Timeliness-Based Wait-Freedom: A Gracefully Degrading Progress Condition

Marcos K. Aguilera*
Microsoft Research Silicon Valley
Mountain View, CA, USA

Sam Toueg†
University of Toronto
Toronto, ON, Canada

ABSTRACT

We introduce a simple progress condition for shared object implementations that is gracefully degrading depending on the degree of synchrony in each run. This progress property, called timeliness-based wait-freedom, provides a gradual bridge between obstruction-freedom and wait-freedom in partially synchronous systems. We show that timeliness-based wait-freedom can be achieved with synchronization primitives that are very weak. More precisely, every object has a timeliness-based wait-free implementation that uses only abortable registers (which are weaker than safe registers).

As part of this work, we present a new leader election primitive 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 primitive can be implemented using abortable registers.

Categories and Subject Descriptors

B.3.2 [**Memory structures**]: Design Styles—*Shared memory*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*; D.4.1 [**Operating systems**]: Process Management—*Concurrency*; F.2.m [**Analysis of Algorithms and Problem Complexity**]: Miscellaneous

General Terms

Algorithms, Design, Theory

Keywords

Shared memory, memory contention, graceful degradation, wait-freedom, obstruction-freedom, abortable registers, universal constructions, dynamic leader election

*Work developed while author was at HP Labs.

†This author was partially supported by the National Science and Engineering Research Council of Canada.

PODC'08, August 18–21, 2008, Toronto, Ontario, Canada.

1. INTRODUCTION

1.1 A new progress condition

Three progress 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* [9, 10].

In this paper, we first propose a new progress property, called *timeliness-based wait-freedom* (or briefly TBWF), that provides a natural bridge between the above well-known progress properties in partially synchronous systems. The strength of the progress guarantee provided by TBWF depends on the amount of partial synchrony, if any, in each run. As the degree of partial synchrony "increases", the progress 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 progress property provides *graceful degradation* depending on the degree of synchrony in a run. 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 progress 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 get penalized, but not others. With timeliness-based wait-freedom, 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 progress, but even if they are very slow or unstable (e.g., they repeatedly oscillate between being timely and very slow) they cannot hinder the progress of timely processes. We now explain this progress condition in more detail.

Intuitively, *timeliness-based wait-freedom* 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 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 timeliness-based wait-freedom to obstruction-freedom, lock-freedom and wait-freedom.

We first note that any TBWF implementation *is necessarily obstruction-free.* To see this, consider a TBWF imple-

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

mentation 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, then, by definition, timeliness-based wait-freedom 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, timeliness-based wait-freedom implies obstruction-freedom.

Now consider a TBWF 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 TBWF implementation of O still guarantees some progress, and the amount of progress depends on the amount of synchrony in R. If some process p is timely in R, then the TBWF 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 TBWF 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 TBWF implementation of O is "wait-free in R". Thus, as the number of timely processes increases from 1 to n, the progress guarantee of a TBWF implementation goes from lock-freedom incrementally all the way to wait-freedom.

1.2 Achieving timeliness-based wait-freedom

We next consider the problem of obtaining implementations that are timely-based wait-free. It is well-known that any object has a wait-free (and *a fortiori* TBWF) implementation, provided one is allowed to use some strong synchronization primitives like *compare-and-swap* [9]. But such primitives 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 TBWF 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 TBWF 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 $\perp$). A write operation that aborts may or may not take effect and, since the writer gets back $\perp$ in either case, it does not know whether its write operation succeeded or not.[2]

[2]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.

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

1. We first introduce a dynamic leader election primitive, 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 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 a TBWF 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.[3]

   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 a TBWF 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 TBWF implementations is similar to the "boosting" of obstruction-free implementations into wait-free implementations using synchrony [7, 11] or failure detectors (which in turn can be implemented using synchrony) [8]. In contrast to TBWF implementations, however, the wait-free implementations obtained by boosting in [7, 11, 8] are not gracefully degrading: the boosting algorithms assume that all the correct processes are (eventually) timely,[4] 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, 8] and [11] use atomic registers and compare-and-swap, respectively).

As a final remark, the implementation of Ω_Δ using abortable registers (given in Section 6) implies that one can implement Ω — a failure detector which is sufficient to solve consensus [4] — in a system with abortable registers and

[3]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 $\perp$ 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.

[4]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, 11, 8] and here).

