Crash Fault Detection in Celerating Environments

Srikanth Sastry, Scott M. Pike, and Jennifer L. Welch
Department of Computer Science and Engineering
Texas A&M University
College Station, TX 77843-3112, USA
{sastry, pike, welch}@cse.tamu.edu

Abstract

Failure detectors are a service that provides (approximate) information about process crashes in a distributed system. The well-known "eventually perfect" failure detector, $\diamond\mathcal{P}$, has been implemented in partially synchronous systems with unknown upper bounds on message delay and relative process speeds. However, previous implementations have overlooked an important subtlety with respect to measuring the passage of time in "celerating" environments, in which absolute process speeds can continually increase or decrease while maintaining bounds on relative process speeds. Existing implementations either use action clocks, which fail in accelerating environments, or use real-time clocks, which fail in decelerating environments. We propose the use of bichronal clocks, which are a composition of action clocks and real-time clocks. Our solution can be readily adopted to make existing implementations of $\diamond\mathcal{P}$ robust to process celeration, which can result from hardware upgrades, server overloads, denial-of-service attacks, and other system volatilities.

1 Introduction

A *failure detector* can be viewed as a distributed oracle that can be queried for (potentially incorrect) information about process crashes. Despite such unreliability, failure detectors can solve many classic problems that are *not* solvable in crash-prone asynchronous message-passing systems (for example, fault tolerant consensus [5], and wait-free dining [21]).

Failure detectors were introduced by Chandra and Toueg in [5] to circumvent the impossibility of fault-tolerant consensus in asynchrony [12]. This seminal paper introduced a hierarchy of eight failure detector classes. Subsequent work introduced several other failure detector classes (*e.g.*, Heartbeat [1], Marabout [13], Omega [3], Accrual [15], and Trusting [6]).

This paper focuses on the *eventually perfect failure detector* – $\diamond\mathcal{P}$ – defined in the original Chandra-Toueg hierarchy. Informally, $\diamond\mathcal{P}$ can give arbitrary (incorrect) information about process crashes for a finite computational prefix. Eventually, however, it provides perfect information about process crashes. Note that time after which $\diamond\mathcal{P}$ starts providing perfect information is potentially unknown. More precisely, $\diamond\mathcal{P}$ satisfies the following properties [5]:

- **Strong Completeness**: Every crashed process is eventually and permanently suspected by every correct process.
- **Eventual Strong Accuracy**: For each run, there exists a time after which no correct process is suspected by any correct process.

The failure detector $\diamond\mathcal{P}$ is sufficiently powerful to solve many classic problems in distributed computing, including fault tolerant consensus [5], stable leader election [2], quiescent reliable communication [1], wait-free contention management [14], crash-locality-1 dining philosophers [20], and wait-free dining under eventual weak exclusion [21].

However, $\diamond\mathcal{P}$ cannot be implemented in purely asynchronous systems. It requires the underlying system to provide some temporal guarantees on communication and computation. Such systems are said to be *partially synchronous*.

1.1 Partial synchrony

As mentioned above, system models providing certain temporal guarantees on communication and computation are said to be *partially synchronous* ([7, 8]).

Among the various partially synchronous models defined in [7] and [8], the most popular ones are denoted $\mathcal{M}_1$ and $\mathcal{M}_2$ (denoted as such in [5]).

Informally stated, in model $\mathcal{M}_1$, there exist *unknown* bounds on relative process speeds and message delay, and these bounds hold *perpetually*. Model $\mathcal{M}_2$, on the other hand, states that there exist *known* bounds on relative process speeds and message delay, but these bounds hold only *eventually*. In addition to $\mathcal{M}_1$ and $\mathcal{M}_2$, there is a third popular model, denoted $\mathcal{M}_3$. The model $\mathcal{M}_3$, defined in [5], is derived from $\mathcal{M}_1$ and $\mathcal{M}_2$, and states that there exist *unknown* bounds on relative process speeds and message delay, and these bounds hold only *eventually*. The models $\mathcal{M}_1$, $\mathcal{M}_2$, and $\mathcal{M}_3$, are collectively referred to as $\mathcal{M}_*$.

1.2 Process Celeration

It is widely believed that $\diamond\mathcal{P}$ can be implemented in the $\mathcal{M}_*$ models. While this belief happens to be true, implementations of $\diamond\mathcal{P}$ in such models have overlooked an important subtlety with respect to measuring the passage of time in *celerating environments*, wherein absolute process speeds can continually increase and/or decrease while maintaining bounds on relative process speeds. Such traditional implementations of $\diamond\mathcal{P}$ can precipitate an infinite number of failure detector mistakes in celerating environments.

1.3 Contribution

We show that existing implementations of $\diamond\mathcal{P}$ in $\mathcal{M}_*$ models, although correct in static environments, fail to behave correctly in *celerating environments*. We define a new type of clock called *bichronal* clocks, and show that these clocks can be used to implement $\diamond\mathcal{P}$ in such celerating partially synchronous environments. Additionally, we discuss the advantages of bichronal clocks in improving scalability and performance of $\diamond\mathcal{P}$ during volatile periods like hardware upgrades, denial-of-service attacks, and server overloads.

2 Existing implementations of $\diamond\mathcal{P}$ in $\mathcal{M}_*$

There are several implementations of $\diamond\mathcal{P}$ in $\mathcal{M}_*$ and other models of partial synchrony in the existing literature [5, 19, 4, 17, 10, 2, 18]. These implementations are based on the deduction that upper bounds on relative process speeds and message delay translate to an (unknown) upper bound on end-to-end, and round trip, communication delay. The end-to-end communication delay is the duration between the send of a message and its receipt. Similarly, a round-trip communication delay is the duration between the send of a message, and the receipt of its corresponding ack. These bounds on communication delay are adaptively estimated using some form of adaptive timers. Such adaptive timers start with some initial estimate of the bound on end-to-end (or round trip) communication delay and eventually converge to the de facto bound, after which correct processes are never suspected as crashed.

These adaptive timers use one of two techniques to measure time locally:

- **Action Clocks:** One technique to measure the passage of time is to count the number of actions executed locally at each process. Action clocks accomplish this by incrementing their clock value by one every time the process takes a step.

