads R and the read aborts then q knows that p is writing some value to R, even if q does not know what the value is. Thus, an abort response indicates that p is alive. However, it does not indicate that p is q-timely: p may be

[8] A single-writer single-reader abortable register is an abortable register in which there is one designated process that can write to it and one designated process that can read it.

[9] An operation invoked by a process that crashes spans a finite interval of time which may extend beyond the time of the crash.

Fig. 4 Implementation of Ω_Δ using abortable registers—procedures for communicating the final value of a variable that stops changing

```
CODE FOR PROCESS p:
{ Initial state }
    ∀q ∈ Π − {p} : MsgRegister[p, q] = ⟨0, 0⟩      { abortable register written by p and read by q }
    ∀q ∈ Π − {p} : msgCurr[q] = ⟨0, 0⟩
    ∀q ∈ Π − {p} : prevMsgFrom[q] = ⟨0, 0⟩
    ∀q ∈ Π − {p} : readTimer[q] = 1
    ∀q ∈ Π − {p} : readTimeout[q] = 1
    ∀q ∈ Π − {p} : prevWriteDone[q] = true

procedure WriteMsgs(msgTo)
    for each q ∈ Π − {p} do
        if (not prevWriteDone[q]) or msgCurr[q] ≠ msgTo[q] then
            if prevWriteDone[q] then msgCurr[q] ← msgTo[q]
            res ← WRITE(MsgRegister[p, q], msgCurr[q])
            prevWriteDone[q] ← (res = ok)
    return prevWriteDone

procedure ReadMsgs()
    for each q ∈ Π−{p} do
        if readTimer[q] ≥ 1 then readTimer[q] ← readTimer[q] − 1
        if readTimer[q] = 0 then
            readTimer[q] ← readTimeout[q]
            res[q] ← READ(MsgRegister[q, p])
            if res[q] = ⊥ or res[q] = prevMsgFrom[q]
            then readTimeout[q] ← readTimeout[q] + 1
            else
                prevMsgFrom[q] ← res[q]
                readTimeout[q] ← 1
    return prevMsgFrom
```

slow and takes increasingly long to complete its writes to R, while all the reads by q keep aborting.

We solve this problem by using *two* heartbeat registers: p periodically writes increasing values to both registers, alternating between the two, and q reads both registers in alternation as well; q considers p to be q-timely only if, for both registers, the read aborts or returns a higher value than previously returned. If p took a long time to complete a write to one register, then a read on the other register would neither abort nor return a higher value, so q would not consider p as q-timely.

The details of this mechanism are shown in Fig. 5. Process p periodically calls procedure *SendHeartbeat*(*dest*), where *dest* is a boolean vector indicating to whom p wants to communicate its heartbeat. In this procedure, for every process q such that *dest*[q] is *true*, p writes an ever-increasing value to *HbRegister1*[p, q] and *HbRegister2*[p, q]. Process q calls procedure *ReceiveHeartbeat*() from time to time. In this procedure, q reads *HbRegister1*[p, q] and *HbRegister2*[p, q] every *hbTimeout*[p] invocations, for each process p. If, for both registers, the read aborts or returns a higher value than before, then q adds p to *activeSet*. Otherwise, q removes p from *activeSet* and increments *hbTimeout*[p]. At the end of the procedure, q returns *activeSet*—this is the set of processes that q considers to be q-timely.

The main Ω_Δ algorithm. We use the two communication mechanisms above to implement Ω_Δ. The algorithm, shown in Fig. 6, has some similarities with the algorithm of Sect. 4.2: processes use counters and choose the leader as the process with smallest counter among some set of active processes. However, we use some new techniques to determine the set of active processes and to maintain the counters.

To determine the set of active processes, candidate processes periodically call the procedures *SendHeartbeat* and *ReceiveHeartbeat*, as described above. *ReceiveHeartbeat* returns the set of active processes, which is then stored in a local variable $activeSet_p$ for each participant p.

To maintain the counters used to pick the leader, p keeps its own view of the counter of other processes in a local variable: $counter_p[q]$ has p's view of the counter of q. While p is a candidate for leadership, p communicates its own $counter_p[p]$ to other processes via procedure *WriteMsgs*, described before. Moreover, if p finds that q is not active, p punishes q by asking q to set its counter $counter_q[q]$ beyond the counter of p's current leader—a value sufficiently large to ensure that q is not picked as leader by p. This punishment is communicated also via procedure *WriteMsgs*. Procedure *WriteMsgs* returns a boolean vector, stored in *writeDone*, indicating for each process q whether p wrote successfully to the register readable by q. Recall that *WriteMsgs* only guarantees that a process p communicates a value successfully to q if (a) this value stops changing, and (b) p is q-timely and keeps calling *WriteMsgs* periodically.

In the proofs, we show that (a) always holds, that is, for every process p, both p's counter and any punishments sent by p stop changing. However, (b) poses a problem: if p is not timely then some candidates for leadership may receive

Fig. 5 Implementation of Ω_Δ using abortable registers—procedures for communicating a heartbeat

```
CODE FOR PROCESS p:
{ Initial state }
  ∀q ∈ Π − {p} : HbRegister1[p, q] = 0        { abortable register written by p and read by q }
  ∀q ∈ Π − {p} : HbRegister2[p, q] = 0        { abortable register written by p and read by q }
  ∀q ∈ Π − {p} : hbTimeout[q] = 1
  ∀q ∈ Π − {p} : hbTimer[q] = 1
  ∀q ∈ Π − {p} : prevHbCounter1[q] = 0
  ∀q ∈ Π − {p} : prevHbCounter2[q] = 0
  ∀q ∈ Π − {p} : hbCounter1[q] = 0
  ∀q ∈ Π − {p} : hbCounter2[q] = 0
  hbSendCounter = 0
  activeSet = {p}

procedure SendHeartbeat(dest)
  hbSendCounter ← hbSendCounter + 1
  for each q ∈ Π − {p} do
    if dest[q] then
      WRITE(HbRegister1[p, q], hbSendCounter)
      WRITE(HbRegister2[p, q], hbSendCounter)

procedure ReceiveHeartbeat()                                    { updates activeSet }
  for each q ∈ Π − {p} do
    if hbTimer[q] ≥ 1 then hbTimer[q] ← hbTimer[q] − 1
    if hbTimer[q] = 0 then
      hbTimer[q] ← hbTimeout[q]
      prevHbCounter1[q] ← hbCounter1[q]
      prevHbCounter2[q] ← hbCounter2[q]
      hbCounter1[q] ← READ(HbRegister1[q, p])
      hbCounter2[q] ← READ(HbRegister2[q, p])
      if (hbCounter1[q] = ⊥ or hbCounter1[q] ≠ prevHbCounter1[q]) and
         (hbCounter2[q] = ⊥ or hbCounter2[q] ≠ prevHbCounter2[q])
      then activeSet ← activeSet ∪ {q}
      else
        activeSet ← activeSet − {q}
        hbTimeout[q] ← hbTimeout[q] + 1
```

the latest value of $counter_p[p]$ while others never do so, creating an inconsistency. This is undesirable because it could cause different processes to pick different leaders. To avoid this problem, if p cannot communicate with q via *WriteMsgs* then p stops communicating heartbeats to q. This ensures the property that if q eventually considers p active forever then q eventually learns the final value of $counter_p[p]$—a property that is key for correctness of the algorithm.

Finally, like in the algorithm of Sect. 4.2, every time p becomes a candidate of Ω_Δ, it inflicts a "self-punishment". It does *not* do so simply by increasing $counter_p[p]$ (otherwise $counter_p[p]$ may never stop changing and thus *WriteMsgs* may not be able to communicate its value to other processes) but rather by setting $counter_p[p]$ beyond the counter of p's current leader.

Figure 6 shows the code in detail. Initially, p sets LEADER_p to ?. When p finds that CANDIDATE $= true$, p punishes itself by increasing $counter_p[p]$ beyond the counter of p's leader. While p finds that CANDIDATE $= true$, p repeats the following actions. First, p calls *SendHeartbeat*(*writeDone*), where *writeDone* indicates to whom p should send its heartbeat (its value comes from procedure *WriteMsgs*, below). Then, p calls *ReceiveHeartbeat* to update $activeSet_p$. Next, p picks its leader. For each q not in $activeSet_p$, p sets $actrTo_p[q]$ to be greater than the counter of p's leader (*actrTo* stands for "accusation counter to"). Intuitively, p wants to punish q by asking q to set its counter to at least $actrTo_q[p]$. Next, p assembles a message $msgTo_p[q]$ to be sent to q via procedure *WriteMsgs*. This message consists of $counter_p[p]$ and $actrTo_p[q]$. Then, p calls *WriteMsgs* and sets *writeDone* to the result—a boolean vector indicating whether, for each process q, p wrote successfully to the register readable by q. (Recall that *writeDone* determines to whom p communicates its heartbeat when p calls *SendHeartbeat*.) Next, p calls *ReadMsgs* to receive the pairs of counters and punishments that other processes are communicating to p. Using this information, p updates $counter_p[q]$, for every $q \neq p$, and p increases $counter_p[p]$ according to the punishments it received.

Correctness of this algorithm is given by the following:

Theorem 8 *The algorithm in Figs. 4, 5, and 6 implements Ω_Δ in a system with abortable registers.*

We now proceed to show this theorem. Henceforth, we consider an arbitrary run R of this algorithm.

Lemma 25 *Every correct process completes every iteration of the do-while loop in lines 44–57 that it starts.*

Fig. 6 Implementation of Ω_Δ using abortable registers—main code

```
CODE FOR PROCESS p:

{ Ω_Δ-Input : CANDIDATE }
{ Ω_Δ-Output : LEADER }

{ Initial state }
    LEADER = ?
    leader = p
    ∀q ∈ Π : counter[q] = 0
    ∀q ∈ Π − {p} : actrTo[q] = 0                  { actr stands for "accusation counter" }
    ∀q ∈ Π − {p} : writeDone[q] = false

{ Main code }
40  repeat forever
41      LEADER ← ?
42      while CANDIDATE = false do skip
43      counter[p] ← max{counter[p], counter[leader] + 1}
44      do
45          SendHeartbeat(writeDone)
46          ReceiveHeartbeat()                     { this computes the activeSet }

47          leader ← ℓ such that (counter[ℓ], ℓ) = min{(counter[q], q) : q ∈ activeSet}
48          LEADER ← leader

49          for each q ∈ Π − {p} do
50              if q ∉ activeSet then actrTo[q] ← max{actrTo[q], counter[leader] + 1}
51              msgTo[q] ← ⟨counter[p], actrTo[q]⟩

52          writeDone ← WriteMsgs(msgTo)
53          msgFrom ← ReadMsgs()

54          for each q ∈ Π−{p} do
55              ⟨counter[q], actrFrom[q]⟩ ← msgFrom[q]
56              counter[p] ← max{counter[p], actrFrom[q]}

57      while CANDIDATE = true
```

Proof This is clear because the body of the do-while loop in lines 44–57 has no unbounded loops.

We classify correct processes into the following three subsets (according to their behavior in run R):

Definition 12

- *ncandidates* is the set of correct processes that execute the body of the do-while loop in lines 44–57 finitely many times.
- *infcandidates* is the set of correct processes that execute the body of the do-while loop in lines 44–57 infinitely many times.
- *pcandidates* is the set of correct processes that execute the body of the do-while loop in lines 44–57 infinitely many times *and* eventually execute forever in this loop.

Note that *infcandidates* and *ncandidates* form a partition of the set of correct processes, and *pcandidates* is a subset of *infcandidates*.

To prove that the algorithm satisfies the properties of Ω_Δ, we first relate the sets *pcandidates*, *ncandidates*, and *infcandidates* (which we will use to prove properties of the algorithm) to the sets *Pcandidates*, *Ncandidates*, and *Rcandidates* (which are used to specify Ω_Δ).

Lemma 26 *$Pcandidates \subseteq pcandidates$, $Ncandidates \subseteq ncandidates$, and $Pcandidates \cup Rcandidates \supseteq infcandidates$.*

Proof (Similar to the proof of Lemma 6.) Let $p \in Pcandidates$. By definition, p is correct and there is a time after which $\text{CANDIDATE}_p = true$. Thus, from the code of the algorithm, it is clear that p eventually executes forever in the loop in lines 44–57. By Lemma 25, p executes this loop infinitely many times. Therefore, by definition, $p \in pcandidates$.

Let $p \in Ncandidates$. By definition, p is correct and there is a time after which $\text{CANDIDATE}_p = false$. Thus, from the code of the algorithm, it is clear that p executes the body of the loop in lines 44–57 finitely many times. Therefore, by definition, $p \in ncandidates$.