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 a primitive 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 Section 5.2 in a modular way that shields the implementation from low-level synchrony mechanisms, such as timers and timeouts.

Summary of contributions

- We introduce a new progress property, called timeliness-based wait-freedom, for shared object implementations. This progress 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 progress 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 for timeliness-based wait-freedom: a simple one that uses plain (atomic) registers, and a more complex one that uses only abortable registers. The second construction implies that timeliness-based wait-freedom can be achieved with registers that are weaker than safe.
- We specify a new leader election primitive, denoted Ω_Δ, that allows processes to dynamically compete for leadership. In contrast to previously defined dynamic leader election primitives, 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 p determine the current "status" of 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.

The paper is organized as follows. In Section 2, we discuss related work. In Section 3, we explain our shared-memory model and define the timely-based wait-free property. We define the dynamic leader elector Ω_Δ in Section 4. In Section 5, we implement Ω_Δ using registers, in two steps. First, we define activity monitors and implement them using registers in Section 5.1. Then, we use activity monitors and registers to implement Ω_Δ in Section 5.2. In Section 6, we implement Ω_Δ using abortable registers. We conclude the paper in Section 7 by showing how to use Ω_Δ to achieve a TBWF implementation of an arbitrary type. Due to space limitations, proofs are omitted here but they can be found in [3].

2. RELATED WORK

This work is related to the concepts of obstruction-freedom [10] and wait-freedom [9], to the algorithms that boost obstruction-freedom to wait-freedom given in [7, 8, 11], 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 algorithms that boost obstruction-freedom to wait-freedom given in [7, 11] assume that all the correct processes are timely.[5] If some correct processes are not timely, however, it is not difficult to see that these algorithms have runs such that no correct process (not even the timely ones) makes any progress [3]. So they are not gracefully degrading when synchrony decreases.

In [8], Guerraoui *et al.* determine the weakest failure detectors to boost obstruction freedom. In particular, [8] 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 process are timely, the non-timely processes can prevent all the timely processes from making progress [3].

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 [8] 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 TBWF 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 [8] uses atomic registers and assumes that all processes are timely.

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

3. 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 and abortable registers. In our model, time values are taken from the set $\mathbf{N}$ of positive integers.

Processes are (finite or infinite) deterministic automata that execute by taking steps. In each step, a process p can

[5]Recall that "timely" and "eventually timely" are equivalent when the timeliness bounds are not known and depend on each run.

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. Process p is allowed to invoke an operation on a register only if it received a response from its last invoked operation. 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. 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).*

Note that if p is timely in a run then p is correct in that run. Moreover, it is easy to show that a correct process is timely if and only if there exists an integer $i \geq 1$ such that, for every i consecutive process steps in R, there is at least one step of p.

We consider the following progress condition for object implementations in shared memory systems:

DEFINITION 3. *An object implementation is timeliness-based wait-free 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.*

Henceforth, 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 $k \in \mathbf{N}$ there is a time after which $v > k$.

4. THE DYNAMIC LEADER ELECTOR Ω_Δ

Intuitively, Ω_Δ is a dynamic leader election primitive 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 local *input* and *output* variables denoted CANDIDATE_p and LEADER_p, respectively. 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, and define the set *Timely*, 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* $\text{CANDIDATE}_q =$ ***false***$\}$.
- *Pcandidates* $= \{q : q$ *is correct and there is a time after which* $\text{CANDIDATE}_q =$ ***true***$\}$.
- *Rcandidates* $= \{q : q$ *is correct and* $\text{CANDIDATE}_q =$ ***true*** *infinitely often and* $\text{CANDIDATE}_q =$ ***false*** *infinitely often* $\}$.

Let Timely be the set of processes that are timely in R.

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

DEFINITION 5. *In every run R of Ω_Δ, the following properties hold:*

1. *If Pcandidates $\cap$ Timely $\neq \emptyset$ then there is a process $\ell \in$ (Pcandidates $\cup$ Rcandidates) $\cap$ Timely 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* $\text{LEADER}_p = ?$.

Note that the above specification of Ω_Δ states that the elected leader ℓ is in (*Pcandidates* $\cup$ *Rcandidates*) $\cap$ *Timely*, so ℓ may 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 now show that Ω_Δ can be used in a way that eliminates this problem.