- **Real-time Clocks:** Alternatively, each process is assumed to have a real-time local clock which measures time with bounded drift. In practice, these clocks are realized by hardware devices such as crystal oscillators. Note that the real-time clocks need not measure real time perfectly. They can tick at different rates at each process. The only requirement is that there exist some bound on their drift.

3 End-to-end communication delay in $\mathcal{M}_*$

Note that the correctness of implementations cited in Section 2 depends on the existence of bounds on end-to-end communication delay. While the existence of such bounds is fairly straightforward in non-celerating environments, demonstrating the same in celerating environments (where processes can accelerate and/or decelerate continually) is more subtle.

3.1 Non-celerating $\mathcal{M}_*$ environments.

In non-celerating $\mathcal{M}_*$ environments, the process speeds, by definition, do not accelerate or decelerate continually. In other words, there exist some (potentially unknown) upper and lower bounds on absolute process speeds. Given such bounds on absolute process speeds, in conjunction with an upper bound on message delay, demonstrating an upper bound on end-to-end communication delay is straightforward[1].

[1]Note that time can be measured either in real-time clock ticks or in action clock ticks. Consequently, we must demonstrate the upper bound on end-to-end message delay in both types of clocks.

Action Clocks. Since we know that there exists an upper bound on absolute process speeds in non-celerating environments, there exists an upper bound on the number of actions a process can execute while a message is in transit (note that there exists an upper bound on message delay). Additionally, sending and receiving a message is assumed to be atomic and hence requires exactly one local action each. Since there exists an upper bound on relative process speeds, there exists an upper bound on the number of actions executed at *each* process while a message is either being sent or received. In other words, there exists an upper bound on the number of action clock ticks while a message is being sent, is in transit, and is being received. That is, there exists an upper bound on end-to-end message delay as measured by an action clock.

Real-time Clocks. A lower bound on absolute process speeds in non-celerating environments implies an upper bound on the real-time duration for a message to be generated and sent, as well as an upper bound on the real-time duration to complete the receipt of the message. Since the upper bound on message delay is assumed, there exists an upper bound on end-to-end communication delay, as measured by real-time clocks.

3.2 Celerating $\mathcal{M}_*$ environments

Celerating environments allow processes to accelerate and/or decelerate continually. Hence, absolute process speeds for (live) processes may be unbounded in such environments. This makes demonstrating an upper bound on end-to-end communication delay problematic.

Consider end-to-end communication in three such $\mathcal{M}_*$ environments: *accelerating* environments, *decelerating* environments, and environments that are both accelerating and decelerating (denoted **-celerating*).

Accelerating environments. In accelerating $\mathcal{M}_*$ environments, processes may accelerate continually, but they do not decelerate continually. In other words, while there may be no upper bound on absolute process speeds, there exists a lower bound on absolute process speeds. This lower bound on absolute process speeds (coupled with the upper bound on relative process speeds and the upper bound on message delay) yields an upper bound on end-to-end communication delay in terms of real-time clock ticks. However, there exists no upper bound on end-to-end communication delay in terms of action clock ticks (detailed discussion in Section 4).

Decelerating environments. In decelerating $\mathcal{M}_*$ environments, processes may decelerate continually, but they do not accelerate continually. In other words, while there may be no lower bound on absolute process speeds, there exists an upper bound on absolute process speeds. This upper bound (coupled with the upper bound on relative process speeds and the upper bound on message delay) yields an upper bound on end-to-end communication delay in terms of action clock ticks. However, there exists no upper bound on end-to-end communication delay in terms of real-time clocks (detailed discussion in Section 4).

***-celerating environments.** In $\mathcal{M}_*$ environments where processes may accelerate and decelerate continually, there exists neither an upper bound, nor a lower bound, on absolute process speeds. This results in unbounded end-to-end communication delay in terms of both real-time clocks and action clocks.

Such unbounded end-to-end communication delay in terms of either real-time clocks, or action clocks, or both, in celerating environments is referred to as the *celeration problem*.

4 The celeration problem

In this section we describe the celeration problem by demonstrating the following:

- In accelerating $\mathcal{M}_*$ environments, the end-to-end communication delay of messages is unbounded when denominated in ticks of an action clock.
- In decelerating $\mathcal{M}_*$ environments, the end-to-end communication delay of messages is unbounded when denominated in ticks of a real-time clock.

4.1 Action clocks in accelerating environments

In accelerating $\mathcal{M}_*$ environments, the end-to-end communication delay is unbounded when denominated in ticks of an action clock. Informally, the argument is as follows: As processes accelerate, their action clocks tick faster. Consequently, there are an increasingly greater number of action-clock ticks per unit real-time. Therefore, although there exists an upper bound on real-time message delay, an action clock can tick an unbounded number of times while a message is in transit. Since message delay is unbounded in terms of action clock ticks, so is end-to-end communication delay.

More formally: Consider an accelerating $\mathcal{M}_*$ environment. Let time be measured locally by action clocks. For the purpose of contradiction, assume that there exists an upper bound on message delay as measured by an

action clock. Let such a bound be k. Note that the $\mathcal{M}_*$ environment guarantees an upper bound on message delay in real time. Let this bound be Δ time units. Since processes are continually accelerating, eventually (say, after real time t) process speeds exceed $\lceil \frac{k}{\Delta} \rceil$ actions per unit real time. Let some message m sent after time t experience a delay of Δ real-time units. The message delay for m measured in action clock ticks exceeds k (because $\lceil \frac{k}{\Delta} \rceil \cdot \Delta \geq k$). However, this contradicts our assumption that the upper bound on message delay as measured by an action clock does not exceed k. This implies that message delay measured in action clock ticks is unbounded. Therefore, in accelerating $\mathcal{M}_*$ environments, the end-to-end communication delay of messages is unbounded when denominated in ticks of an action clock.

4.2 Real-time clocks in decelerating environments

In decelerating $\mathcal{M}_*$ environments, the end-to-end communication delay is unbounded when denominated in ticks of a real-time clock. Informally, the argument is as follows: as processes decelerate, each action takes increasingly longer to execute. There is no upper bound on the time taken to execute an action. Since it takes at least one action to generate and to receive a message, there is no upper bound on the time taken to generate a message and to receive a message. Consequently, there is no upper bound on end-to-end communication delay.