Let $p \in infcandidates$. Thus, p is correct and $p \notin ncandidates$. By the above, $p \notin Ncandidates$. Thus, $p \in Pcandidates \cup Rcandidates$.

Lemma 27 *For every process $p \in ncandidates$, there is a time after which $\text{LEADER}_p = ?$.*

Proof (Similar to the proof of Lemma 7.) Let $p \in ncandidates$. By definition of *ncandidates* and Lemma 25, it is clear that p eventually executes forever in

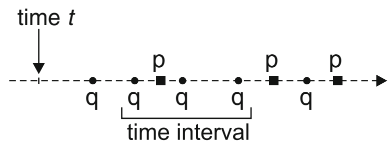

Fig. 7 After time t, p takes at least one step every 3 steps of process q

the empty loop of line 42. Note that just before entering this loop, p sets LEADER to ? in line 41.

Corollary 6 *For every process $p \in Ncandidates$, there is a time after which* $\text{LEADER}_p = ?$.

Proof Clear from Lemmas 26 and 27.

By the above corollary, Property (2) of Ω_Δ is satisfied in run R of the algorithm. We now proceed to show that Property (1) of Ω_Δ is also satisfied in run R.

Definition 13 Let $Timely = \{q : q$ is timely in run $R\}$.

If $Pcandidates \cap Timely = \emptyset$, then Property (1) of Ω_Δ is trivially satisfied. Henceforth (from Lemmas 28 to 49) we assume that

Assumption 9 $Pcandidates \cap Timely \neq \emptyset$

and show that Property (1) of Ω_Δ is also satisfied in this case.

Lemma 28 $pcandidates \cap Timely \neq \emptyset$.

Proof Clear from Assumption 9 and Lemma 26.

Definition 14 We say that *"process p does X every k steps of process q"* if p does X during any time interval that contains k steps of process q.

Similarly, we define the following:

Definition 15 We say that *"after time t, process p does X every k steps of process q"* if p does X during any time interval that starts after time t and that contains k steps of process q.

For example, when we say "after time t, p takes at least one step every 3 steps of process q" we mean that, in any time interval after time t containing 3 steps of q, p takes at least one step, as illustrated in Fig. 7.

Lemma 29 *There exists an integer C_0 such that, for every process $p \in pcandidates \cap Timely$ and every process q, p takes at least one step every $C_0 + 1$ steps of q.*

Proof For every process $p \in pcandidates \cap Timely$ and every process q, p is q-timely so there is an integer i_{pq} such that every time interval containing i_{pq} steps of q has at least one step of p. Let $C_0 = \max\{i_{pq} : p \in pcandidates \cap Timely$ and $q \in \Pi\}$.

Consider a process $p \in pcandidates \cap Timely$ and a process q. Any time interval with $C_0 + 1$ steps of q includes at least $i_{pq} + 1$ steps of q and hence a step of p.

Definition 16 Let C_0 be the integer from Lemma 29.

Corollary 7 *For every process $p \in pcandidates \cap Timely$, every process q, and every integer $k \geq 1$, p takes at least k steps every $kC_0 + 1$ steps of q.*

Proof Clear from Lemma 29.

Definition 17 For processes p and r, we say that *p writes a message successfully to r at time t* if, at time t,

- p executes in line 3 with $q = r$ and the if guard evaluates to *false*, or
- p receives a response *ok* from the write to $MsgRegister[p, q]$ with $q = r$ in line 5.

Intuitively, p writes a message successfully to r if either the value it wants to write to $MsgRegister[p, r]$ has already been written previously (the guard in line 3 evaluates to *false*) or p actually writes the value to $MsgRegister[p, r]$ and the write returns an *ok* response.

The body of the do-while loop in lines 44–57 has no unbounded loops. Therefore, we can define the following:

Definition 18 Let C_M and C_m be the maximum and minimum, respectively, number of steps to execute one complete iteration of the do-while loop in lines 44–57.

Note that the values of C_M and C_m depend on the code alone, and not on how fast or slow a process executes the code.

Definition 19 Let T_0 be the time after which processes in *pcandidates* never exit the do-while loop in lines 44–57.

We now generalize Definition 14 for properties that hold during a time interval:

Definition 20 In the following, we say "*during times* $[t, t']$, *process p does X every k steps of process q*" if p does X during any time interval that is contained in $[t, t']$ and that contains k steps of process q.

The next lemma and corollary state sufficient conditions for a process p to periodically write a message successfully to a process q.

Lemma 30 *For all processes $p \neq q$, if $p \in pcandidates \cap Timely$ and $q \in infcandidates$ then there exists an integer c and a time $t > T_0$ such that, after time t, p writes a message successfully to q at least once every $c + 1$ steps of q.*

Proof Consider two processes $p \neq q$ and suppose that $p \in pcandidates \cap Timely$ and $q \in infcandidates$. Let $\alpha = \lceil (C_0 C_M + 1)/C_m \rceil$.

Claim 1 *After time T_0, if q executes $2\alpha C_m$ steps without reading variable MsgRegister$[p, q]$ then p writes a message successfully to q at some time between the first and last of those $2\alpha C_m$ steps of q.*

To show Claim 1, suppose that some time after T_0, q executes $2\alpha C_m$ steps without reading variable *MsgRegister*$[p, q]$. During such steps of q, by Corollary 7, p executes at least $\lfloor(2\alpha C_m - 1)/C_0\rfloor$ steps. Since $\lfloor(2\alpha C_m - 1)/C_0\rfloor \geq 2C_M$, during those steps p executes procedure *WriteMsgs* in its entirety at least once. In this procedure, when p executes line 3 for q, if the guard evaluates to *false* then p writes a message successfully to q by definition. Otherwise, p writes to *MsgRegister*$[p, q]$ in line 5. Since q does not read variable *MsgRegister*$[p, q]$ during those steps of p, by the non-triviality property of abortable registers the write returns *ok*. Thus, p writes a message successfully to q by definition. This shows Claim 1.

Let $c = 12(\alpha + 1)\alpha C_m$ and $t = T_0 + 1$. To prove the lemma, we now show that after time t, p writes a message successfully to q at least once every $c + 1$ steps of q.

Suppose, by contradiction, that for some $t' > t$, starting at time t', q takes $c + 1$ steps without p writing a message successfully to q. Let t'' be the time when q takes the last of those $c + 1$ steps.

Claim 2 *During the time interval $[t', t'']$, there is at most one value that p can write to MsgRegister$[p, q]$.*

To show Claim 2, note that if p never writes to *MsgRegister*$[p, q]$ during $[t', t'']$ then the claim holds vacuously. Now, suppose that p writes to *MsgRegister*$[p, q]$ during $[t', t'']$. Consider the first such a write, and let v be the value being written. Then, $v = msgCurr_p[q]$ at the time the write occurs. Neither this first write nor any subsequent writes to *MsgRegister*$[p, q]$ until time t'' return *ok* since p does not write a message successfully to q during $[t', t'']$. Therefore, after the first write, $prevWriteDone_p[q]$ is set to *false* in line 6 and then it is never set to *true* before time t''. Thus, after the first write until time t'', p does not change $msgCurr_p[q]$ because of the guard in line 4. Thus, any subsequent writes to *MsgRegister*$[p, q]$ until time t'' are for value v. This shows Claim 2.

Claim 3 *During times $[t', t'']$, q finds that $res_q[p] \neq \perp$ and $res_q[p] \neq prevMsgFrom[p]$ in line 14 every $4(\alpha + 1)\alpha C_m$ steps of q.*[10]

To show Claim 3, consider any time interval $[u', u'']$ contained in $[t', t'']$ in which q executes $4(\alpha+1)\alpha C_m$ steps. From Claim 1 and the fact that p does not write a message successfully to q during times $[t', t'']$, we know that (*) during times $[t', t'']$, q reads *MsgRegister*$[p, q]$ at least once every $2\alpha C_m$ steps of q. (Note that this read occurs in line 13.) Therefore, during $[u', u''], q$ reads *MsgRegister*$[p, q]$ at least $2(\alpha + 1) = 2\alpha + 2$ times, storing the result in $res_q[p]$. We now prove that at least once in the first 2α times that this happens, $res_q[p] \neq \perp$ and $res_q[p] \neq$ *prevMsgFrom*$[p]$ (this implies Claim 3). Suppose, by contradiction, that in the first 2α times during $[u', u'']$ when q reads *MsgRegister*$[p, q]$ in line 13, the result $res_q[p]$ satisfies $res_q[p] = \perp$ or $res_q[p] = prevMsgFrom[p]$. Then, by the guard in line 14, q increments $readTimeout_q[p]$ in line 15 at least 2α times without resetting it to 1 in line 18. Clearly, $readTimeout_q[p]$ is always a positive integer. Therefore, after being incremented 2α times, $readTimeout_q[p]$ is set to at least $2\alpha + 1$. Thus, the next time when q reads *MsgRegister*$[p, q]$ in line 13, $readTimer_q[p] \geq 2\alpha + 1$ because of the assignment in line 12. Subsequently, by the way $readTimer_q[p]$ works, q executes at least 2α complete iterations of the do-while loop in lines 44–57 without reading *MsgRegister*$[p, q]$, and this happens before the $(2\alpha + 2)$-th reading of *MsgRegister*$[p, q]$ during $[u', u'']$. Since each loop iteration takes at least C_m steps, q takes $2\alpha C_m$ steps without reading *MsgRegister*$[p, q]$. This contradicts (*) and shows Claim 3.

Since q executes $12(\alpha+1)\alpha C_m$ steps during $[t', t'']$, from Claim 3, there are at least three times during $[t', t'']$ when q finds that (**) $res_q[p] \neq \perp$ and $res_q[p] \neq prevMsgFrom[p]$ in line 14. Consider the first three such times and let r_j be the value of $res_q[p]$ in the j-th time, for $j = 1, 2, 3$. Then, from (**), $r_j \neq \perp$ for $j = 1, 2, 3$. Moreover, from (**) and the fact that p sets $prevMsgFrom_q[p]$ to r_j in line 17, we have that $r_1 \neq r_2$ and $r_2 \neq r_3$.

Note that r_j is the value returned by the read of *MsgRegister*$[p, q]$ in line 13, for $j=1, 2, 3$. By Claim 2, there is at most one value that p can write to *MsgRegister*$[p, q]$ during $[t', t'']$. Therefore, by linearizability of abortable registers, it is not possible for the non-$\perp$ values read from *MsgRegister*$[p, q]$ to change more than once. Thus, either $r_1 = r_2$ or $r_2 = r_3$. This contradicts the fact that $r_1 \neq r_2$ and $r_2 \neq r_3$.

Corollary 8 *There exists an integer C_1 and a time $T_1 > T_0$ such that, for all processes $p \neq q$ such that $p \in$ pcandidates $\cap$ Timely and $q \in$ infcandidates, after time T_1, p writes a message successfully to q at least once every $C_1 + 1$ steps of q.*

Proof Immediate from Lemma 30 and the fact that the system has only finitely many processes.

Definition 21 Let C_1 and T_1 be the integer and time from Corollary 8.

[10] Recall from Definition 20 the meaning of the statement "during times $[t', t'']$, process p does X every k steps of q".

We now show that if p periodically writes messages successfully to q and p's message to q stops changing, then q eventually sees the message provided that $q \in infcandidates$.

Lemma 31 *For all processes $p \neq q$, if*

(a) *p writes a message successfully to q infinitely often,*
(b) *there is a value v and a time after which $msgTo_p[q] = v$, and*
(c) *$q \in infcandidates$*

then there is a time after which $msgFrom_q[p] = v$.

Proof Consider two processes $p \neq q$, and suppose that p writes a message successfully to q infinitely often, there is a value v and a time after which $msgTo_p[q] = v$, and $q \in infcandidates$. Let t_1 be the time after which $msgTo_p[q] = v$.

Claim 1 *There is a time $t_2 > t_1$ after which $msgCurr_p[q] = msgTo_p[q]$.*

Suppose, by contradiction, that $msgCurr_p[q] \neq msgTo_p[q]$ infinitely often. The only place where $msgCurr_p[q]$ changes is in line 4, where it is set to $msgTo_p[q]$. Thus, since after time t_1 $msgTo_p[q] = v$ and $msgCurr_p[q] \neq msgTo_p[q]$ infinitely often, there is a time after which p does not set $msgCurr_p[q]$ in line 4. Thus, (*) there is a time t'_1 after which $msgCurr_p[q]$ does not change and $msgCurr_p[q] \neq msgTo_p[q]$. After time t'_1, every time q executes line 3, the if guard evaluates to *true*. Since p writes messages successfully to q infinitely often, there is some time after $\max\{t_1, t'_1\}$ when p writes a message to q successfully. At such a time, by Definition 17, p executes line 5 and receives an *ok* response from the write to $MsgRegister[p, q]$. After doing so, p sets $prevWriteDone_p[q]$ to *true*. Since $prevWriteDone_p[q]$ can change only in line 6, the next time p executes line 3, its guard evaluates to true by (*) and $prevWriteDone_p[q]$ is still *true*. Thus p executes line 4 and sets $msgCurr_p[q]$ to $msgTo_p[q]$. This contradicts (*) and shows Claim 1.