DEFINITION 6. *We say that 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****.*

In the full paper, we show that when Ω_Δ is used in the canonical way, the leader ℓ elected by Ω_Δ is a process in *Pcandidates* $\cap$ *Timely*, i.e., a timely process that competes for leadership "forever":

THEOREM 7. *With a canonical use of Ω_Δ, the following properties hold (in every run R):*

1. *If $Pcandidates \cap Timely \neq \emptyset$ then there is a process $\ell \in Pcandidates \cap Timely$ 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* $\text{LEADER}_p = ?$.

COROLLARY 8. *With a canonical use of Ω_Δ, the following properties hold (in every run R):*

If $Pcandidates \cap Timely \neq \emptyset$ then there is a process $\ell \in Pcandidates \cap Timely$ such that
   (a) *There is a time after which* $\text{LEADER}_\ell = \ell$.
   (b) *For every correct process $p \neq \ell$, there is a time after which* $\text{LEADER}_p \neq p$.

5. 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 (Section 5.1). We then use activity monitors and registers to implement Ω_Δ (Section 5.2).

5.1 Definition and implementation of activity monitors

For any two processes p and q, a *(dynamic) activity monitor* $\mathcal{A}(p, q)$ is a primitive 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.

Process p tells $\mathcal{A}(p, q)$ to turn the monitoring of q on or off by writing *on* or *off* to a local variable $\text{MONITORING}_p[q]$ (which 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 local variable $\text{ACTIVE-FOR}_q[p]$ (which is also periodically read by $\mathcal{A}(p, q)$). If q is alive and $\text{ACTIVE-FOR}_q[p] = on$ 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 local output variables, denoted $\text{STATUS}_p[q]$ and $\text{FAULTCNTR}_p[q]$, at process p.

Intuitively, $\text{STATUS}_p[q] = active$, $inactive$ 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 $\text{FAULTCNTR}_p[q]$ 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 $\text{MONITORING}_p[q]$ and $\text{ACTIVE-FOR}_q[p]$ 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 specify $\mathcal{A}(p, q)$, we must define its output in all the above cases.

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

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

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

DEFINITION 9. *In every run R of $\mathcal{A}(p, q)$, if p is correct in R then the following properties hold:*

- $\text{STATUS}_p[q]$ *properties*
  1. *If there is a time after which* $\text{MONITORING}_p[q]{=}off$ *then there is a time after which* $\text{STATUS}_p[q]{=}?$.
  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* $\text{MONITORING}_p[q] = on$

Note the Property 6 is not the converse of Property 5 (for example, 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. The detailed algorithm code is given in Figure 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 stop-

CODE FOR MONITORED PROCESS q:

{ $\mathcal{A}(p,q)$-*Input* : ACTIVE-FOR[p] }

{ **Initial state** }

HbRegister$[q,p] = -1$ { shared register written by q and read by p }
hbCounter $= 0$ { local variable }

{ **Main code** }

```
1  repeat forever
2      WRITE(HbRegister[q,p], −1)
3      while ACTIVE-FOR[p] = off do skip
4      while ACTIVE-FOR[p] = on do
5          hbCounter ← hbCounter + 1
6          WRITE(HbRegister[q,p], hbCounter)
```

CODE FOR MONITORING PROCESS p:

{ $\mathcal{A}(p,q)$-*Input* : MONITORING[q] }
{ $\mathcal{A}(p,q)$-*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** }

```
7  repeat forever
8      STATUS[q] ← ?
9      while MONITORING[q] = off do skip
10     hbTimer ← hbTimeout
11     while MONITORING[q] = on do
12         if hbTimer ≥ 1 then hbTimer ← hbTimer − 1
13         if hbTimer = 0 then
14             hbTimer ← hbTimeout
15             prevHbCounter ← hbCounter
16             hbCounter ← READ(HbRegister[q,p])
17             if hbCounter < 0 then STATUS[q] ← inactive
18             if hbCounter ≥ 0 and hbCounter > prevHbCounter then
19                 STATUS[q] ← active
20                 allow_increment ← true
21             if hbCounter ≥ 0 and hbCounter ≤ prevHbCounter then
22                 STATUS[q] ← inactive
23                 if allow_increment then
24                     FAULTCNTR[q] ← FAULTCNTR[q] + 1
25                     hbTimeout ← hbTimeout + 1
26                     allow_increment ← false
```

Figure 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.

ping willingly (instead of crashing). When p does not monitor q, p sets STATUS$_p[q]$ to "?". When p monitors q, p checks if R increases periodically and, if so, p sets 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 STATUS$_p[q]$ to *inactive* and p may or may not increment FAULTCNTR$_p[q]$: p increments FAULTCNTR$_p[q]$ if (a) $R \neq -1$ *and* (b) R increased since the last time p incremented FAULTCNTR$_p[q]$. Condition (a) prevents FAULTCNTR$_p[q]$ from increasing forever if q intermittently stops being active for p, which is necessary to ensure part (c) of Property 5 above. Condition (b) prevents FAULTCNTR$_p[q]$ from increasing forever if q crashes, which is necessary to ensure part (b) of Property 5 above.