More formally: Consider a decelerating $\mathcal{M}_*$ environment. Let time be measured locally by perfect real-time clocks[2]. For the purpose of contradiction, we assume that there exists an upper bound on real-time end-to-end communication delay. Let this bound be k real-time units. As processes decelerate, an increasingly greater duration of time elapses between consecutive steps. Eventually (say, after time t), the time between consecutive steps exceeds k. Since it requires at least one action to send or receive a message, after time t, generation of a message m takes longer than k real-time units. Consequently, the end-to-end communication delay for m exceeds k. This, however, contradicts our earlier assumption that end-to-end communication delay is bounded above by k. In other words, in decelerating $\mathcal{M}_*$ environments, the end-to-end communication delay of a message is unbounded when denominated in ticks of a real-time clock.

[2]Although it is not necessary for the real-time clocks to be perfect, we strengthen the clock specification in order to strengthen the negative result

5 Impact of the celeration problem

In this section, we discuss the impact of the celeration problem on failure detector correctness and system model specifications. As explained below, many existing implementations of failure detectors fail to behave correctly in celerating environments. This is symptomatic of how time is measured in these implementations. However, it is possible to circumvent this problem by either strengthening existing system models, or by adopting alternate system models. In fact, both approaches have been explored in recent work on failure detector implementations in partial synchrony.

5.1 Correctness of failure detector implementations

Recall from Section 2 that the existing implementations of failure detectors like $\diamond\mathcal{P}$ adaptively estimate the upper bound on end-to-end, or round-trip, communication delay. This upper bound is measured using either action clocks or real-time clocks. However, in celerating environments (as described in Section 4) the use of either clock (in isolation) is problematic.

Consider the $\diamond\mathcal{P}$ implementations that use action clocks. In runs where processes accelerate continually, although there exists an upper bound on end-to-end communication delay in real-time ticks, there is no upper bound on action-clock ticks. Hence, such implementations fail in accelerating environments.

On the other hand, consider the $\diamond\mathcal{P}$ implementations that use real-time clocks. In runs where processes decelerate arbitrarily, although there exists an upper bound on end-to-end communication delay when denominated in action-clock ticks, there is no upper bound on the number of real-time ticks. Hence, such implementations fail in decelerating environments.

That is, regardless of the choice of clock to measure time (either action clocks, or real-time clocks), these $\diamond\mathcal{P}$ implementations fail in celerating environments.

5.2 Existing approaches to failure detector correctness

Arguably, the celeration problem is not unknown to the research community. Recent work on failure detectors have addressed this problem by either (a) strengthening the system model to preclude certain celerating runs which would otherwise precipitate incorrect behavior, or (b) adopting alternate system models which encapsulate process celeration by subsuming restrictions

on process speeds in restrictions on composite system behavior. We briefly visit both the approaches.

5.2.1 Strengthening system models

Recently proposed models like the Finite Average Response Time (FAR) model [11] and the Average Delay/Drop (ADD) model [23] make additional assumptions on absolute process speeds to circumvent the celeration problem. For instance, the FAR model [11] assumes an upper bound on absolute process speeds[3]. This assumption precludes continual acceleration of processes and allows failure detector implementations in the FAR model to use action clocks which are immune to continual process deceleration. In contrast, the ADD model [23] assumes a lower bound on absolute process speeds. This assumption precludes continual deceleration of processes and allows failure detector implementations in the ADD model to use real-time clocks which are immune to continual process acceleration.

5.2.2 Alternate system models

Alternate system models have been proposed that encapsulate process celeration in alternate specifications of system behavior, thereby voiding the celeration problem. Specifically, such models do not explicitly restrict message delay or process speeds. Instead, they encapsulate restrictions on message delay and process speeds in restrictions on composite system behavior (described below). Notable examples include the Asynchronous Bounded-Cycle (ABC) model [22] and the Theta model [16] which allow processes to accelerate and/or decelerate continually.

The ABC model imposes a restriction on the ratio of the number of messages that can be exchanged between pairs of processes in certain "relevant" segments of an asynchronous execution. On the other hand, the Theta model imposes a restriction on the ratio of the end-to-end communication delay of messages that are simultaneously in transit. Note that the ratio of the number of messages exchanged between processes and the ratio of end-to-end communication delay experienced by messages is determined by message delay and process speeds. Therefore, any restriction on such composite system behavior ultimately imposes restrictions on message delay and process speeds.

[3]In fact, the FAR model assumes that there exists a lower bound on the time it takes to increment an integer. This assumption ensures that the action clock (referred to as "weak clock" in [11]) does not accelerate continually.

5.3 Summary

Existing $\diamond\mathcal{P}$ implementations in $\mathcal{M}_*$ and related models fail to behave correctly in celerating environments. Clearly, the celeration problem can be sidestepped in $\mathcal{M}_*$ models by imposing bounds on absolute process speeds. Alternatively, the celeration problem can be voided by abandoning $\mathcal{M}_*$ models in favor of models like the ABC model or the Theta model. But is either approach necessary to overcome the celeration problem? Is it possible to implement $\diamond\mathcal{P}$ in $\mathcal{M}_*$ models without any additional assumptions? We answer the latter question in the affirmative by implementing $\diamond\mathcal{P}$ in $\mathcal{M}_*$ without any additional assumptions.

6 Solving the celeration problem

In this section we introduce a new technique to overcome the celeration problem without assuming lower or upper bounds on absolute process speeds. Our solution is based on a composition of action clocks and real-time clocks and is motivated by the following observations: Message delay is bounded above in terms of real time. Therefore, there exists an upper bound on the number of ticks of a real-time clock while a message is in transit. Similarly, relative process speeds are bounded as well. Therefore, regardless of process celeration, there exists an upper bound on the number of local actions executed while messages that have been delivered at the recipient's receive buffer are being processed. In other words, there exists an upper bound on the number of action clock ticks in the duration it takes for a message to be processed. Therefore, running real-time clock timers when messages are in transit, and running action clock timers when messages are being processed, should make $\diamond\mathcal{P}$ implementations immune to the celeration problem. We explore this intuition in the remainder of this section.

6.1 System model

Before proceeding to the $\diamond\mathcal{P}$ implementation, we explicitly define the $\mathcal{M}_*$ system model under which the failure detector will be implemented. The specification of $\mathcal{M}_*$ models has been marginally modified (especially in the definition of global time) from [7] and [8] in order to simplify analysis. However, the specification still reflects the basic behavior of unknown bounds on message delay and relative process speeds which hold eventually.