Claim 2 *There is a time after which p never writes to $MsgRegister[p, q]$ in line 5.*

Suppose, by contradiction, that p writes to $MsgRegister[p, q]$ in line 5 infinitely often. Then, (**) p executes line 3 infinitely often with the if guard evaluating to *true*. Let t'_2 be some time after t_2 when p executes line 3 and the guard evaluates to *true*. Then, by Claim 1, at time t'_2 we have that $prevWriteDone_p[q] = false$. After time t'_2, if p ever gets an *ok* response from the write to $MsgRegister[p, q]$ in line 5 then p sets $prevWriteDone_p[q]$ to *true* in line 6 and, because $prevWriteDone_p[q]$ is not changed anywhere else, in every subsequent execution of line 3, the guard evaluates to *false* and therefore $prevWriteDone_p[q]$ remains *true* forever, and this contradicts (**). Therefore, (***) after time t'_2, every time p executes line 5, p gets a $\perp$ response from the write to $MsgRegister[p, q]$. Since p writes a message successfully to q infinitely often, it does so at some time $t''_2 > t'_2$. At time t''_2, from (***) and Definition 17, p executes in line 3 and the guard evaluates to *false*. Therefore, $prevWriteDone_p[q] = true$ and, by the same argument above, $prevWriteDone_p[q]$ remains *true* forever after, which contradicts (**). This shows Claim 2.

Claim 2 implies that (a) eventually $MsgRegister[p, q]$ stops changing, (b) p eventually stops changing $prevWriteDone_p[q]$ (since line 6 is the only place where this happens), and (c) the final value of $prevWriteDone_p[q]$ is *true* (otherwise p keeps writing to $MsgRegister[p, q]$ by the guard in line 3). Thus, at the last time that p sets $prevWriteDone_p[q]$ (which could be on initialization), p sets it to *true*, and so $MsgRegister[p, q] = msgCurr_p[q]$. Moreover, at this time, $msgCurr_p[q] = v$ (otherwise, subsequently p finds that $msgCurr_p[q] \neq msgTo_p[q]$ in line 3 and sets $prevWriteDone_p[q]$ again). Thus, there is a time $t_3 > t_2$ after which $MsgRegister[p, q] = v$.

From Claim 2, there is a time $t_4 > t_3$ after which p does not access $MsgRegister[p, q]$. Since $q \in infcandidates$, eventually q tries to read $MsgRegister[p, q]$ in line 13 after time t_4. When this happens, the read does not abort and it returns v. Thus, q sets $prevMsgFrom_q[p]$ to v (if it is not set to that value already). Subsequently, any time q tries to read $MsgRegister[p, q]$, the read returns v. Thus, there is a time after which $prevMsgFrom_q[p] = v$.

The next lemma states sufficient conditions for a process p to periodically write to its two heartbeat registers that are read by process q.

Lemma 32 *There exists an integer C_2 and a time $T_2 > T_0$ such that, for every processes $p \neq q$, if $p \in pcandidates \cap Timely$ and $q \in infcandidates$, then after time T_2, p writes to $HbRegister1[p, q]$ and $HbRegister2[p, q]$ in lines 24–25 at least once every $C_2 + 1$ steps of q.*

Proof Let $C_2 = C_1 + C_0(C_M + 1)$ and $T_2 = T_1$.

Consider two processes $p \neq q$ such that $p \in pcandidates \cap Timely$ and $q \in infcandidates$. Let t_1 be some time after T_2. By Corollary 8, starting at time t_1, q takes at most C_1 steps before p writes a message successfully to q. Let t_2 be the first time after t_1 when this happens. Next, q takes at most $C_0(C_M + 1)$ steps before p has executed C_M steps (by Corollary 7). We now consider what p does during those C_M steps. After writing a message successfully to q, p returns in line 7 with $prevWriteDone_p[q] = true$. Thus, p sets *writeDone* in line 52 so that $writeDone_p[q] = true$. Next, p executes *SendHeartbeat*(*dest*) with $dest[q] = true$. Inside this procedure, p writes to $HbRegister1[p, q]$ and $HbRegister2[p, q]$ in lines 24–25.

Definition 22 Let C_2 and T_2 be the integer and time from Lemma 32.

Definition 23 We say that a process p *times out on a process q at time t* if p removes q from $activeSet_p$ in line 38 at time t.

We now give sufficient conditions for a process q not to timeout on a process p.

Lemma 33 *For every process $p \in pcandidates \cap Timely$ and every process $q \neq p$, there is a time after which q does not time out on p.*

Proof Suppose, by contradiction, that there is a timely process $p \in pcandidates \cap Timely$ and a process q such that q times out on p infinitely often. Then $hbTimeout_q[p]$ grows without bound (because q increments $hbTimeout_q[p]$ right after q times out on p, and $hbTimeout_q[p]$ is monotonically nondecreasing).

Since $p \in pcandidates \cap Timely$, by Lemma 32, (*) after time T_2, p writes to $HbRegister1[p, q]$ and $HbRegister2[p, q]$ in lines 24–25 at least once every $C_2 + 1$ steps of q.

From the code in Fig. 5, q repeats the following cycle: (1) it sets $hbTimer_q[p]$ to $hbTimeout_q[p]$, and (2) q executes $hbTimeout_q[p]$ iterations of the do-while loop in lines 44–57 until $hbTimer_q[p]$ reaches 0, and (3) q executes line 30. From the time (1) occurs to the time (3) occurs, q does not read $HbRegister1[p, q]$. Thus, since $hbTimeout_q[p]$ grows without bound, there is a time t when $hbTimeout_q[p]$ reaches a large enough value so that, after t, q invokes a read operation on $HbRegister1[p, q]$ (in line 33) at most once every $C_2 + 5$ steps of q.

Consider any time $t' > \max\{T_2, t\}$ when q invokes a read operation on $HbRegister1[p, q]$. In its next 3 steps, q gets a response for the read, invokes a read operation on $HbRegister2[p, q]$, and gets a response. Subsequently, q executes at least $C_2 + 1$ steps without invoking a read on $HbRegister1[p, q]$ again (since $t' > t$). From (*), while q executes those steps, p writes to $HbRegister1[p, q]$ and to $HbRegister2[p, q]$ at least once. Moreover, when either of these writes happen, p is the only process accessing $HbRegister1[p, q]$ or $HbRegister2[p, q]$ (since the only processes that access this register are p and q). Thus, neither write of p aborts, and so they take effect, causing the values of $HbRegister1[p, q]$ and $HbRegister2[p, q]$ to increase. The next time $hbTimer_q[p]$ reaches 0, q reads $HbRegister1[p, q]$ and $HbRegister2[p, q]$ again. For each of these, either q reads $\perp$ or it reads a value different from what it read before. Therefore, the guard in line 35 evaluates to *true*, and so q does not timeout on p.

Thus, we have shown that if q reads $HbRegister1[p, q]$ at a time $t' > \max\{T_2, t\}$ then the next time $hbTimer_q[p]$ reaches 0, q does not timeout on p. Therefore, there is a time after which q never times out on p—a contradiction.

Since $activeSet_p$ is initialized to $\{p\}$ and p never removes itself from $activeSet_p$, we have the following:

Observation 10 For every process p, $p \in activeSet_p$.

Lemma 34 *For every process $p \in pcandidates \cap Timely$ and every process $q \in infcandidates$, there is a time after which $p \in activeSet_q$.*

Proof Let $p \in pcandidates \cap Timely$ and $q \in infcandidates$. If $p = q$ then the result follows from Observation 10. So assume $p \neq q$. Process q calls procedure *ReceiveHeartbeat* infinitely many times. In each execution of this procedure, q decrements $hbTimer_q[p]$ by one, until it reaches 0. When it reaches 0, q resets $hbTimer_q[p]$ to $hbTimeout_q[p]$ and executes the if statement in line 35. This happens infinitely many times. The if statement results in either q adding p to $activeSet_q$ in line 36 or q removing p from $activeSet_q$ in line 38. By Lemma 33, there is a time t after which q does not time out on p. Therefore, q adds p to $activeSet_q$ infinitely often and there is a time after which q does not remove p from $activeSet_q$. Thus, there is a time after which $p \in activeSet_q$.

Observation 11 For every process p, $counter_p[p]$ is monotonically nondecreasing with time.

Lemma 35 *For all processes $p \neq q$, if there is time after which $actrTo_q[p]$ stops changing then there is a time after which $actrFrom_p[q]$ stops changing.*

Proof Consider two processes $p \neq q$, and assume that there is time after which $actrTo_q[p]$ stops changing. Since $msgTo_q[p]$ can be set only to $\langle counter_q[q], actrTo_q[p]\rangle$ (in line 51), there is a time after which the second component of $msgTo_q[p]$ stops changing. Since $msgCurr_q[p]$ can be set only to $msgTo_q[p]$ (in line 4), there is a time after which the second component of $msgCurr_q[p]$ stops changing. Since $MsgRegister[q, p]$ is linearizable and it can be written only with the value of $msgCurr_q[p]$ (in line 5) there is a time after which the non-$\perp$ values read from $MsgRegister[q, p]$ (in line 13) always have the same second component. Since $prevMsgFrom_p[q]$ can be set only to a value read from $MsgRegister[q, p]$ (in line 17), there is a time after which the second component of $prevMsgFrom_p[q]$ stops changing. Since $msgFrom_p$ can be set only to a value returned from procedure *WriteMsgs* (in line 53), and this procedure returns the value of $prevMsgFrom_p$, there is a time after which the second component of $msgFrom_p[q]$ stops changing. Since $actrFrom_p[q]$ can be set only to the second component of $msgFrom_p[q]$ (in line 55), there is a time after which $actrFrom_p[q]$ stops changing.

We now show that the counter of a process in *pcandidates* ∩ *Timely* eventually stops changing. We later extend this result to show that the counter of *every* process eventually stops changing.

Lemma 36 *For every process* $p \in pcandidates \cap Timely$, *there exists an integer* c_p *and a time after which* $counter_p[p] = c_p$.

Proof Let $p \in pcandidates \cap Timely$.

Claim *For every process* $q \neq p$, *there is a time after which* $actrTo_q[p]$ *stops changing.*

Consider a process $q \neq p$. If $q \notin infcandidates$ then there is a time after which p does not execute the body of the do-while loop in lines 44–57, and so eventually $actrTo_q[p]$ stops changing. If $q \in infcandidates$ then, by Lemma 34, there is a time after which $p \in activeSet_q$. The claim now follows since $actrTo_q[p]$ can be changed only in line 50, and only if $p \notin activeSet_q$.

The only places where p changes $counter_p[p]$ is in lines 43 or 56. Since $p \in pcandidates$, there is a time after which p does not execute line 43. In line 56, p sets $counter_p[p]$ to $\max\{counter_p[p], actrFrom_p[q]\}$ for some $q \neq p$. By the claim and Lemma 35, for every process $q \neq p$, there is a time after which $actrFrom_p[q]$ stops changing. Thus, for every process q, there is a time after which line 56 does not change $counter_p[p]$. Thus, there is a time after which $counter_p[p]$ does not change.

Recall that by Lemma 28, $pcandidates \cap Timely \neq \emptyset$. A process in this set intersection enjoys some strong properties on its interactions with other processes, as we showed in previous lemmas. We now pick an arbitrary process in this set intersection and use it to prove properties about other processes.

Definition 24 Let s be some fixed process in $pcandidates \cap Timely$.

Note that, by Lemma 36, there exists an integer c_s and a time after which $counter_s[s] = c_s$.

Lemma 37 *For every process* $p \in infcandidates$, *there is a time after which* $counter_p[leader_p] \leq c_s$.

Proof Let $p \in infcandidates$. Since $s \in pcandidates \cap Timely$, by Lemma 34, there is a time t_1 after which $s \in activeSet_p$. Since $p \in infcandidates$, p executes line 47 infinitely often. Let t_2 be the first time after t_1 when p sets $leader_p$ in line 47. Then, from time t_2 onwards, $counter_p[leader_p] \leq counter_p[s]$, since p chooses $leader_p$ in line 47 as the process q in $activeSet_p$ with the smallest $(counter_p[q], q)$. Moreover, at any given time, $counter_p[s] \leq counter_s[s]$ since values of $counter_p[s]$ come from the first component of $msgFrom_p[s]$, which come from the first component of $MsgRegister[s, p]$, which come from the first component of $msgTo_s[p]$, which come from $counter_s[s]$ in line 51. At any time, $counter_s[s] \leq c_s$, by definition of c_s and the fact that $counter_s[s]$ is monotonically nondecreasing (Observation 11). Thus, after time t_2, $counter_p[leader_p] \leq counter_p[s] \leq counter_s[s] \leq c_s$.