THEOREM 10. *For any pair of processes $p \neq q$, the algorithm in Figure 2 implements an activity monitor $\mathcal{A}(p,q)$ using registers.*[6]

5.2 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 Figure 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 FAULTCNTR$_p[q]$ and a status STATUS$_p[q]$. Process p uses FAULTCNTR$_p[q]$ to detect "bad" processes: if p sees that FAULTCNTR$_p[q]$ increases then p increments *CounterRegister*$[q]$. Process p uses the vector STATUS$_p$ to determine the set *activeSet*$_p$ of processes q with STATUS$_p[q]$ = *active*; p also includes itself in *activeSet*$_p$. Process p picks its leader as the process ℓ in *activeSet*$_p$ with smallest *CounterRegister*$[\ell]$. If p picks itself as leader then p sets $\mathcal{A}(p,q)$'s ACTIVE-FOR$_p[q]$ to *on* (for every process q). Otherwise, p sets 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 *leader*$_p$ to ?, MONITORING$_p[q]$ to *off* and ACTIVE-FOR$_p[q]$ to *off* for every process q. While CANDIDATE$_p$ = **false**, p does nothing. When p finds that CANDIDATE$_p$ = **true**, p sets 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 CANDIDATE$_p$ = **true**,

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

CODE FOR PROCESS p:

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

{ $\mathcal{A}(p,q)$-*Input* : MONITORING[q] }
{ $\mathcal{A}(p,q)$-*Output* : $\langle$STATUS[q], FAULTCNTR[q]$\rangle$ }

{ $\mathcal{A}(q,p)$-*Input* : ACTIVE-FOR[q] }

{ **Initial state** }

LEADER $= ?$
$\forall q \in \Pi$: MONITORING[q] $=$ *off* $\wedge$ ACTIVE-FOR[q] $=$ *off*
$\forall q \in \Pi$: *faultCntr*[q] $= 0$ $\wedge$ *maxFaultCntr*[q] $= 0$ { local variables }
$\forall q \in \Pi$: *counter*[q] $= 0$ { local variable }
activeSet $= \{p\}$ { local variables }
CounterRegister[p] $= 0$ { shared register }

{ **Main code** }

1 **repeat forever**
2 LEADER $\leftarrow ?$
3 **for each** $q \in \Pi$ **do** MONITORING[q] $\leftarrow$ *off*
4 **for each** $q \in \Pi$ **do** ACTIVE-FOR[q] $\leftarrow$ *off*
5 **while** CANDIDATE $=$ **false do skip**
6 **for each** $q \in \Pi$ **do** MONITORING[q] $\leftarrow$ *on*
7 *counter*[p] $\leftarrow$ READ(*CounterRegister*[p])
8 WRITE(*CounterRegister*[p], *counter*[p] $+ 1$)
9 **while** CANDIDATE $=$ **true do**
10 **for each** $q \in \Pi$ **do**
{ consult activity monitor $\mathcal{A}(p,q)$ about status of q }
11 **repeat** $\langle$*status*[q], *faultCntr*[q]$\rangle \leftarrow \langle$STATUS[$q$], FAULTCNTR[$q$]$\rangle$
until *status*[q] $\neq ?$
12 *activeSet* $\leftarrow \{q : q \in \Pi \wedge$ *status*[q] $=$ *active*$\} \cup \{p\}$
13 **for each** $q \in \Pi$ **do** *counter*[q] $\leftarrow$ READ(*CounterRegister*[q])
14 LEADER $\leftarrow \ell$ such that $(counter[\ell], \ell) = \min\{(counter[q], q) : q \in activeSet\}$
15 **if** LEADER $= p$ **then**
16 **for each** $q \in \Pi$ **do** ACTIVE-FOR[q] $\leftarrow$ *on*
17 **else for each** $q \in \Pi$ **do** ACTIVE-FOR[q] $\leftarrow$ *off*
18 **for each** $q \in \Pi$ **do**
19 **if** *faultCntr*[q] $>$ *maxFaultCntr*[q] **then**
20 WRITE(*CounterRegister*[q], *counter*[q] $+ 1$)
21 *maxFaultCntr*[q] $\leftarrow$ *faultCntr*[q]

Figure 3: Implementation of Ω_Δ using activity monitors and registers.