- **Processes.** The system has a finite fixed set of processes Π. Processes execute actions in atomic

steps. In an atomic step, a process receives at most one message from each process, makes a state transition, and sends at most one message to each process.

- **Time.** We posit the existence of a continuous Newtonian global time[4] measured by a fictitious global clock. The processes do not have access to the global clock; it is merely a modeling device.

- **Runs.** A run consists of an infinite sequence of steps taken by processes while executing an algorithm. For terminating algorithms, processes are modeled as having reached a final state S_f with an infinite sequence of (no-op) steps thereafter so that the process transitions from S_f to S_f. Note that not all processes may execute an infinite sequence of steps. Only *correct*, *i.e.*, non-faulty, processes execute an infinite sequence of steps.

  In any given run, each atomic step taken by a process is associated with a unique global time instance called the *occurrence time*, which is the time at which the atomic step is executed. Such a sequence of occurrence times at each process is non-decreasing, and for every finite closed interval of time $[t_1, t_2]$ (where $t_1 \leq t_2$), there are only finitely many steps whose occurrence time is within that interval[5].

- **Faults.** In each run, processes are either *correct* or *faulty*. Correct processes execute actions according to their algorithm specification, and never fail. *Faulty* processes, on the other hand, fail after finite time. Processes can fail only by *crashing*, which occurs when a process ceases execution without warning and never recovers. Any process that is not crashed is considered to be *live*.

- **Channels.** Processes send and receive messages to each other through channels. Each process is assumed to be connected to all the processes in the system by bi-directional reliable channels.

- **Clocks.** Processes are assumed to have access to a local real-time clock which measures time in discrete integer valued steps with some unknown bound D on the drift rate[6]. A clock is said to have a drift rate of D, if the following holds true: for every interval of t global time units, the clock measures no less than $(\frac{t}{D})$ time units and no greater than $(t \cdot D)$ time units.

- **Message delay.** Channels are assumed to deliver messages at the recipient's receive buffer within some unknown bound Δ global time units on delay. We assume no lower bounds on message delay.

- **Message buffering delay.** Message buffering delay is the duration between the time that a message is delivered at the recipient's receive buffer and the time that the message is actually processed by the receiving process. The message buffering delay is assumed to be bounded above by B local actions at the receiving process, where B in unknown. In other words, if a message m arrives at a process p's receive buffer at time t, then m is processed by p within the next B local actions at p.

- **Relative process speeds.** The relative process speeds are assumed to be bounded above by (an unknown) Φ, *i.e.*, in the duration that a correct process executes Φ atomic steps, each correct process executes at least 1 atomic step. Note that absolute process speeds are not bounded, and processes may accelerate and/or decelerate continually while maintaining the bound on relative process speeds.

6.2 Bichronal clocks

We introduce a new clock called a *bichronal clock*. It is a composition of an action clock and a real-time clock. A bichronal clock has the following properties:

- **Composition:** A bichronal clock consists of an action clock a and a real-time clock r.

- **Two-dimensional Time:** The time on a bichronal clock is the vector $\langle a.time, r.time \rangle$ where $a.time$ is the value of the action clock, and $r.time$ the value of the real-time clock. These time components are independent and will not be ordered lexicographically.

Similarly, a bichronal timer consists of an action clock timer and a real-time clock timer. It counts down from a given value (a_t, r_t) where a_t is the starting value for the action clock timer, and r_t is the starting value for

[4]Traditionally, global time is assumed to be discrete. However, in celerating environments processes are allowed to accelerate arbitrarily. Modeling time as a discrete entity implies that accelerating processes can take multiple actions in a single time tick. This implication makes system analysis problematic. Hence, for the ease of understanding and analysis, we assume a continuous global time base.

[5]This specification rules out the possibility of accelerating processes exhibiting "Zeno behavior"'.

[6]Note that the local real-time clocks at each process tick independently. The clocks need not be synchronized, and there need not be a bound on the drift rate across all runs.

the real-time clock timer. A bichronal timer is said to have timed out iff both the action clock timer and the real-time clock timer have timed out. The pseudo-code for a bichronal timer is shown in Fig. 1.

```
class bichronalTimer()
 actionClockTimer a
 realtimeClockTimer r
1 :   method start (integer actionTime, realTime)
2 :     a.start(actionTime)
3 :     r.start(realTime)
4 :     upon (a.expire() and r.expire())
5 :       send timer expiry notification
6 :   end method
7 :   method stop ()
8 :     a.stop(); r.stop()
9 :   end method
```

Figure 1. Implementation of a bichronal timer.

Bichronal timers satisfy the following two lemmas which are based on their specification.

Lemma 6.1. *If a bichronal timer tmr starts with a value $(t_a, D \cdot t_r)$, then tmr runs for at least t_r global time units.*

Proof. If a bichronal timer tmr starts with a value $(t_a, D \cdot t_r)$, then it starts two timers concurrently: an action clock timer a with value t_a, and a real-time clock timer r with value $D \cdot t_r$. The timer tmr does not expire until both a and r expire. But r expires only after $D \cdot t_r$ real-time clock ticks. Therefore, tmr does not expire until $D \cdot t_r$ real-time clock ticks. However, r has a bound D on drift rate, hence r takes at least t_r global time units to tick $D \cdot t_r$ times. In other words, tmr runs for at least t_r global time units. □

Lemma 6.2. *If a bichronal timer tmr starts with a value $(t_a, D \cdot t_r)$, then tmr runs for at least t_a local actions.*

Proof. If a bichronal timer tmr starts with a value $(t_a, D \cdot t_r)$, then it starts two timers concurrently: an action timer a with value t_a, and a real-time clock timer r with value $D \cdot t_r$. The timer tmr does not expire until both a and r expire. But a expires after t_a action clock ticks. Therefore, tmr runs for at least t_a local actions. □

The idea of composing multiple time bases into a composite time base is not new. For instance, Fetzer and Raynal proposed such a composition in [9] called Elastic Vector time. Like a bichronal clock, elastic vector time is a composition of two scalar time bases: logical time and real time. However, the main distinction between the compositional techniques of elastic vector time and bichronal clocks is the following: In elastic vector time, the logical clocks are updated to the value of the real time clocks infinitely often (subject to certain conditions). On the other hand, in bichronal clocks, the two time bases (logical and real) run independently of each other and are not correlated in any way (except insofar as both time bases eventually move forward in non-faulty processes).