From the way p modifies $actrTo_p[q]$ in the algorithm (in line 50), it is clear that:

Observation 12 For all processes $p \neq q$, $actrTo_p[q]$ is monotonically nondecreasing with time.

We show that the accusation counter of q at $p \neq q$ eventually stops changing:

Lemma 38 *For all processes* $p \neq q$, *there exists an integer* a_{pq} *and a time after which* $actrTo_p[q] = a_{pq}$.

Proof Consider two processes $p \neq q$. The only place where p sets $actrTo_p[q]$ is in line 50. If $p \notin infcandidates$ then there is a time after which p does not execute the do-while loop in lines 44–57. Thus, there is a time after which $actrTo_p[q]$ does not change, and the lemma follows.

Now assume $p \in infcandidates$. When p changes $actrTo_p[q]$, it changes it to $\max\{actrTo_p[q], counter_p[leader_p]\}$ in line 50. By Lemma 37, there is a time after which $counter_p[leader_p] \leq c_s$. Therefore, there is a time after which $actrTo_p[q]$ does not change, and the lemma follows.

We now extend Lemma 36 to show that the counter of every process eventually stops changing.

Lemma 39 *For every process* p, *there exists an integer* c_p *and a time after which* $counter_p[p] = c_p$.

Proof Let p be a process. The only places where p changes $counter_p[p]$ are in lines 43 or 56.

If $p \notin infcandidates$ then there is a time after which p does not execute either of these lines. Thus, there is a time after which $counter_p[p]$ does not change, and the lemma follows.

Now assume $p \in infcandidates$. By Observation 11, $counter_p[p]$ is monotonically nondecreasing. By Lemma 37, there is a time after which $counter_p[leader_p] \leq c_s$. Thus, there is a time after which line 43 does not increase $counter_p[p]$. By Lemma 38, for every process $q \neq p$, there is a time after which $actrTo_q[p]$ stops changing. By Lemma 35, for every process $q \neq p$, there is a time after which $actrFrom_p[q]$ stops changing. Thus, there is a time after which line 56 does not increase $counter_p[p]$. Thus, there is a time after which $counter_p[p]$ does not change, and the lemma follows.

Definition 25 For all processes $p \neq q$, let a_{pq} and c_p be the integers from Lemmas 38 and 39, respectively.

The next lemma states that if $p \in \mathit{infcandidates}$ then the message p writes to another process q eventually stops changing and remains equal to $\langle c_p, a_{pq} \rangle$.

Lemma 40 *For all processes $p \neq q$, if $p \in \mathit{infcandidates}$ then there is a time after which $\mathit{msgTo}_p[q] = \langle c_p, a_{pq} \rangle$.*

Proof Consider two processes $p \neq q$ such that $p \in \mathit{infcandidates}$. The only place where p sets $\mathit{msgTo}_p[q]$ is in line 51. Since $p \in \mathit{infcandidates}$, p executes this line infinitely many times. In this line, p changes $\mathit{msgTo}_p[q]$ to $\langle \mathit{counter}_p[p], \mathit{actrTo}_p[q] \rangle$. By Lemma 39, there is a time after which $\mathit{counter}_p[p] = c_p$. By Lemma 38, there is a time after which $\mathit{actrTo}_p[q] = a_{pq}$. So, there is a time after which $\mathit{msgTo}_p[q] = \langle c_p, a_{pq} \rangle$.

We now give sufficient conditions for p to write its message successfully to q.

Lemma 41 *For all processes $p \neq q$, if $p \in \mathit{activeSet}_q$ infinitely often and $q \in \mathit{infcandidates}$ then p writes a message successfully to q infinitely often.*

Proof Consider two processes $p \neq q$ such that $p \in \mathit{activeSet}_q$ infinitely often and $q \in \mathit{infcandidates}$. Suppose, by contradiction, that p writes a message successfully to q only finitely often. We claim that p writes to $\mathit{HbRegister1}[p, q]$ and $\mathit{HbRegister2}[p, q]$ in lines 24 and 25 only finitely often. Indeed, if $p \notin \mathit{infcandidates}$ then p executes lines 24 and 25 only finitely often. Now suppose $p \in \mathit{infcandidates}$. Then p executes procedure *WriteMsgs* infinitely often. Thus, since p writes a message successfully to q only finitely often, there is a time after which $\mathit{prevWriteDone}_p[q] = \mathit{false}$, and so there is a time after which $\mathit{writeDone}_p[q] = \mathit{false}$. Since p always calls procedure *SendHeartbeat* with parameter $\mathit{dest} = \mathit{writeDone}$, by the guard in line 23, p writes to $\mathit{HbRegister1}[p, q]$ and $\mathit{HbRegister2}[p, q]$ in lines 24 and 25 only finitely often. This shows the claim.

Since $q \in \mathit{infcandidates}, q$ calls procedure *ReceiveHeartbeat* infinitely many times. By the code, q infinitely often finds that $\mathit{hbTimer}_q[p] = 0$ in line 29 and executes the reads in line 33–34. Since there are only finitely many writes to $\mathit{HbRegister1}[p, q]$ and to $\mathit{HbRegister2}[p, q]$ there is a time after which every read on $\mathit{HbRegister1}[p, q]$ returns the same non-$\perp$ value v_1, and there is a time after which every read on $\mathit{HbRegister2}[p, q]$ returns the same non-$\perp$ value v_2. Thus, there is a time after which $\mathit{prevHbCounter1}_q[p] = \mathit{hbCounter1}_q[p] = v_1 \neq \perp$ and $\mathit{prevHbCounter2}_q[p] = \mathit{hbCounter2}_q[p] = v_2 \neq \perp$. Thus, q adds p to $\mathit{activeSet}_q$ in line 36 only finitely many times, and q removes p from $\mathit{activeSet}_q$ in line 38 infinitely many times. Thus, there is a time after which $p \notin \mathit{activeSet}_q$. This contradicts the fact that $p \in \mathit{activeSet}_q$ infinitely often.

The next lemma and corollary give conditions for a process q to learn about the counter and accusation counter that process p writes.

Lemma 42 *For all processes $p \neq q$, if $p \in \mathit{activeSet}_q$ infinitely often and $q \in \mathit{infcandidates}$ then there is a time after which (a) $\mathit{counter}_q[p] = c_p$, (b) $\mathit{actrFrom}_q[p] = a_{pq}$, and (c) $\mathit{counter}_q[q] \geq a_{pq}$.*

Proof Consider two processes $p \neq q$ such that $p \in \mathit{activeSet}_q$ infinitely often and $q \in \mathit{infcandidates}$. By Lemma 41, p writes a message successfully to q infinitely often, and so $p \in \mathit{infcandidates}$. By Lemma 40, there is a time after which $\mathit{msgTo}_p[q] = \langle c_p, a_{pq} \rangle$. Therefore, by Lemma 31, (*) there is a time after which $\mathit{msgFrom}_q[p] = \langle c_p, a_{pq} \rangle$.

Since $q \in \mathit{infcandidates}$, q executes lines 55 and 56 infinitely often. In line 55, q sets $\langle \mathit{counter}_q[q], \mathit{actrFrom}_q[p] \rangle$ to $\mathit{msgFrom}_q[p]$. Thus, there is a time after which (a) $\mathit{counter}_q[p] = c_p$ and (b) $\mathit{actrFrom}_q[p] = a_{pq}$. Moreover, by the way q sets $\mathit{counter}_q[q]$ in line 56, and since $\mathit{counter}_q[q]$ is monotonically non-decreasing (Observation 11), there is a time after which (c) $\mathit{counter}_q[q] \geq a_{pq}$.

Corollary 9 *For all processes $p \neq q$, if $p \in \mathit{pcandidates} \cap \mathit{Timely}$ and $q \in \mathit{infcandidates}$ then there is a time after which (a) $\mathit{counter}_q[p] = c_p$, (b) $\mathit{actrFrom}_q[p] = a_{pq}$, and (c) $\mathit{counter}_q[q] \geq a_{pq}$.*

Proof Consider two processes $p \neq q$ such that $p \in \mathit{pcandidates} \cap \mathit{Timely}$ and $q \in \mathit{infcandidates}$. By Lemma 34, there is a time after which $p \in \mathit{activeSet}_q$. The corollary now follows from Lemma 42.

Intuitively, a process q should not think that a process in *ncandidates* is active. Indeed, this holds if q is in *infcandidates*:

Lemma 43 *For every process $q \in \mathit{infcandidates}$, there is a time after which $\mathit{activeSet}_q \subseteq \mathit{infcandidates}$.*

Proof Consider a process $q \in \mathit{infcandidates}$. Since there are only finitely many processes, there is a time after which $\mathit{activeSet}_q$ contains only processes that are in $\mathit{activeSet}_q$ infinitely often. Suppose $p \in \mathit{activeSet}_q$ infinitely often. If $p = q$ then $p \in \mathit{infcandidates}$ since $q \in \mathit{infcandidates}$. If $p \neq q$, then by Lemma 41, p writes a message successfully to q infinitely often, and so $p \in \mathit{infcandidates}$.

We now define ℓ as the process p in *pcandidates* with smallest c_p, breaking ties using the process id. Note that ℓ is well defined because, by Lemma 28, the set *pcandidates* is not empty.

Definition 26 Let ℓ be the process in *pcandidates* such that $(c_\ell, \ell) = \min\{(c_p, p) : p \in \mathit{pcandidates}\}$.

The next two lemmas show that not only ℓ is the process in *pcandidates* with smallest counter; ℓ is also the process in *infcandidates* with smallest counter.

Lemma 44 *For every process* $p \in \mathit{infcandidates} - \mathit{pcandidates}$, $(c_\ell, \ell) < (c_p, p)$.

Proof Suppose, by contradiction, there is a process in $p \in \mathit{infcandidates} - \mathit{pcandidates}$ such that $(c_p, p) \leq (c_\ell, \ell)$. Let p be such a process with smallest (c_p, p). Then, by definition of ℓ and the fact that $(c_p, p) \leq (c_\ell, \ell)$, p is the process in *infcandidates* with smallest (c_p, p).

By Lemmas 43 and 42, we can find a time t after which (a) $\mathit{activeSet}_p$ contains only processes in *infcandidates*, and (b) for every process $q \in \mathit{activeSet}_p$, $\mathit{counter}_p[q] = c_q$. Since $p \in \mathit{infcandidates}$, p sets leader_p in line 47 infinitely many times. After time t, whenever p sets leader_p after time t in line 47, p sets leader_p to p (this is because p is the process with smallest (c_p, p) in *infcandidates* and $p \in \mathit{activeSet}_p$). Thus, there is a time $t' > t$ after which $\mathit{leader}_p = p$ and $\mathit{counter}_p[p] = c_p$.

Since $p \in \mathit{infcandidates} - \mathit{pcandidates}$, p sets $\mathit{counter}_p[p]$ in line 43 infinitely many times. When p does so after time t', p sets $\mathit{counter}_p[p]$ to $c_p + 1$, a contradiction to the fact that $\mathit{counter}_p[p] = c_p$ after time t'.

We now show that ℓ is the process in *infcandidates* with smallest c_p, breaking ties using the process id.

Lemma 45 $(c_\ell, \ell) = \min\{(c_p, p) : p \in \mathit{infcandidates}\}$.

Proof Let $p \in \mathit{infcandidates}$. If $p \in \mathit{pcandidates}$ then $(c_\ell, \ell) \leq (c_p, p)$ by definition of ℓ. If $p \in \mathit{infcandidates} - \mathit{pcandidates}$ then $(c_\ell, \ell) < (c_p, p)$ by Lemma 44.

Recall that s is some fixed process in $\mathit{pcandidates} \cap \mathit{Timely}$ (see Definition 24). In the next two lemmas and the following corollary, we use s to show properties about ℓ.

Lemma 46 *There is a time after which* $\ell \in \mathit{activeSet}_s$.

Proof Suppose, by contradiction, that $\ell \notin \mathit{activeSet}_s$ infinitely often. Then $\ell \neq s$. Moreover, (*) infinitely often s sets $\mathit{activeSet}_s$ in line 46 to a set that does not contain ℓ. By Lemmas 43 and 42, we can find a time t after which (a) $\mathit{activeSet}_s$ contains only processes in *infcandidates*, and (b) for every process $q \in \mathit{activeSet}_s$, $\mathit{counter}_s[q] = c_q$. By (*), we can find a time $t' > t$ when s sets $\mathit{activeSet}_s$ in line 46 to a set that does not contain ℓ. Then, s sets leader_s to some process $q \neq \ell$ in line 47. Moreover, by (a), $q \in \mathit{infcandidates}$. Therefore, by Lemma 45, $(c_\ell, \ell) < (c_q, q)$. Thus, $c_q \geq c_\ell$.