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 $activeSet_p$ 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 $activeSet_p$ with smallest $CounterRegister[\ell]$. If p picks itself, it sets ACTIVE-FOR$_p[q]$ to *on* otherwise it sets it to *off*, for every process q. Next, if p finds that FAULTCNTR$_p[q]$ increased then p increments $CounterRegister[q]$.

THEOREM 11. *The algorithm in Figure 3 implements Ω_Δ in a system with registers where every pair of processes (p,q) is equipped with an activity monitor $\mathcal{A}(p,q)$.*

From Theorems 10 and 11, we have

THEOREM 12. *The algorithm obtained by combining the algorithms in Figures 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".

6. IMPLEMENTING Ω_Δ USING ABORTABLE REGISTERS

We now show how to implement Ω_Δ using (single-writer single-reader) abortable registers.[7] An abortable register is a very weak object because its operations may abort if they are concurrent. 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 its 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 *msgToq*. To do so, whenever p sees that *msgToq* 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 the eventually p's write is successful. In fact, if *msgToq* stops changing, eventually p writes successfully the final value of *msgToq* 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 *msgToq* 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 Figure 4. There is a vector $MsgRegister[p,q]$ of abortable registers written by p and read by q, for every 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

[7] 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.

CODE FOR PROCESS p:

{ **Initial state** }

$\forall q \in \Pi - \{p\}$: *MsgRegister*$[p, q] = \langle 0, 0 \rangle$
{ abortable register written by p and read by q }
$\forall q \in \Pi - \{p\}$: *msgCurr*$[q] = \langle 0, 0 \rangle$
$\forall q \in \Pi - \{p\}$: *prevMsgFrom*$[q] = \langle 0, 0 \rangle$
$\forall q \in \Pi - \{p\}$: *readTimer*$[q] = 1$
$\forall q \in \Pi - \{p\}$: *readTimeout*$[q] = 1$
$\forall q \in \Pi - \{p\}$: *prevWriteDone*$[q]$ = **true**

1 **procedure** *WriteMsgs*(*msgTo*)
2 **for each** $q \in \Pi - \{p\}$ **do**
3 **if** (**not** *prevWriteDone*$[q]$) **or** *msgCurr*$[q] \neq$ *msgTo*$[q]$ **then**
4 **if** *prevWriteDone*$[q]$ **then** *msgCurr*$[q]$:= *msgTo*$[q]$
5 *res* $\leftarrow$ WRITE(*MsgRegister*$[p, q]$, *msgCurr*$[q]$)
6 *prevWriteDone*$[q] \leftarrow$ (*res* = *ok*)
7 **return** *prevWriteDone*

8 **procedure** *ReadMsgs*()
9 **for each** $q \in \Pi - \{p\}$ **do**
10 **if** *readTimer*$[q] \geq 1$ **then** *readTimer*$[q] \leftarrow$ *readTimer*$[q] - 1$
11 **if** *readTimer*$[q] = 0$ **then**
12 *readTimer*$[q] \leftarrow$ *readTimeout*$[q]$
13 *res*$[q] \leftarrow$ READ(*MsgRegister*$[q, p]$)
14 **if** *res*$[q] = \perp$ or *res*$[q]$ = *prevMsgFrom*$[q]$
15 **then** *readTimeout*$[q] \leftarrow$ *readTimeout*$[q] + 1$
16 **else**
17 *prevMsgFrom*$[q] \leftarrow$ *res*$[q]$
18 *readTimeout*$[q] \leftarrow 1$
19 **return** *prevMsgFrom*