6.3 Implementing $\diamond\mathcal{P}$

In this section, we present an implementation of $\diamond\mathcal{P}$ in $\mathcal{M}_*$ using bichronal clocks. Let the set of processes in the $\mathcal{M}_*$ system be Π. At each process p in Π, the action system $\diamond\mathcal{P}$-$exec$ (in Fig. 2) for the local $\diamond\mathcal{P}$ module is executed for each process q that p monitors. Thus, each process p runs $|\Pi|$ instances of $\diamond\mathcal{P}$-$exec$.

The algorithm shown in Fig. 2 is a ping-ack based implementation of $\diamond\mathcal{P}$. It uses a bichronal timer tmr, and an initial estimate on the timeouts for the bichronal timer ($timerValue$). The algorithm starts by executing *Action 1* in which p sends a $ping$ to process q, starts the bichronal timer tmr, and sets the variable $phase$ to 1. If tmr expires before receiving an ack from q, *Action 2* is enabled. In *Action 2*, if $phase \neq 4$, then p restarts tmr and increments $phase$ else p suspects q as having crashed. If p receives an ack, *Action 1* is enabled. In *Action 1*, if q is suspected as having crashed, then it could be a false suspicion (because p just received an ack from q). Hence, p trusts q, and increments the estimate on timeouts on the bichronal clock. Process p then sends a new $ping$ message to q, and restarts tmr. In *Action 3*, p sends an ack for every ping that it receives from q.

7 Proof of correctness

The algorithm runs a ping-ack protocol between p and q. The ping-ack protocol consists of a sequence of ping-ack transactions, each of which consists of four phases: *ping-in-transit* phase, *ack-generation* phase, *ack-in-transit* phase, and *ack-processing* phase, in that order. The *ping-in-transit* phase begins when p sends a ping to q, and ends when the ping is delivered to q's receive buffer. The *ack-generation* phase begins when the ping is delivered to q's buffer, and ends when q sends an ack (for the ping) back to p. The *ack-in-transit* phase

```
service ◇P-exec(process q)
    bichronalTimer tmr                                  // Bichronal timer for adaptive timeout
    integer timerValue ← 1                              // Some initial estimate
    integer phase ← 0                                   // Each ping-ack transaction has 4 phases (1 thru 4). Phase 0 is the initial value
    initially trust q                                   // Start by trusting q as being correct
-------------------------------------------------------------------------------------------------
1 :  { (upon receive ⟨ack⟩ from q) or (phase = 0)} ⟶     Action 1
2 :      if ( suspect q)                                // False suspicion
3 :        trust q                                      // Trust upon receiving an ack
4 :        timerValue ← timerValue + 1                  // Increment timer value
5 :      send ⟨ping⟩ to q                               // Send a ping and wait for an ack
6 :      tmr.start(timerValue, timerValue)              // Start bichronal timer
7 :      phase ← 1                                      // Transit to Phase 1
-------------------------------------------------------------------------------------------------
8 :  { upon tmr.expire() } ⟶                            Action 2
9 :      if (phase = 4)                                 // Should have received an ack by the end of 4 phases
10 :       suspect q                                    // Suspect upon timer expiry
11 :     else
12 :       phase ← phase + 1                            // Transit to the next phase, i.e., from phase 1 to 2, 2 to 3, or from phase 3 to 4
13 :       tmr.start(timerValue, timerValue)            // Restart the bichronal timer in the new phase
-------------------------------------------------------------------------------------------------
14 : { upon receive ⟨ping⟩ from q } ⟶                   Action 3
15 :     send ⟨ack⟩ to q                                // Upon receiving a ping, send an ack
```

Figure 2. Implementation of $\diamond\mathcal{P}$ in $\mathcal{M}_*$ using a bichronal timer

begins when q sends the ack to p, and ends when the ack is delivered to p's receive buffer. The *ack-processing* phase begins when the ack is delivered to p's receive buffer, and ends when a new ping is sent to q.

Note that the *ping-in-transit* phase and the *ack-in-transit* phase can last at most Δ global time units each. It takes q at most B local actions to generate an ack after the ping has been delivered to its receive buffer. During this time, p can execute at most $B \cdot \Phi$ local actions each. Therefore, the *ack-generation* phase can last at most $B \cdot \Phi$ local actions at p. Since it takes at most B local actions at p to process an ack after it has been delivered to its receive buffer, the *ack-processing* phase can last at most B local actions at p.

Intuitively, in the algorithm in Fig. 2, the variable $phase$ strives to follow (lag behind) the corresponding 4 phases of each ping-ack transaction. Eventually (and this is demonstrated in the proofs), when $phase = 1$, the current ping-ack transaction is in the *ping-in-transit* phase or later; when $phase = 2$, the current ping-ack transaction is in the *ack-generation* phase or later; when $phase = 3$, the current ping-ack transaction is in the *ack-in-transit* phase or later; and when $phase = 4$, the current ping-ack transaction is in the *ack-processing* phase, or has already terminated.

In order to prove correctness, we have to show that the algorithm in Fig. 2 satisfies the *strong completeness* and *eventual strong accuracy* properties of $\diamond\mathcal{P}$. The intuitive basis for the correctness is as follows:

Strong Completeness. If p is correct and q crashes, then eventually q does not send an ack in response to p's ping. When this happens, the bichronal timer tmr eventually expires, the $phase$ variable is incremented, and tmr is restarted. But eventually the $phase$ variable is incremented to 4, and when the bichronal timer expires while $phase$ equals 4, p starts suspecting q. Since p never receives an ack from q in the future, p continues to suspect q permanently thereafter. Since p and q can be any pair of correct and faulty processes, respectively, it follows that every correct process eventually and permanently suspects every crashed process.

Eventual Strong Accuracy. On the other hand, if both p and q are correct, then p may falsely suspect q as having crashed. However, eventually p receives an ack from q (because q is correct, and hence sends acks to p's pings). If p receives an ack while it suspects q, then p increments its bichronal timer value (in *line 4* of *Action 1*). Hence, after $max(D \cdot \Delta, B \cdot \Phi)$ such false positives, the timer value increments to $max(D \cdot \Delta, B \cdot \Phi) + 1$. Refer to this time as t_{con} (for convergence).