Then, s finds that $\ell \notin \mathit{activeSet}_s$ in line 50 and s sets $\mathit{actrTo}_s[\ell]$ to a value $a \geq \mathit{counter}_s[\mathit{leader}_s] + 1$. But $\mathit{leader}_s = q$ and $\mathit{counter}_s[q] = c_q$ by (b). So $a \geq c_q + 1 \geq c_\ell + 1$.

Since $\mathit{actrTo}_s[\ell]$ is monotonically nondecreasing (Observation 12), there is a time after which $\mathit{actrTo}_s[\ell] \geq c_\ell + 1$. Thus, by the definition of $a_{s\ell}$ (Definition 25), $a_{s\ell} \geq c_\ell + 1$.

By definition, $s \in \mathit{pcandidates} \cap \mathit{Timely}$ and $\ell \in \mathit{pcandidates} \subseteq \mathit{infcandidates}$. So by Corollary 9(c), there is a time after which $\mathit{counter}_\ell[\ell] \geq a_{s\ell}$. But $a_{s\ell} \geq c_\ell + 1$, so there is a time after which $\mathit{counter}_\ell[\ell] \geq c_\ell + 1$, which contradicts the definition of c_ℓ (Definition 25).

Lemma 47 $\ell \in \mathit{Timely}$.

Proof Suppose, by contradiction, that $\ell \notin \mathit{Timely}$. From Lemma 46, s removes ℓ from $\mathit{activeSet}_s$ only finitely many times (in line 38). So, (*) there is a time t after which s does not increase $\mathit{hbTimeout}_s[\ell]$ (in line 39). Since $\mathit{hbTimeout}_s[\ell]$ is monotonically nondecreasing, there exists an integer h such that, after time t, $\mathit{hbTimeout}_s[\ell] = h$. Let x_0 be the number of steps of s up to time t.

Since $s \in \mathit{Timely}$ and $\ell \notin \mathit{Timely}$, by Corollary 2, ℓ is not s-timely. So, for every integer i there is a time interval that has i steps of s but no steps of ℓ. In particular, there is a time interval that has $x_0 + (2h + 2)C_M$ steps of s but no steps of ℓ. Thus, we can find a time interval I after time t that has $(2h + 2)C_M$ steps of s but no steps of ℓ. In I, s executes at least $2h$ complete iterations of the do-while loop in lines 44–57. Moreover, since it occurs after time t, from the code, there are at least two iterations in which $\mathit{hbCounter}_s[\ell]$ reaches 0 and s executes the code starting in line 30.

In the first iteration, s reads $\mathit{HbRegister1}[\ell, s]$ and $\mathit{HbRegister2}[\ell, s]$. Let r_1 and r_2 be the responses, respectively. Since ℓ takes no steps during I, ℓ can have an outstanding operation on at most one register during I. Thus, either (1) the read by s on $\mathit{HbRegister1}[\ell, s]$ is not concurrent with any other operations or (2) the read by s on $\mathit{HbRegister2}[\ell, s]$ is not concurrent with any other operations.[11]

Suppose (1) holds (the other case is analogous). By the non-triviality property of abortable registers, the read by s returns a value $v \neq \bot$. In the next iteration in which $\mathit{hbCounter}_s[\ell]$ reaches 0, s reads $\mathit{HbRegister1}[\ell, s]$ again. This read returns the same value v, since ℓ has not taken any steps and it does not have a concurrent operation on $\mathit{HbRegister1}[\ell, s]$. Therefore, the guard in line 35 evaluates to *false* and s increases $\mathit{hbTimeout}_s[\ell]$ in line 39. Since this increase occurs after time t, it contradicts (*) and shows the lemma.

Corollary 10 $\ell \in \mathit{pcandidates} \cap \mathit{Timely}$.

In the final part of the proof, we show that processes in *infcandidates* eventually set their *leader* variable permanently to ℓ. As a result, there is a time after which their

[11] This is the place where we need two heartbeat registers. If there was only one, ℓ may have stopped taking steps while leaving an outstanding write on the heartbeat register, which can cause s to get a $\bot$ value and not time out on ℓ, even though ℓ is slow.

LEADER is either ℓ or ?. Recall that the distinction between *leader* and LEADER is that a process sets LEADER to ? when it stops being a candidate, whereas *leader* is left untouched.

Corollary 11 *For every process $p \in infcandidates$, there is a time after which $\ell \in activeSet_p$.*

Proof Immediately from Lemma 34 and Corollary 10.

Lemma 48 *For every process $p \in infcandidates$, there is a time after which $leader_p = \ell$.*

Proof Let $p \in infcandidates$. By Lemmas 43 and 42, we can find a time t after which (a) $activeSet_p$ contains only processes in *infcandidates*, and (b) for every process $q \in activeSet_p$, $counter_p[q] = c_q$. Since $p \in infcandidates$, p sets $leader_p$ in line 47 infinitely many times. By (a), (b), Lemma 45, Corollary 11, and the way p sets $leader_p$ in line 47, there is a time after which, if p sets $leader_p$, p sets $leader_p$ to ℓ. Thus, there is a time after which $leader_p = \ell$.

From Lemma 48 and the way p sets LEADER_p to $leader_p$ or "?" in the code of Fig. 6, we have:

Corollary 12

(a): *For every process $p \in pcandidates$, there is a time after which $\text{LEADER}_p = \ell$.*
(b): *For every process $p \in infcandidates$, there is a time after which $\text{LEADER}_p \in \{?, \ell\}$.*

Putting together the above results, we get:

Lemma 49 *$\ell \in (Pcandidates \cup Rcandidates) \cap Timely$. Furthermore, the following holds:*

1. *There is a time after which $\text{LEADER}_\ell = \ell$.*
2. *For every process $p \in Pcandidates$, there is a time after which $\text{LEADER}_p = \ell$.*
3. *For every process $p \in Rcandidates$, there is a time after which $\text{LEADER}_p \in \{?, \ell\}$.*

Proof Since $\ell \in pcandidates$, we have that $\ell \in infcandidates$, and so by Lemma 26, $\ell \in Pcandidates \cup Rcandidates$. By Lemma 47, $\ell \in (Pcandidates \cup Rcandidates) \cap Timely$. We now show that the above three properties hold:

1. Since $\ell \in pcandidates$, from Corollary 12(a), there is a time after which $\text{LEADER}_\ell = \ell$.
2. Let $p \in Pcandidates$. By Lemma 26, $p \in pcandidates$. By Corollary 12(a), there is a time after which $\text{LEADER}_p = \ell$.
3. Let $p \in Rcandidates$. Since every process in *Rcandidates* is correct, either $p \in ncandidates$ or $p \in infcandidates$. If $p \in ncandidates$ then, by Lemma 27, there is a time after which $\text{LEADER}_p = ?$. If $p \in infcandidates$ then, by Corollary 12(b), there is a time after which $\text{LEADER}_p \in \{?, \ell\}$. So in both cases there is a time after which $\text{LEADER}_p \in \{?, \ell\}$.

Putting the above facts together, we show that the algorithm described in this section implements Ω_Δ:

Theorem 8 *The algorithm in Figs. 4, 5, and 6 implements Ω_Δ in a system with abortable registers.*

Proof Property (2) of Ω_Δ holds by Corollary 6. If $Pcandidates \cap Timely = \emptyset$, Property (1) of Ω_Δ trivially holds. If $Pcandidates \cap Timely \neq \emptyset$, Assumption 9 holds. In this case, we can apply Lemma 49 which shows that Property (1) of Ω_Δ holds.

6 Using Ω_Δ to achieve adaptive progress

We now explain how Ω_Δ can be used to obtain an AP implementation of an object O of type T, for any type T.

Given any type T, we first use the universal construction of [2] to get a wait-free implementation of an object O_{QA} of type T_{QA}—the query-abortable counterpart of T. Intuitively, an object O_{QA} of type T_{QA} behaves like an object O of type T except that (a) if an operation executes concurrently with another operation, it may abort, with or without taking effect, and return a special value $\perp$; and (b) there is an additional operation called QUERY. A process can call QUERY to determine the fate of its last non-QUERY operation *op* on the object: if *op* took effect then QUERY returns the response that should have been returned by *op*; otherwise, QUERY returns a special value $\mathcal{F}$ to indicate that *op* did not take effect. As with other operations, a QUERY operation can also abort and return $\perp$ (this can occur only if it is concurrent with other operations on the object). A formal definition of the query-abortable type T_{QA} is given in [2].

We then use Ω_Δ to transform the wait-free implementation of O_{QA} of type T_{QA} into an AP implementation of O of type T, as shown in Fig. 8. Intuitively, when p wants to execute an operation *op* on O, p first waits until $\text{LEADER}_p \neq p$ (to ensure that the use of Ω_Δ is canonical), and then p sets the input variable CANDIDATE_p of Ω_Δ to *true*, to indicate that it now wants to compete for the leadership. If Ω_Δ tells p that it is the leader (i.e., $\text{LEADER}_p = p$) then p executes a sequence of *op* and QUERY operations on O_{QA}, as illustrated in Fig. 9, until p is successful. The first operation is *op* (shown by the double circle), and the corresponding response is either a "normal" response $v \neq \perp$ or $\perp$ (indicating that the operation aborted).

If it is a normal response $v \neq \perp$ then p is done; in this case, p sets CANDIDATE_p to *false* to relinquish the leadership and exits the procedure *invoke*(*op*, O, T) by returning v. If the response is $\perp$, p is uncertain whether the aborted operation *op* took effect or not. In this case, p executes a QUERY

Fig. 8 AP implementation of any type T from its query-abortable counterpart T_{QA} and $\Omega_\triangle$

```
CODE FOR PROCESS p:          { O and O_QA are objects of type T and T_QA, respectively }
{ Initial state }
   CANDIDATE = false
{ Main code }
1    { procedure to execute an operation op of object O of type T }
     procedure invoke(op, O, T)
2       wait until LEADER ≠ p                       { Ω_△ is used in the canonical way }
3       CANDIDATE ← true                            { p now competes for the leadership }
4       op' ← op
5       repeat forever
6          if LEADER = p do
7             res ← invoke(op', O_QA, T_QA)
8             if res ∉ {⊥, F} then CANDIDATE ← false; return res
9             if res = ⊥ then op' ← QUERY
10            if res = F then op' ← op
```

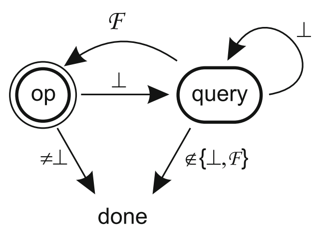

Fig. 9 Sequence of operations executed on object O_{QA} of type T_{QA} by the implementation in Fig. 8

operation to try to find out. While QUERY returns $\perp$, p continues executing QUERY operations. If a QUERY returns a "normal" response $v \notin \{\perp, \mathcal{F}\}$ then p knows that its previous execution of *op* took effect and that v is the corresponding response—so p is done. If QUERY returns $\mathcal{F}$ then p knows that its previous execution of *op* did not take effect, so p tries to execute *op* again. If, at any time, $\Omega_\triangle$ tells p that it is not the leader anymore, (i.e., $\text{LEADER}_p \neq p$) then p stops trying to execute operations on O_{QA}.

It is worth pointing out that the wait for $\text{LEADER}_p \neq p$ in line 2, which ensures a canonical use of $\Omega_\triangle$, is crucial for obtaining an implementation that achieves adaptive progress. Without it, a strict subset of timely processes would be able to monopolize the access to the implemented object O: they would be able to execute an infinite sequence of operations on O and win every competition for leadership among themselves, thereby preventing all the other timely processes from executing their operations. However, the enhanced leader election properties that we get from a canonical use of $\Omega_\triangle$ ensure that the access to the object O remains fair among all the timely processes, so they all eventually complete all their operations on O. Intuitively, this is because when $\Omega_\triangle$ is used in a canonical way, a subset of timely processes cannot pass the leadership back and forth between themselves while preventing the other timely processes, who are also candidates, from getting the leadership forever: such a behavior would contradict Corollary 3 that states that eventually the leader is elected among the set of timely processes who remain candidate forever! This intuitive argument is used in a more precise way in the proof of Theorem 51.

We now show the correctness of this algorithm. Henceforth we consider an arbitrary run R of the algorithm.