Figure 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** }

$\forall q \in \Pi - \{p\}$: *HbRegister1*$[p, q] = 0$
{ abortable register written by p and read by q }
$\forall q \in \Pi - \{p\}$: *HbRegister2*$[p, q] = 0$
{ abortable register written by p and read by q }
$\forall q \in \Pi - \{p\}$: *hbTimeout*$[q] = 1$
$\forall q \in \Pi - \{p\}$: *hbTimer*$[q] = 1$
$\forall q \in \Pi - \{p\}$: *prevHbCounter1*$[q] = 0$
$\forall q \in \Pi - \{p\}$: *prevHbCounter2*$[q] = 0$
$\forall q \in \Pi - \{p\}$: *hbCounter1*$[q] = 0$
$\forall q \in \Pi - \{p\}$: *hbCounter2*$[q] = 0$
hbSendCounter $= 0$
activeSet $= \{p\}$

20 **procedure** *SendHeartbeat*(*dest*)
21 *hbSendCounter* $\leftarrow$ *hbSendCounter* $+ 1$
22 **for each** $q \in \Pi - \{p\}$ **do**
23 **if** *dest*$[q]$ **then**
24 WRITE(*HbRegister1*$[p, q]$, *hbSendCounter*)
25 WRITE(*HbRegister2*$[p, q]$, *hbSendCounter*)

26 **procedure** *ReceiveHeartbeat*()
27 **for each** $q \in \Pi - \{p\}$ **do**
28 **if** *hbTimer*$[q] \geq 1$ **then** *hbTimer*$[q] \leftarrow$ *hbTimer*$[q] - 1$
29 **if** *hbTimer*$[q] = 0$ **then**
30 *hbTimer*$[q] \leftarrow$ *hbTimeout*$[q]$
31 *prevHbCounter1*$[q] \leftarrow$ *hbCounter1*$[q]$
32 *prevHbCounter2*$[q] \leftarrow$ *hbCounter2*$[q]$
33 *hbCounter1*$[q] \leftarrow$ READ(*HbRegister1*$[q, p]$)
34 *hbCounter2*$[q] \leftarrow$ READ(*HbRegister2*$[q, p]$)
35 **if** (*hbCounter1*$[q] = \perp$ **or** *hbCounter1*$[q] \neq$ *prevHbCounter1*$[q]$) **and**
(*hbCounter2*$[q] = \perp$ **or** *hbCounter2*$[q] \neq$ *prevHbCounter2*$[q]$)
36 **then** *activeSet* $\leftarrow$ *activeSet* $\cup \{q\}$
37 **else**
38 *activeSet* $\leftarrow$ *activeSet* $- \{q\}$
39 *hbTimeout*$[q] \leftarrow$ *hbTimeout*$[q] + 1$
40 **return** *activeSet*

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

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 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 Figure 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 Figure 6, has some similarities with the algorithm

CODE FOR PROCESS p:

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

{ **Initial state** }

LEADER $= ?$
leader $= p$
$\forall q \in \Pi$: *counter*$[q] = 0$
$\forall q \in \Pi - \{p\}$: *actrTo*$[q] = 0$
$\forall q \in \Pi - \{p\}$: *writeDone*$[q]$ = **false**

{ **Main code** }

41 **repeat forever**
42 LEADER $\leftarrow ?$
43 **while** CANDIDATE = **false do skip**
44 *counter*$[p] \leftarrow \max\{$*counter*$[p]$, *counter*[*leader*$] + 1\}$
45 **do**
46 *SendHeartbeat*(*writeDone*)
47 *activeSet* $\leftarrow$ *ReceiveHeartbeat*()
48 *leader* $\leftarrow \ell$ such that
$(\text{counter}[\ell], \ell) = \min\{(\text{counter}[q], q) : q \in \text{activeSet}\}$
49 LEADER $\leftarrow$ *leader*
50 **for each** $q \in \Pi - \{p\}$ **do**
51 **if** $q \notin$ *activeSet*
52 **then** *actrTo*$[q] \leftarrow \max\{$*actrTo*$[q]$, *counter*[*leader*$] + 1\}$
53 *msgTo*$[q] \leftarrow \langle$*counter*$[p]$, *actrTo*$[q]\rangle$
54 *writeDone* $\leftarrow$ *WriteMsgs*(*msgTo*)
55 *msgFrom* $\leftarrow$ *ReadMsgs*()
56 **for each** $q \in \Pi - \{p\}$ **do**
57 $\langle$*counter*$[q]$, *actrFrom*$[q]\rangle \leftarrow$ *msgFrom*$[q]$
58 *counter*$[p] \leftarrow \max\{$*counter*$[p]$, *actrFrom*$[q]\}$
59 **while** CANDIDATE = **true**

Figure 6: Implementation of Ω_Δ using abortable registers — main code.

of Section 5.2: 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 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 Section 5.2, every time p becomes a candidate of Ω_Δ, it inflicts a "self-punishment". It does *not* do so by increasing $counter_p[p]$ (otherwise $counter_p[p]$ may never stop changing and thus *WriteMsgs* could not 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 obtain $activeSet_p$. 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. 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. Process p updates the $counter_p[q]$ vector using this information, and increases $counter_p[p]$ according to the punishments it received.