Note that tmr does not expire until the real-time clock timer expires, which is at least $D \cdot \Delta$ real-time clock ticks, which is at least Δ global time units. Similarly, tmr does not expire until the action clock timer expires, which is at least $B \cdot \Phi$ local actions.

Consider a ping sent from p to q after time t_{con}. When the bichronal timer tmr starts for the first time, $phase = 1$, and the ping-ack transaction is in the *ping-in-transit* phase. The timer tmr runs for at least Δ global time units. Therefore, when tmr expires while $phase = 1$, the ping would have been delivered at q (because Δ is the upper bound on message delay). In other words, by the time tmr expires, the ping-ack transaction is in the *ack-generation* phase or later. Therefore, when tmr expires for the first time, the ping-ack transaction is in the *ack-generation* phase or later

If tmr expires while $phase = 1$, p increments $phase$ to 2, and restarts tmr. Since tmr runs for at least $B \cdot \Phi$ local actions at p, process q executes at least B actions before timer expiry, which is a sufficient number of actions for q to generate and send the ack (if it has not done so already). In other words, by the time tmr expires for the second time, the ping-ack transaction is in *ack-in-transit* phase or later.

If tmr expires while $phase = 2$, p increments $phase$ to 3, and restarts tmr. Again, tmr runs for at least Δ global time units, which is sufficient time for the ack to be delivered at p's receive buffer (if it has not already been delivered). In other words, by the time tmr expires for the third time, the ping-ack transaction is in the *ack-processing* phase or later.

If tmr expires while $phase = 3$, p increments $phase$ to 4, and restarts tmr. Again, tmr does not expire until p executes $B \cdot \Phi$ actions. However, p takes receipt of the ack in its receive buffer within B local actions (if it has not done so already). Therefore, p receives the ack for the ping before tmr expires for the forth time (when $phase = 4$). Hence, p does not suspect q for pings sent after time t_{con}.

In other words, p may falsely suspect q only finitely many times after which it never suspects q. This holds true for all pairs of correct processes p and q. Thus, the algorithm in Fig. 2 satisfies eventual strong accuracy.

The formal proof is given next.

7.1 Strong completeness

Theorem 7.1. *The algorithm in Fig. 2 satisfies* strong completeness.

Proof. Recall that the strong completeness property states that every crashed process is eventually and permanently suspected by all correct processes.

Consider a run of the algorithm in Fig. 2 where process p is correct and process q is faulty. Let q crash at time t. At time t, one of the following four cases holds: (a) a ping is in transit from p to q, or (b) q has received a ping from p, but has not executed *Action 3*, or (c) an ack is in transit from q to p, or (d) p has received an ack from q but has not executed *Action 1* since.

Cases (a) and (b). Process p never receives an ack from q in the future (because q is crashed). Consequently, the bichronal timer tmr at p eventually expires, $phase$ is incremented, and tmr is restarted. Eventually, $phase$ is set to 4. When $phase$ equals 4 and the bichronal clock expires, p starts suspecting q. Since no ack from q arrives in the future, *Action 1* is never enabled and *line 3* in *Action 1* is never executed. Therefore, p suspects q permanently thereafter.

Cases (c) and (d). Process p eventually receives (or already has received) an ack from q. This enables *Action 1*, and a ping is sent from p to q. This reduces to *Case (a)* (which has been analyzed in the previous paragraph).

Hence, if a process q crashes, then a correct process p eventually and permanently suspects q. Since p and q can be any pair of correct and faulty processes, respectively, it follows that every crashed process is eventually and permanently suspected by all correct processes. □

7.2 Eventual strong accuracy

Recall that eventual strong accuracy states that for each run, there exists a time after which no correct process is suspected by any correct process. Consider two correct processes p and q in a run α of the algorithm in Fig. 2.

Lemma 7.2. *If p sends a ping to q at global time t_1, where $timerValue \geq \lceil max(B \cdot \Phi, D \cdot \Delta) + 1 \rceil$, and the bichronal timer tmr expires when $phase = 1$ at global time t_2 in the current ping-ack transaction, then at time t_2 the ping-ack transaction is in the ack-generation phase or later.*

Proof. If p sends a ping to q at global time t_1, then $phase = 1$ at t_1 (from *Action 1* in Fig. 2). Let $timerValue \geq \lceil max(B \cdot \Phi, D \cdot \Delta) + 1 \rceil$ at t_1. From *Action 1* in Fig. 2 we know that the bichronal timer tmr starts with value $(timerValue, timerValue)$ when p sends a ping to q. If tmr expires when $phase = 1$ at time t_2, then from Lemma 6.1 we know that tmr ran for at least Δ global time units. Recall that the upper bound on message delay is Δ global time units. There-

fore, by time t_2 the ping sent by p at time t_1 has already been delivered to q's receive buffer (and may or may not have been processed by q). Note that the *ack-generation* phase starts when the ping is delivered to q's receive buffer. Therefore, at time t_2 the ping-ack transaction is in the *ack-generation* phase or later. □

Lemma 7.3. *If p sends a ping to q at global time t_1, where $timerValue \geq \lceil max(B \cdot \Phi, D \cdot \Delta) + 1 \rceil$, and the bichronal timer tmr expires when $phase = 2$ at global time t_3 in the current ping-ack transaction, then at time t_3 the ping-ack transaction is in the ack-in-transit phase or later.*

Proof. If p sends a ping to q at global time t_1, then $phase = 1$ at t_1 (from *Action 1* in Fig. 2). Let the bichronal timer tmr expire when $phase = 1$ at time t_2. From Lemma 7.2, we know that at time t_2 the ping-ack transaction is in the *ack-generation* phase or later. From *Action 2* in Fig. 2, we know that after time t_2 the variable $phase = 2$, and tmr is restarted with value $(timerValue, timerValue)$. Since $timerValue > B \cdot \Phi$, if tmr expires when $phase = 2$ at time t_3, then from Lemma 6.2 we know that tmr ran for at least $B \cdot \Phi$ local actions at p after t_2. In other words, p executed at least $B \cdot \Phi$ actions after the ping-ack transaction was in *ack-generation* phase. Since the upper bound on relative process speeds is Φ, during the time that p executes $B \cdot \Phi$ actions, q executes at least B actions. In other words, when tmr expired at time t_3, process q had executed at least B actions after the ping was delivered to q's receive buffer. Therefore by time t_3, process q had already processed the ping (because a message in the receive buffer is processed within B local actions) and sent the ack. Note that the *ack-in-transit* phase starts when the ack is sent from q to p. Therefore, at time t_3, the ping-ack transaction is in the *ack-in-transit* phase or later. □