Lemma 50 *For every process p, when p is in line 2, $\text{CANDIDATE}_p = \textit{false}$.*

Proof Let p be any process. Initially, $\text{CANDIDATE}_p = \textit{false}$. Moreover, when p executes procedure *invoke* in line 1, p sets CANDIDATE_p to *false* before returning. So whenever p enters the procedure *invoke* in line 1, it does so with $\text{CANDIDATE}_p = \textit{false}$.

From Lemma 50 and p's code, it is clear that in the algorithm in Fig. 8 the use of $\Omega_\triangle$ is canonical.

Lemma 51 *For every operation op of type T, if a timely process p calls procedure invoke(op, O, T) in line 1 then p eventually returns from this procedure.*

Proof Suppose, by contradiction, that there is an operation *op* of type T and a timely process p that calls procedure *invoke*(*op*, O, T) in line 1, but p never returns from this procedure. Since p is timely, p is correct, and so p executes forever in the procedure. By Lemmas 50 and 4, p does not wait forever in line 2. Thus, p loops forever in the repeat loop of line 5. Before entering this loop, p sets CANDIDATE_p to *true*. Since p never returns, it is clear from p's code that CANDIDATE_p remains *true* forever. Therefore, $p \in \textit{Pcandidates}$. So there is at least one timely process in *Pcandidates* (namely, p). Since $\Omega_\triangle$ is used in the canonical way, by Corollary 3, there is a timely process ℓ in *Pcandidates* such that:

(a) there is a time after which $\text{LEADER}_\ell = \ell$, and
(b) there is a time after which, for every correct process $p \neq \ell$, $\text{LEADER}_p \neq p$.

Since $\ell \in$ *Pcandidates*, there is a time after which $\text{CANDIDATE}_\ell = true$. Thus, from Lemma 50 and ℓ's code, it is clear that there is a time T_0 after which ℓ loops forever in the repeat loop of line 5. By (a) above, ℓ executes lines 7–10 infinitely many times.

Claim *There is a time T_1 after which no process $p \neq \ell$ executes lines* 7–10.

The proof of this claim is immediate from (b) above, the guard in line 6, and the fact that O_{QA} is wait-free.

Therefore, there is a time $T_2 > \max\{T_0, T_1\}$ after which ℓ starts executing an operation on O_{QA} (in line 7), and this execution is not concurrent with any other operation executions on this object. Since O_{QA} is query-abortable, this execution returns a value $v \neq \perp$. If $v \neq \mathcal{F}$ then ℓ subsequently exits the *invoke* procedure in line 8—which contradicts the definition of T_0. So, $v = \mathcal{F}$, and ℓ sets op' to op in line 10. Note that since op is an operation of O, $op \neq$ QUERY. Thus, in the next iteration of the repeat loop in line 5, ℓ executes operation $op' \neq$ QUERY on O_{QA} in line 7. Since this execution is not concurrent with any other operation executions on O_{QA} and $op' \neq$ QUERY, it returns some value $v' \notin \{\perp, \mathcal{F}\}$. Therefore p exits the *invoke* procedure in line 8, and it does so after time T_0—a contradiction to the definition of T_0 that concludes the proof of the lemma.

Theorem 13 *The algorithm in Fig. 8 uses Ω_Δ to obtain an AP implementation of an arbitrary type T from a wait-free implementation of its query-abortable counterpart T_{QA}.*

Proof Let T be an arbitrary type and T_{QA} be its query-abortable counterpart. Consider a correct process p that executes *invoke*(op, O, T) in the algorithm of Fig. 8. This execution can cause executions of op or QUERY operations on O_{QA} only according to the pattern shown in Fig. 9. Note that op can take effect at most once (because p re-executes op on O_{QA} only if it determines that its previous execution of op on O_{QA} aborted without taking effect). Moreover, if p returns from *invoke*(op, O, T) then op takes effect exactly once, and p returns a correct non-$\perp$ response (the response is correct because O_{QA} is the query-abortable counterpart of O). Therefore, Fig. 8 is an implementation of type T from T_{QA} and Ω_Δ. From Lemma 51, this is an AP implementation.

Let T be an arbitrary object type. Since (a) there is an implementation of its query-abortable counterpart T_{QA} from abortable registers [2], and (b) there is an implementation of Ω_Δ using only abortable registers (Theorem 8), from Theorem 13 we conclude the following:

Theorem 14 *Every type T has an AP implementation from abortable registers.*

7 Related work

This work is related to notion of partial synchrony [6], to the concepts of obstruction-freedom [11] and wait-freedom [10], to the algorithms that boost obstruction-freedom to wait-freedom given in [7,9,15], to the algorithms that implement failure detector Ω in partially-synchronous systems given in [1], and to the work on abortable and query-abortable object types described in [2].

The notion of partial synchrony was introduced by Dwork et al. [6] for message-passing systems, where timeliness involves not only processes but also communication links. That work shows how to solve consensus in a system with process crashes, assuming that *all* correct processes and links between them are eventually timely.

Algorithms that boost obstruction-freedom to wait-freedom are given in [7,15]. The key idea in these algorithms is to avoid contention so that a process can execute solo and hence terminate the obstruction-free operation. These algorithms work assuming that all correct processes are timely, i.e., the whole system is partially synchronous. If some correct processes are not timely, however, these algorithms have runs such that no correct process (not even the timely ones) makes any progress. Intuitively, this is because a single slow or unstable process can prevent all correct processes from making progress. So they are not gracefully degrading when synchrony decreases.

Going into more detail, the basic technique to avoid contention in [7] is similar to the one in the greedy contention manager [8]: processes obtain a timestamp and the process p_s with smallest timestamp is allowed to execute while others must wait for p_s to finish. This scheme by itself cannot tolerate crashes: for example, if p_s crashes, other processes block forever. To overcome this limitation, [7] proposes that (a) p_s periodically increments a heartbeat and processes use a timeout on the heartbeat to stop waiting on p_s, and (b) if there is a premature timeout, p_s causes other processes to wait again and increase the timeout value. This transformation uses atomic registers, and it would not work with abortable registers. Moreover, if p_s is not timely, then p_s may not make progress and it may also prevent timely processes from making progress.

In [15], the basic technique to avoid contention is to use a lock to provide mutual exclusion. To tolerate crashes, the process holding a lock periodically increments a heartbeat and processes use a timeout on the heartbeat to release the lock and let another process acquire the lock. A premature timeout causes the lock to be released even though the (former) lock holder is still executing. In that case, the former lock holder waits until the new holder releases it or times out, and increases the timeout value. This transformation uses compare-and-swap objects to implement the lock, which is a much stronger object than the abortable registers we use.

We should note that the work in [15] is concerned about efficiency, that is, ensuring that processes terminate their operations in a small number of steps. Efficiency is provided under the assumption that all correct processes are timely. In contrast, our work is concerned about termination of timely processes, and we ensure this property independent of the behavior of other processes. We do not focus on efficiency here, but this may be a topic for future research.

As in [7,15], the core idea in our algorithm is to choose a process to run solo, and we make this choice in a fair manner to avoid starvation. In contrast to those works, however, we choose this process using Ω_Δ, a modular abstraction that selects a leader among the current set of contenders, provided that at least one of them is timely. Our implementation of Ω_Δ includes new techniques to prevent an unstable process from being repeatedly re-elected as the leader forever—a behavior that could prevent timely processes from making progress. For example, in our implementation of Ω_Δ, in contrast to the timestamps used in [7] (which are fixed for each process's operation) the counter of a process p may change during the execution of an operation by p, to repeatedly "punish" p if p is unstable. Moreover, processes must use Ω_Δ in a particular way to ensure that the leadership rotates fairly among contenders, as we explain in Sect. 3. Finally, in the implementation of Ω_Δ using abortable registers, we introduce techniques to coordinate the reading and writing of the register to prevent operations from always aborting, as explained in Sect. 5.

In [9], Guerraoui et al. determine the weakest failure detectors to boost obstruction-freedom. In particular, [9] describes (a) an algorithm that boosts obstruction-freedom to wait-freedom using $I_{\diamond P}$ (a failure detector that is equivalent to the *eventually perfect failure detector* $\diamond P$) and (b) an algorithm that implements $I_{\diamond P}$ in a system where all the correct processes are timely. By combining these two algorithms, one obtains wait-free implementations in systems where all correct processes are timely. But this combined algorithm is not gracefully degrading: if only some of the correct processes are timely, the non-timely processes can prevent all the timely processes from making progress.

Ω_Δ, a dynamic variant of failure detector Ω [5,4], is specified in terms of the timeliness properties (if any) of the candidates for leadership. Our algorithms for Ω_Δ include several techniques that were first proposed in [1] for implementing Ω in systems with weak reliability and synchrony assumptions. Another dynamic variant of Ω, denoted I_{Ω^*}, was previously proposed in [9] to boost obstruction-freedom to lock-freedom. In contrast to Ω_Δ, the specification of I_{Ω^*} does not refer to process timeliness (and so it is not useful to obtain AP implementations: the progress property of such implementations is stated in terms of the degree of synchrony of each process). The implementation of I_{Ω^*} given in [9] uses atomic registers and assumes that all processes are timely.

Finally, our AP implementations use the universal construction of query-abortable types given in [2].

Acknowledgments The authors are grateful to Stephanie L. Horn and the anonymous referees for their many helpful comments.

Appendix: Implementing activity monitors using registers

Figure 2 gives an algorithm that implements the activity monitor $\mathcal{A}(p, q)$ for any pair of distinct processes p and q.[12] This algorithm uses a shared register $HbRegister[q, p]$ that is written by q and read by p. Intuitively, q periodically increments $HbRegister[q, p]$ when $\text{ACTIVE-FOR}_q[p] = on$, and q sets $HbRegister[q, p]$ to -1 and sleeps when $\text{ACTIVE-FOR}_q[p] = off$. Process p monitors $HbRegister[q, p]$ when $\text{MONITORING}_p[q] = on$ (otherwise p sleeps). To monitor $HbRegister[q, p]$, p executes in a loop and, every $hbTimeout_p$ iterations of the loop, p reads $HbRegister[q, p]$ and decides on one of three possibilities: (1) if $HbRegister[q, p]$ has a negative value, p sets $\text{STATUS}_p[q]$ to *inactive*; (2) otherwise, if $HbRegister[q, p]$ increased since the last time p checked, p sets $\text{STATUS}_p[q]$ to *active* and $allow\_increment_p$ to *true*; (3) otherwise, $HbRegister[q, p]$ has not changed since the last time p checked, so p "times out" on q : p sets $\text{STATUS}_p[q]$ to *inactive*. Moreover, if $allow\_increment_p$ is *true*, p increments $\text{FAULTCNTR}_p[q]$ and $hbTimeout_p$, and p sets $allow\_increment_p$ to *false*. The role of $allow\_increment_p$ is to ensure that p increments $\text{FAULTCNTR}_p[q]$ only if p sees that q is active and subsequently times out on q. This prevents p from incrementing $\text{FAULTCNTR}_p[q]$ infinitely often if q crashes.

We now show that, for any two processes $p \neq q$, the algorithm in Fig. 2 implements an activity monitor $\mathcal{A}(p, q)$ using registers. Henceforth, we consider an arbitrary run R of this algorithm such that p is correct (note that if p is not correct, then the properties of $\mathcal{A}(p, q)$ are trivially satisfied).

In the following, the value of var_p at time t is denoted by var_p^t.

Lemma 52 *(1) $hbTimeout_p \geq 1$ and $hbTimeout_p$ is monotonically nondecreasing. (2) $hbTimer_p \geq 0$.*

Proof (1) Initially, $hbTimeout_p = 1$. Subsequently, $hbTimeout_p$ can only change by being incremented. Thus, $hbTimeout_p \geq 1$ and $hbTimeout_p$ is monotonically nondecreasing.

(2) Initially, $hbTimer_p = 1$. Moreover, $hbTimer_p$ is changed in only two ways: (a) p sets $hbTimer_p$ to $hbTimeout_p$, or (b) p decrements $hbTimer_p$ only if $hbTimer_p \geq 1$. In either case, $hbTimer_p \geq 0$.

12 Note that it is trivial to implement the activity monitor $\mathcal{A}(p, q)$ when $p = q$.

Lemma 53 *If q is correct and there is a time after which* ACTIVE-FOR$_q[p]$=*on then*

(a) *there is a time after which $HbRegister[q, p] \geq 0$,*
(b) *there is a time after which $HbRegister[q, p]$ is monotonically nondecreasing, and*
(c) *q increments $HbRegister[q, p]$ infinitely often.*

Proof Suppose q is correct and there is a time after which ACTIVE-FOR$_q[p]$=*on*. Then, it is clear from q's code that eventually q loops forever in the while loop of line 4. So it is clear that (a), (b), and (c) hold.