THEOREM 13. *The algorithm in Figures 4, 5, and 6 implements Ω_Δ in a system with abortable registers.*

7. USING Ω_Δ TO ACHIEVE TIMELINESS-BASED WAIT-FREEDOM

We now explain how Ω_Δ can be used to obtain a TBWF 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, QUERY can

CODE FOR PROCESS p:

{ **Initial state** }

CANDIDATE $=$ **false**
O_{QA}'s initial state = O's initial state { O_{QA} is an object of type T_{QA} }

{ **Main code** }

```
1  { procedure to execute an operation op of object O of type T }
   procedure invoke(op, O, T)
2     while LEADER = p do skip
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
```

Figure 7: TBWF implementation of any type T from its query-abortable counterpart T_{QA} and Ω_Δ.

also abort and return $\perp$. (A more precise 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 a timeliness-based wait-free implementation of O of type T, as shown in Figure 7. 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 Figure 8, until p is successful. The first operation is op (shown by the double circle). The corresponding response is either $\perp$ or a "normal" response $v \neq \perp$. If the result is a normal response then p is done. Otherwise, p is uncertain whether op took effect or not, so p next executes QUERY 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 invocation 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 invocation of op did not take effect, so p tries to execute op again. If, at any time, Ω_Δ 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 Ω_Δ, is crucial for obtaining an implementation that is timeliness-based wait-free. Without it, a timely process would be able to monopolize the access to the implemented object O: it would be able to execute an infinite sequence of operations on O and win every competition for leadership, 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 Ω_Δ 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.

THEOREM 14. *The algorithm in Figure 7 uses Ω_Δ to obtain a timeliness-based wait-free implementation of an arbitrary type T from a wait-free implementation of its query-abortable counterpart T_{QA}.*

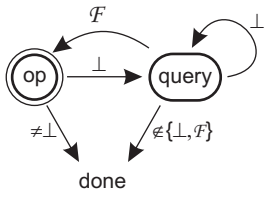

Figure 8: Sequence of operations invoked on object O_{QA} of type T_{QA} by the implementation in Figure 7.

Since abortable registers can be used to implement Ω_Δ (Theorem 13) and also to obtain a wait-free implementation of type T_{QA} for any type T [2], we conclude that

THEOREM 15. *Every type T has a timeliness-based wait-free implementation from abortable registers.*

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

8. REFERENCES

[1] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. On implementing Omega with weak reliability and synchrony assumptions. In *Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing*, pages 306–314. ACM, July 2003.

[2] M. K. Aguilera, S. Frolund, V. Hadzilacos, S. L. Horn, and S. Toueg. Abortable and query-abortable objects and their efficient implementation. In *Proceedings of the 26th ACM Symposium on Principles of Distributed Computing*, pages 23–32. ACM, Aug. 2007.

[3] M. K. Aguilera and S. Toueg. Timeliness-based wait-freedom: a gracefully degrading progress condition. Full paper, in preparation.

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

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

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

[7] F. E. Fich, V. Luchangco, M. Moir, and N. Shavit. Obstruction-free algorithms can be practically wait-free. In *Proceedings of the 19th International Symposium on Distributed Computing*, volume 3724 of *LNCS*, pages 78–92. Springer, Sept. 2005.

[8] R. Guerraoui, M. Kapalka, and P. Kouznetsov. The weakest failure detectors to boost obstruction-freedom. *Distributed Computing*, 20(6):415–433, Apr. 2008.

[9] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, Jan. 1991.

[10] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, pages 522–529. IEEE Computer Society, May 2003.

[11] G. Taubenfeld. Efficient transformations of obstruction-free algorithms into non-blocking algorithms. In *Proceedings of the 21st International Symposium on Distributed Computing*, volume 4731 of *LNCS*, pages 450–464. Springer, Sept. 2007.