Lemma 7.4. *If p sends a ping to q at global time t_1, where $timerValue \geq \lceil max(B \cdot \Phi, D \cdot \Delta) + 1 \rceil$, and the bichronal timer tmr expires when $phase = 3$ at global time t_4 in the current ping-ack transaction, then at time t_4 the ping-ack transaction is in the ack-processing phase or later.*

Proof. If p sends a ping to q at global time t_1, then $phase = 1$ at t_1 (from *Action 1* in Fig. 2). Let the bichronal timer tmr expire when $phase = 2$ at time t_3. From Lemma 7.3, we know that at time t_3 the ping-ack transaction is in the *ack-in-transit* phase or later. In other words, q sent an ack to p in the current ping-ack transaction at time t_3 or earlier. From *Action 2* in Fig. 2, we know that after time t_3 the variable $phase = 3$, and tmr is restarted with value $(timerValue, timerValue)$. Since $timerValue > D \cdot \Delta$, if tmr expires when $phase = 3$ at time t_4, then from Lemma 6.1 we know that tmr ran for at least Δ global time units. Recall that the upper bound on message delay is Δ global time units. Therefore, by time t_4 the ack sent by q (at time t_3 or earlier) has already been delivered to p's receive buffer (and may or may not have been processed by p). Note that the *ack-processing* phase starts when the ack is delivered to p's receive buffer. Therefore, at time t_4 the ping-ack transaction is in the *ack-processing* phase or later. □

Lemma 7.5. *If p sends a ping to q at global time t_1, where $timerValue \geq \lceil max(B \cdot \Phi, D \cdot \Delta) + 1 \rceil$, and $phase = 4$ and bichronal timer tmr is unexpired at time t_5 in the current ping-ack transaction, then p receives the ack from q before tmr expires.*

Proof. If p sends a ping to q at global time t_1, then $phase = 1$ at t_1 (from *Action 1* in Fig. 2). Let the bichronal timer tmr expire when $phase = 3$ at time t_4. From Lemma 7.4, we know that at t_4 the ping-ack transaction is in the *ack-processing* phase or later. In other words, the ack from q was delivered to p's buffer at time t_4 or earlier. From *Action 2* in Fig. 2, we know that after t_4 the variable $phase = 4$, and tmr is restarted with value $(timerValue, timerValue)$. Since $timerValue > B \cdot \Phi$, we know from Lemma 6.2 that if tmr expires, then it will expire after at least $B \cdot \Phi$ actions at p. However, p processes messages in its receive buffer within B actions after they are delivered. Therefore, p takes receipt of the ack from q before tmr expires. □

Theorem 7.6. *The algorithm in Fig. 2 satisfies eventual strong accuracy*

Proof. Consider a run α of the algorithm in Fig. 2. Let p and q be two correct processes in this run, and let p send a ping to process q at time t. If p's timer tmr expires 4 times successively after time t but before receiving an ack from q, then p suspects q. However, since q is correct, q will eventually send an ack for each ping, and that ack will eventually be received by p. Therefore, every time p suspects q, p eventually receives an ack from q and stops suspecting q (*Action 1* in Fig. 2).

Consider the following metric $m(timerValue) = \lceil max(B \cdot \Phi, D \cdot \Delta) \rceil + 1 - timerValue$. Every time p receives an ack from q while q is suspected, p increments its $timerValue$ (line 4, *Action 1* in Fig. 2). Thus, every such false suspicion decrements the metric $m(timerValue)$. If $m(timerValue)$ never becomes 0

in α, then it implies that p suspects q only finitely many times.

On the other hand, if $m(timerValue)$ becomes 0 in α, it implies $timerValue = \lceil max(B \cdot \Phi, D \cdot \Delta) \rceil + 1$. This happens after $\lceil max(B \cdot \Phi, D \cdot \Delta) \rceil$ false suspicions. Let $m(timerValue) = 0$ at time t_{per} (after which $\diamond\mathcal{P}$ provides *perfect* information as demonstrated next). Consider any time $t_{suff} > t_{per}$ (in the *suffix*) at which p sends a ping to q.

Upon sending a ping to q, p starts tmr with timer value $tmr \geq \lceil max(B \cdot \Phi, D \cdot \Delta) \rceil + 1$ and $phase = 1$. Every time tmr expires before p receives an ack from q, it increments $phase$ and restarts tmr until $phase = 4$.

If p receives an ack from q before $phase = 4$, then p does not suspect q. From Lemma 7.5, we know that if $phase = 4$, $timerValue \geq \lceil max(B \cdot \Phi, D \cdot \Delta) \rceil + 1$, and tmr is unexpired, then p receives the ack from q before tmr expires. This enables *Action 1* and p does not suspect q. Therefore, if $m(timerValue) = 0$ in α, then p does not suspect q in an infinite suffix of α.

Thus, for any two correct processes p and q, p suspects q only finitely many times. In other words, there exists an unknown time after which no correct process is suspected by any correct process. □

Theorem 7.7. *The algorithm in Fig. 2 implements $\diamond\mathcal{P}$.*

Proof. From Theorems 7.1 and 7.6, we know that the algorithm in Fig. 2 satisfies *strong completeness* and *eventual strong accuracy*, thus implementing $\diamond\mathcal{P}$. □

8 Conclusion

Portability. The novelty in this paper is the introduction of bichronal clocks. In the implementation in Fig. 2, bichronal timers are used like action clock and real-time clock timers. Thus, existing implementations of $\diamond\mathcal{P}$ can be modified to use bichronal clocks instead of either action or real-time clocks, and automatically be made immune to the celeration problem. This is indicative of the portability and universal applicability of bichronal clocks in $\mathcal{M}_*$ and, possibly, other customized models of partial synchrony.

Heartbeat-based implementations. Although we have demonstrated the use of bichronal timers only in a ping-ack based $\diamond\mathcal{P}$ implementation, bichronal timers may be used in heartbeat-based $\diamond\mathcal{P}$ implementations too (*e.g.* [2, 23, 5, 4]). In heartbeat-based implementations, the local failure detector module at each process p sends periodic "heartbeats" to all the processes that monitor p. The monitoring processes then adaptively estimate the inter-arrival time of these heartbeats[7].