Lemma 54 *For all t and t', if $t \leq t'$ and $HbRegister[q, p]^{t'} \geq 0$ then $HbRegister[q, p]^t \leq HbRegister[q, p]^{t'}$.*

Proof Let t and t' be such that $t \leq t'$ and $HbRegister[q, p]^{t'} \geq 0$. If $HbRegister[q, p]^t < 0$ then the lemma trivially holds. Now assume $HbRegister[q, p]^t \geq 0$. Note that (a) when q sets $HbRegister[q, p]$ to a non-negative value, q sets it to $hbCounter_q$, and (b) $hbCounter_q$ is monotonically nondecreasing.

Lemma 55 *If q is p-timely then there exists an integer $j \geq 1$ such that for every time interval $[t, t']$ containing at least j steps of p, if $HbRegister[q, p]^{t'} \geq 0$ then $HbRegister[q, p]^t < HbRegister[q, p]^{t'}$.*

Proof Assume that q is p-timely. Since q is correct, from the code of the algorithm, it is clear that there exists an integer $i \geq 1$ such that if, at any time t, $HbRegister[q, p]^t \geq 0$ then q executes one of the following two statements within i steps:

– (a) q increases $HbRegister[q, p]$ by 1 (in line 6), or
– (b) q sets $HbRegister[q, p]$ to -1 (in line 2).

Since q is p-timely, there exists an integer $k \geq 0$ such that (*) every time interval containing $k + 1$ steps of p has at least one step of q.

Let $j = ik + 2$ and consider any time interval $[t, t']$ containing at least j steps of p. If $HbRegister[q, p]^t < 0$ then the lemma trivially follows, so assume that $HbRegister[q, p]^t \geq 0$. Time interval $[t + 1, t']$ has at least $j - 1 = ik + 1$ steps of p. By (*), time interval $[t + 1, t']$ has at least i steps of q. Thus, at some time in $[t + 1, t']$, (a) or (b) occurs.

Consider the first time t'' in $[t + 1, t']$ when (a) or (b) occurs. There are two possible cases:

– If (a) occurs then $HbRegister[q, p]^{t''} = HbRegister[q, p]^t + 1$. Since $t'' \leq t'$ and $HbRegister[q, p]^{t'} \geq 0$, by Lemma 54, $HbRegister[q, p]^{t''} \leq HbRegister[q, p]^{t'}$. Thus, $HbRegister[q, p]^t + 1 \leq HbRegister[q, p]^{t'}$.
– If (b) occurs then note that at time t'', $hbCounter_p$ is equal to $HbRegister[q, p]^t$, that is, $hbCounter_p^{t''} = HbRegister[q, p]^t$. At time t'', $HbRegister[q, p]^{t''} = -1$, and at time $t' \geq t''$, $HbRegister[q, p]^{t'} \geq 0$. Thus, at some time in $[t'' + 1, t']$, q sets $HbRegister[q, p]$ to a non-negative value (this must occur in line 6). Let t''' be the first time in $[t'' + 1, t']$ when this occurs. At time t''', $HbRegister[q, p]$ is set to $hbCounter_p^{t''} + 1$ (because p increments $hbCounter_p$ in line 5). Thus $HbRegister[q, p]^{t'''} = hbCounter_p^{t''} + 1 = HbRegister[q, p]^t + 1$. Since $t''' \leq t'$ and $HbRegister[q, p]^{t'} \geq 0$, by Lemma 54, $HbRegister[q, p]^{t'''} \leq HbRegister[q, p]^{t'}$. Thus $HbRegister[q, p]^t + 1 \leq HbRegister[q, p]^{t'}$.

In both cases above, $HbRegister[q, p]^t < HbRegister[q, p]^{t'}$.

Lemma 56 *If q is p-timely then p increments $hbTimeout_p$ only finitely many times.*

Proof Assume, by contradiction, that q is p-timely and p increments $hbTimeout_p$ infinitely many times. Note that p increments $hbTimeout_p$ only in line 25.

Claim 1 *There is a time after which, each time p executes line 21, p finds that the guard "$hbCounter_p \geq 0$ and $hbCounter_p \leq prevHbCounter_p$" in line 21 is false.*

We now prove this claim. Since p increments $hbTimeout_p$ infinitely many times in line 25, p sets $hbCounter_p$ to $HbRegister[q, p]$ infinitely many times in line 16. For each $i \geq 1$, let t_i be the time when p sets $hbCounter_p$ to $HbRegister[q, p]$ for i-th time (in line 16). For convenience, let $t_0 = 0$. Let c_i be the value of $hbCounter_p$ at time t_i. Thus, $c_i = hbCounter_p^{t_i} = HbRegister[q, p]^{t_i}$. It is clear from lines 15 and 16 that (a) for all $i \geq 1$, $prevHbCounter_p^{t_i} = hbCounter_p^{t_{i-1}} = c_{i-1}$.

Since q is p-timely, by setting $t = t_{i-1}$ and $t' = t_i$ in Lemma 55, we have (b) there exists an integer j such that, for every $i \geq 1$, if time interval $[t_{i-1}, t_i]$ has j steps of p and $c_i \geq 0$ then $c_{i-1} < c_i$.

Claim 1.1 *There exists k such that, for every $i \geq k$, time interval $[t_{i-1}, t_i]$ has at least j steps of p.*

To show Claim 1.1, first note that $hbTimeout_p$ is monotonically nondecreasing (Lemma 52). Since, p increments $hbTimeout_p$ infinitely many times (by assumption), $hbTimeout_p$ increases without bound. For each $i \geq 0$, let τ_i be the value of $hbTimeout_p$ at time t_i. Thus, $\lim_{i\to\infty} \tau_i = \infty$. It is clear from p's code that, from time t_i to time t_{i+1}, p decrements $hbTimer_p$ in line 12 at least τ_i times until $hbTimer_p$ reaches 0. Therefore, from time t_i to t_{i+1}, p executes at least τ_i steps. Since $\lim_{i\to\infty} \tau_i = \infty$, there exists k such that, for every $i \geq k$, $\tau_{i-1} \geq j$. So, for every $i \geq k$, time interval $[t_{i-1}, t_i]$ has at least j steps of p, which shows Claim 1.1.

From (b) and Claim 1.1, for every $i \geq k$, if $c_i \geq 0$ then $c_{i-1} < c_i$. Thus, from (a) and the definition of c_i, for every $i \geq k$, if $hbCounter_p^{t_i} \geq 0$ then $prevHbCounter_p^{t_i} < hbCounter_p^{t_i}$. So, for every $i \geq k$, the condition "$hbCounter_p^{t_i} \geq 0$ and $hbCounter_p^{t_i} \leq prevHbCounter_p^{t_i}$" is *false*. From p's code it is now clear that Claim 1 holds.

Note that p can increment $hbTimeout_p$ only in line 25, and only if the guard "$hbCounter_p \geq 0$ and $hbCounter_p \leq prevHbCounter_p$" in line 21 is *true*. Thus, Claim 1 implies that p increments $hbTimeout_p$ only finitely many times—a contradiction that shows the lemma.

In the next six lemmas we prove that the six properties of $\mathcal{A}(q, p)$ are satisfied.

Lemma 57 *If there is a time after which* MONITORING$_p[q]$= *off then there is a time after which* STATUS$_p[q]$=?.

Proof Suppose there is a time after which MONITORING$_p[q]$ = *off*. Since p is correct, from p's code it is clear that p eventually loops forever in the while loop of line 9. Before getting stuck in this loop, p sets STATUS$_p[q]$ to ? and p does not set STATUS$_p[q]$ afterwards.

Lemma 58 *If there is a time after which* MONITORING$_p[q]$= *on then there is a time after which* STATUS$_p[q]$≠?.

Proof Suppose there is a time after which MONITORING$_p[q]$= *on*. Since p is correct, from p's code it is clear that p eventually loops forever in the while loop of line 11. Before getting stuck in this loop, p sets $hbTimer_p$ to $hbTimeout_p$, where $hbTimeout_p \geq 1$ by Lemma 52. From the way p decrements $hbTimer_p$ in line 12, it is clear that eventually p executes line 13 with $hbTimer_p = 0$. Then, p finds that one of the three if statements in lines 17, 18, or 21 has a condition that is satisfied, and p sets STATUS$_p[q]$ to *inactive*, *active*, or *inactive*, respectively. Thereafter, STATUS$_p[q]$≠?.

Lemma 59 *If q crashes or there is a time after which* ACTIVE-FOR$_q[p]$=*off then there is a time after which* STATUS$_p[q]$≠*active.*

Proof Suppose q crashes or there is a time after which ACTIVE-FOR$_q[p]$=*off*. Initially, STATUS$_p[q]$ = ?. If p never sets STATUS$_p[q]$ to *active*, then the lemma trivially holds. Now assume that p sets STATUS$_p[q]$ to *active* at least once. Note that p sets STATUS$_p[q]$ to *active* only in line 19.

We claim that p executes line 19 only finitely many times. Assume, by contradiction, that p executes line 19 infinitely many times. Since q crashes or there is a time after which ACTIVE-FOR$_q[p]$=*off*, from q's code, there is a time after which $HbRegister[q, p]$ does not change. Note that p sets $hbCounter_p$ only in line 16, and p sets it to $HbRegister[q, p]$. Thus, there is a time after which $hbCounter_p$ does not change. Since p executes line 19 infinitely many times, p sets $prevHbCounter_p$ to $hbCounter_p$ in line 15 infinitely many times. Thus, there is a time after which $hbCounter_p = prevHbCounter_p$. So, from the guard "$hbCounter > prevHbCounter$" in line 18, it is clear that p executes line 19 only finitely many times—a contradiction that shows the claim.

Let t be the time when p executes line 19 for the last time. There are two cases:

(1) *After time t, p remains forever in the loop of line* 11. By Lemma 52, $hbTimer_p \geq 0$. Since p is correct, from p's code it is clear that p eventually executes line 13 with $hbTimer_p = 0$ after time t. Then, p finds that one of the three if statements in lines 17, 18, or 21 has a condition that is satisfied. From the definition of t, it cannot be the if statement in line 18. Thus, p sets STATUS$_p[q]$ to *inactive* in line 17 or 22. Thereafter, STATUS$_p[q]$ ≠ *active*.

(2) *After time t, p exits the loop of line* 11. Since p is correct, p sets STATUS$_p[q]$ to ? in line 8 after time t. Thereafter, STATUS$_p[q]$ ≠ *active*.

Lemma 60 *If q is p-timely and there is a time after which* ACTIVE-FOR$_q[p]$=*on then there is a time after which* STATUS$_p[q]$≠*inactive.*

Proof Suppose q is p-timely, and there is a time after which ACTIVE-FOR$_q[p]$=*on*. Initially, STATUS$_p[q]$ = ?. If p never sets STATUS$_p[q]$ to *inactive*, then the lemma trivially holds. Now assume that p sets STATUS$_p[q]$ to *inactive* at least once. Note that p sets STATUS$_p[q]$ to *inactive* only in lines 17 or 22.

Claim 1 *p sets* STATUS$_p[q]$ *to inactive in line* 17 *only finitely many times.*

To prove this claim, note that before executing line 17, p sets $hbCounter_p$ to $HbRegister[q, p]$ in line 16. Since q is p-timely, q is correct. Since q is correct and there is a time after which ACTIVE-FOR$_q[p]$=*on*, by Lemma 53, there is a time after which $HbRegister[q, p] \geq 0$. Therefore, the guard "$hbCounter < 0$" in line 17 can evaluate to *true* only finitely many times. So p sets STATUS$_p[q]$ to *inactive* in line 17 only finitely many times. So Claim 1 holds.

Claim 2 *p sets* STATUS$_p[q]$ *to inactive in line* 22 *only finitely many times.*

Assume, by contradiction, that (a) p sets STATUS$_p[q]$ to *inactive* in line 22 infinitely many times. From this assumption and p's code, it is clear that p executes each of the three if statements in lines 17, 18, and 21 infinitely many times. Furthermore, since q is correct, from Lemma 53 and the way p sets $prevHbCounter_p$ and $hbCounter_p$ in lines 15 and 16, it is clear that p executes the if statement of line 18 infinitely many times while the guard "$hbCounter_p \geq 0$ and $hbCounter_p > prevHbCounter_p$" is *true*. So, (b) p sets $allow\_increment_p$ to *true* infinitely many times in line 20.

Claim 2.1 *p increments $hbTimeout_p$ infinitely often.*

To prove this claim, we now show that for each time t, there exists $t' > t$ such that p increments $hbTimeout_p$ at time t' (in line 25). Consider any time t. Let $t_1 > t$ be the first time after t when p sets $allow\_increment_p$ to *true* in line 20; note that t_1 exists by (b). Let $t_2 > t_1$ be the first time after t_1 when p sets $\text{STATUS}_p[q]$ to *inactive* in line 22; note that t_2 exists by (a). Furthermore, since p can set $allow\_increment_p$ to *false* only in line 26, $allow\_increment_p$ is still *true* at time t_2. Thus, after p executes line 22 at time t_2, p finds that $allow\_increment_p = true$ in line 23, and so p increments $hbTimeout_p$ in line 25. This shows Claim 2.1.

Since q is p-timely, by Lemma 56, p increments $hbTimeout_p$ only finitely many times. This contradicts Claim 2.1 and concludes the proof of Claim 2.

From Claims 1 and 2, p sets $\text{STATUS}_p[q]$ to *inactive* only finitely many times. Let t be the time when p sets $\text{STATUS}_p[q]$ to *inactive* for last time. There are two cases:

(1) *After time t, p remains forever in the loop of line* 11. By Lemma 52, $hbTimer_p \geq 0$. Since p is correct, from p's code it is clear that p eventually executes line 13 with $hbTimer_p = 0$ after time t. After that, p finds that one of the three if statements in lines 17, 18, or 21 has a guard that is satisfied. From the definition of t, it cannot be the if statement in line 17 or 21. Thus, p sets $\text{STATUS}_p[q]$ to *active* in line 19. Thereafter, $\text{STATUS}_p[q] \neq inactive$.

(2) *After time t, p exits the loop of line* 11. Since p is correct, p sets $\text{STATUS}_p[q]$ to ? in line 8 after time t. Thereafter, $\text{STATUS}_p[q] \neq inactive$.

In both cases above, there is a time after which $\text{STATUS}_p[q] \neq inactive$.

Lemma 61 $\text{FAULTCNTR}_p[q]$ *is bounded if any of the following conditions hold:*

(a) *q is p-timely*
(b) *q crashes*
(c) *there is a time after which* $\text{ACTIVE-FOR}_q[p] = off$
(d) *there is a time after which* $\text{MONITORING}_p[q] = off$

Proof (a): If q is p-timely then, by Lemma 56, p increments $hbTimeout_p$ only finitely many times. Thus, p executes line 25 only finitely many times. So, p executes line 24 only finitely many times. Therefore, p increments $\text{FAULTCNTR}_p[q]$ only finitely many times and $\text{FAULTCNTR}_p[q]$ is bounded.

(b) and (c): Assume q crashes or there is a time after which $\text{ACTIVE-FOR}_q[p]{=}off$. By Lemma 59, there is a time after which $\text{STATUS}_p[q]{\neq}active$. So, p sets $\text{STATUS}_p[q]$ to *active* in line 19 only finitely many times. Thus, (i) p sets $allow\_increment_p$ to *true* in line 20 only finitely many times.

Suppose, by contradiction, that $\text{FAULTCNTR}_p[q]$ is not bounded. Then p increments $\text{FAULTCNTR}_p[q]$ in line 24 infinitely many times. Since p executes line 24 infinitely many times, we have (ii) p sets $allow\_increment_p$ to *false* in line 26 infinitely many times.

From (i) and (ii), there is a time after which $allow\_increment_p = false$, that is, the guard in line 23 is *false*. Therefore, p increments $\text{FAULTCNTR}_p[q]$ in line 24 only finitely many times—a contradiction. So, $\text{FAULTCNTR}_p[q]$ is bounded.

(d): If there is a time after which $\text{MONITORING}_p[q] = off$ then it is clear from p's code that eventually p loops forever in the while loop of line 9. So, $\text{FAULTCNTR}_p[q]$ is bounded.

Lemma 62 $\text{FAULTCNTR}_p[q]$ *increases without bound if all of the following conditions hold:*

(a) *q is not p-timely*
(b) *q is correct*
(c) *there is a time after which* $\text{ACTIVE-FOR}_q[p] = on$
(d) *there is a time after which* $\text{MONITORING}_p[q] = on$

Proof Suppose that conditions (a), (b), (c), and (d) hold. First note that p can change $\text{FAULTCNTR}_p[q]$ only by incrementing it in line 24, and so $\text{FAULTCNTR}_p[q]$ is monotonically nondecreasing. There are two possible cases:

(I) *p increments* $\text{FAULTCNTR}_p[q]$ *infinitely many times*. In this case, $\text{FAULTCNTR}_p[q]$ increases without bound.

(II) *p increments* $\text{FAULTCNTR}_p[q]$ *finitely many times*. In this case, it is clear that p changes $hbTimeout_p$ (in line 25) only finitely many times. So $hbTimeout_p$ is bounded.

Since p is correct and (d) holds, p eventually loops forever in the while loop of line 11. Thus, it is clear from p's code that p sets $hbCounter_p$ to $HbRegister[q, p]$ in line 16 infinitely many times.

For each $i \geq 1$, let t_i be the time when p sets $hbCounter_p$ to $HbRegister[q, p]$ for i-th time (in line 16).

Let K be large enough so that, from time t_K onwards, p loops forever in the while loop of line 11.

Claim 1 *There exists an integer $j \geq 1$ such that, for every $i \geq K$, time interval $[t_i, t_{i+1}]$ has at most j steps of p.*

To show this claim, note from the above that $hbTimeout_p$ is bounded by some value $B_1 \geq 1$. So, from p's code and the definitions of K, t_i, and t_{i+1}, for every $i \geq K$, p executes at most B_1 complete loop iterations of the while loop of line 11 (and p does not execute outside the loop) between times t_i and t_{i+1}. From p's code it is also clear that there is a bound $B_2 \geq 1$ on the number of steps that p takes to execute each iteration of this while loop. Let $j = B_1 B_2$. Then, for every $i \geq K$, time interval $[t_i, t_{i+1}]$ has at most j steps of p, where $j \geq 1$. This shows Claim 1.

Since (b) and (c) holds, by Lemma 53, we have (1) there is a time t' after which $HbRegister[q, p] \geq 0$, (2) there is a time after which $HbRegister[q, p]$ is monotonically non-decreasing, and (3) q increments $HbRegister[q, p]$ infinitely often.

Since p sets $hbCounter_p$ to $HbRegister[q, p]$ in line 16 infinitely many times, it is clear from the code that p also executes the if statement of line 18 infinitely many times. From (1), (2), and (3) above, and the way p sets $prevHbCounter_p$ and $hbCounter_p$ in lines 15–16, it is clear that p executes the if statement of line 18 infinitely many times while the guard "$hbCounter_p \geq 0$ and $hbCounter_p > prevHbCounter_p$" is *true*. So, p sets $allow\_increment_p$ to *true* infinitely many times in line 20.

Let t be a time such that p never increments $\text{FAULTCNTR}_p[q]$ after time t; note that t exists by the assumption of case (II). Let t_{allow} be the first time after $\max\{t, t', t_K\}$ when p sets $allow\_increment_p$ to *true* in line 20. Since $\text{FAULTCNTR}_p[q]$ is not incremented after time t (in line 25), $allow\_increment_p$ is not set to *false* after time t. Thus, after time t_{allow}, $allow\_increment_p = true$ forever.

Since q is not p-timely and q is correct, for every integer $k \geq 1$ there exists a time interval that has k steps of p but no steps of q. Let s_{allow} be the number of steps of p up to time t_{allow}. Pick $k = 3j + s_{allow}$, where j is the bound of Claim 1. Then there exists a time interval that has k steps of p but no steps of q. Thus, there exists a time interval $[u, u']$ with $u > t_{allow}$ such that $[u, u']$ has $3j$ steps of p but no steps of q.

Note that $u > t_K$ (because $u > t_{allow} > \max\{t, t', t_K\}$). Thus, by Claim 1, (i) time interval $[u, u']$ contains time interval $[t_g, t_{g+2}]$ for some $g \geq K$. Note that q does not take a step during $[u, u']$ and q is the only process that writes to $HbRegister[q, p]$. Therefore, the value read from $HbRegister[q, p]$ can change at most once during $[u, u']$ (it could change once since q may have an outstanding write at time u). At times t_g, t_{g+1}, and t_{g+2}, process p reads $HbRegister[q, p]$ and stores the result in $hbCounter_p$. Therefore, either $hbCounter_p^{t_g} = hbCounter_p^{t_{g+1}}$ or $hbCounter_p^{t_{g+1}} = hbCounter_p^{t_{g+2}}$. Assume that $hbCounter_p^{t_g} = hbCounter_p^{t_{g+1}}$ (the other case is analogous and omitted). From p's code, $prevHbCounter_p^{t_{g+1}} = hbCounter_p^{t_g}$. Thus, $prevHbCounter_p^{t_{g+1}} = hbCounter_p^{t_g} = hbCounter_p^{t_{g+1}}$. In other words, at time t_{g+1}, $prevHbCounter_p = hbCounter_p$.

By (i), $t_{g+1} > u$. Since $u > t_{allow} > \max\{t, t', t_K\}$, we have $t_{g+1} > t'$. Thus, at time t_{g+1}, $HbRegister[q, p] \geq 0$. Therefore, $hbCounter_p^{t_{g+1}} \geq 0$.

In summary, at time t_{g+1}, p is in line 16 and $hbCounter_p \geq 0$ and $prevHbCounter_p = hbCounter_p$. Thus, when p reaches the if statement in line 21, the guard evaluates to *true*, and so p reaches the if statement in line 23. Recall that, after time t_{allow}, $allow\_increment_p = true$ forever. Since $t_{g+1} > t_{allow}$, the guard in line 23 also evaluates to *true*, and p increments $\text{FAULTCNTR}_p[q]$ in line 24. This incrementing occurs after time t, which contradicts the definition of t. Thus, case (II) cannot occur and this concludes the proof.

Theorem 3 *For any pair of processes $p \neq q$, the algorithm in Fig. 2 implements an activity monitor $\mathcal{A}(p, q)$ using registers.*

Proof Lemmas 57–62 show that the 6 properties of $\mathcal{A}(p, q)$ hold.

References

1. Aguilera, M.K., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: On implementing Omega in systems with weak reliability and synchrony assumptions. Distrib. Comput. **21**(4), 239–314 (2008). A preliminary version of this work appeared in the Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing, pp. 306–314. ACM, New York (2003)
2. Aguilera, M.K., Frolund, S., Hadzilacos, V., Horn, S.L., Toueg, S.: Abortable and query-abortable objects and their efficient implementation. In: Proceedings of the 26th ACM Symposium on Principles of Distributed Computing, pp. 23–32. ACM, New York (2007)
3. Aguilera, M.K., Toueg, S.: Timeliness-based wait-freedom: a gracefully degrading progress condition. In: Proceedings of the 27th ACM Symposium on Principles of Distributed Computing, pp. 305–314. ACM, New York (2008)
4. Chandra, T.D., Hadzilacos, V., Toueg, S.: The weakest failure detector for solving consensus. J. ACM **43**(4), 685–722 (1996)
5. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. J. ACM **43**(2), 225–267 (1996)
6. Dwork, C., Lynch, N.A., Stockmeyer, L.: Consensus in the presence of partial synchrony. J. ACM **35**(2), 288–323 (1988)
7. Fich, F.E., Luchangco, V., Moir, M., Shavit, N.: Obstruction-free algorithms can be practically wait-free. In: Proceedings of the 19th International Symposium on Distributed Computing, vol. 3724 of *LNCS*, pp. 78–92. Springer, Heidelberg (2005)
8. Guerraoui, R., Herlihy, M., Pochon, B.: Toward a theory of transactional contention managers. In: Proceedings of the 24th ACM Symposium on Principles of Distributed Computing, pp. 258–264. ACM, New York (2005)
9. Guerraoui, R., Kapalka, M., Kouznetsov, P.: The weakest failure detectors to boost obstruction-freedom. Distrib. Comput. **20**(6), 415–433 (2008)
10. Herlihy, M.: Wait-free synchronization. ACM Trans. Program. Lang. Syst. **13**(1), 124–149 (1991)
11. Herlihy, M., Luchangco, V., Moir, M.: Obstruction-free synchronization: Double-ended queues as an example. In: Proceedings of the 23rd International Conference on Distributed Computing Systems, pp. 522–529. IEEE Computer Society, New York (2003)
12. Herlihy, M., Wing, J.: Linearizability: a correctness condition for concurrent objects. Trans. Program. Lang. Syst. **12**(3), 463–492 (1990)
13. Lamport, L.: On interprocess communication; part I: basic formalism. Distrib. Comput. **1**(2), 77–85 (1986)
14. Lamport, L.: On interprocess communication; part II: algorithms. Distrib. Comput. **1**(2), 86–101 (1986)
15. Taubenfeld, G.: Efficient transformations of obstruction-free algorithms into non-blocking algorithms. In: Proceedings of the 21st International Symposium on Distributed Computing, vol. 4731 of *LNCS*, pp. 450–464. Springer, Heidelberg (2007)