In heartbeat-based implementations, each process p will need at least one bichronal timer (or possibly more, depending on the details of the implementation) to determine the time to send the next set of heartbeats to the processes monitoring p. This bichronal timer is run exactly once per heartbeat generation. Process p will also need at least one bichronal timer to estimate the maximum inter-arrival time of heartbeats from the processes being monitored by p. The second bichronal timer would be run at most twice (successively) per heartbeat inter-arrival because the inter-arrival delay consists of two distinct delays: (1) message delay in transit, and (2) processing delay at the recipient's receive buffer.

Performance. In practice, it is often assumed that physical limitations (such as the speed of light, or the size of an atom) impose an upper bound on absolute process speed. Therefore, it can be argued that assuming an upper bound on process speeds and using real-time clocks alone is sufficiently realistic. While this may be true, bichronal clocks, in addition to correctness, boost performance of $\diamond\mathcal{P}$ as well.

In traditional action-clock based implementations of $\diamond\mathcal{P}$, every time hardware is upgraded, processes accelerate. Hence, the $\diamond\mathcal{P}$ module could start falsely timing out on correct processes resulting in longer time to convergence and increased system volatility. Using bichronal-clock based implementations eliminates this behavior because although the action clock component of the bichronal clock may falsely timeout on hardware upgrades, the real-time clock does not.

Similarly, in traditional real-time clock based implementations of $\diamond\mathcal{P}$, during periods of increased stress like server overloads, denial-of-service attacks, and such, processes in the system decelerate globally. The $\diamond\mathcal{P}$ module could start falsely timing out on correct processes due to poor response times from processes. Using bichronal-clock based implementations eliminates this behavior because although the real-time clock component of the bichronal clock may falsely timeout, the ac-

[7] Heartbeat-based implementations are often employed in systems where messages are lost infinitely often. Message loss becomes problematic in ping-ack based implementations because these implementations will have to take into account the possibility of pings and/or acks being dropped by the system. Heartbeat-based implementations are inherently immune to message loss. Such immunity, however, comes at a price. Ping-ack schemes are inherently flow-controlled. A ping-ack protocol typically adapts to varying process speeds better than heartbeat-based protocols. Faster processes are forced to wait for an ack after sending the ping, thus ensuring message flow-control in the system. In contrast, it is common in heartbeat-based protocols for faster processes to flood the slower processes with heartbeats.

tion clock does not.

Thus, in addition to providing immunity from process celeration, bichronal clocks also enable systems to minimize the volatility precipitated by system upgrades and periods of high stress.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable comments and suggestions which have been incorporated in the document. This work was supported in part by the Advanced Research Program of the Texas Higher Education Coordinating Board under Project Numbers 000512-0007-2006 and 000512-0130-2007 and by the National Science Foundation under grant DMIi-0500265.

References

[1] M. K. Aguilera, W. Chen, and S. Toueg. On quiescent reliable communication. *SIAM Journal on Computing*, 29(6):2040–2073, 2000.

[2] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Stable leader election. In *Proceedings of the 15th International Symposium on Distributed Computing*, pages 108–122, 2001.

[3] 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 Annual Symposium on Principles of Distributed Computing*, pages 306–314, 2003.

[4] M. Bertier, O. Marin, and P. Sens. Implementation and performance evaluation of an adaptable failure detector. In *Proceedings of the 32nd International Conference on Dependable Systems and Networks*, pages 354–363, 2002.

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

[6] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, and P. Kouznetsov. Mutual exclusion in asynchronous systems with failure detectors. *Journal of Parallel and Distributed Computing*, 65(4):492–505, 2005.

[7] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, 1987.

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

[9] C. Fetzer and M. Raynal. Elastic vector time. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, pages 284–291, 2003.

[10] C. Fetzer, M. Raynal, and F. Tronel. An adaptive failure detection protocol. In *Proceedings of the 7th Pacific Rim International Symposium on Dependable Computing*, pages 146–153, 2001.

[11] C. Fetzer, U. Schmid, and M. Süsskraut. On the possibility of consensus in asynchronous systems with finite average response times. In *Proceedings of the 25th International Conference on Distributed Computing Systems*, pages 271–280, 2005.

[12] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[13] R. Guerraoui. On the hardness of failure-sensitive agreement problems. *Information Processing Letters*, 79(2):99–104, 2001.

[14] R. Guerraoui, M. Kapalka, and P. Kouznetsov. The weakest failure detectors to boost obstruction-freedom. In *Proceedings of the 20th International Symposium on Distributed Computing*, pages 399–412, 2006.

[15] N. Hayashibara, X. Défago, and a. T. K. Rami Yared. The ϕ accrual failure detector. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems*, pages 66–78, 2004.

[16] J.-F. Hermant and J. Widder. Implementing reliable distributed real-time systems with the θ-model. In *Proceedings of the 9th International Conference on the Principles of Distributed Systems*, pages 334–350, 2005.

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

[18] M. Larrea and A. Lafuente. Communication-efficient implementation of failure detector classes $\diamond\mathcal{P}$ and $\diamond\mathcal{Q}$. In *Proceedings of the 19th International Symposium on Distributed Computing*, pages 495–496, 2005.

[19] A. Mostéfaoui, E. Mourgaya, and M. Raynal. Asynchronous implementation of failure detectors. In *Proceedings of the 33rd International Conference on Dependable Systems and Networks*, pages 351–360, 2003.

[20] S. M. Pike and P. A. G. Sivilotti. Dining philosophers with crash locality 1. In *Proceedings of the 24th International Conference on Distributed Computing Systems*, pages 22–29, 2004.

[21] S. M. Pike, Y. Song, and S. Sastry. Wait-free dining under eventual weak exclusion. In *Proceedings of the 9th International Conference on Distributed Computing and Networking*, pages 135–146, 2008.

[22] P. Robinson and U. Schmid. The Asynchronous Bounded-Cycle Model. In *Proceedings of the 10th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, pages 246–262, 2008.

[23] S. Sastry and S. M. Pike. Eventually perfect failure detection using add channels. In *Proceedings of the 5th international Symposium on Parallel and Distributed Processing and Applications*, pages 483–496, 2007.