Eventually Perfect Failure Detectors using ADD Channels

Srikanth Sastry and Scott M. Pike*

Texas A&M University
Department of Computer Science
College Station, TX 77843-3112, USA
{sastry, pike}@cs.tamu.edu

Abstract. We present a novel implementation of the *eventually perfect failure detector* ($\diamond\mathcal{P}$) from the original hierarchy of Chandra-Toueg oracles. Previous implementations of $\diamond\mathcal{P}$ have assumed models of partial synchrony where point-to-point message delay is bounded and/or communication is reliable. We show how to implement this important oracle under even weaker assumptions using Average Delayed/Dropped (ADD) channels. Briefly, all messages sent on an ADD channel are *privileged* or *non-privileged*. All non-privileged messages can be arbitrarily delayed or even dropped. For each run, however, there exists an unknown window size w, and two *unknown* upper-bounds d and r, where d bounds the average delay of the last w privileged messages, and r bounds the ratio of non-privileged messages to privileged messages per window.

Key words: Failure Detectors, Partial Synchrony, Communication Models

1 Introduction

A *failure detector* can be viewed as a distributed oracle that can be queried for (potentially unreliable) information about process crashes. Unreliable oracles can make mistakes by wrongfully suspecting correct processes, and/or not suspecting crashed processes. Despite such mistakes, many oracles are sufficiently powerful to solve important problems that are *not* solvable in crash-prone asynchronous systems. For example, the hierarchy of Chandra-Toueg oracles [1] was originally introduced to circumvent impossibility results for fault-tolerant consensus [2].

Oracle-based algorithms achieve an essential separation of concerns between detection properties and detection mechanisms. To be implemented in practice, most failure detector classes require some degree of partial or even full synchrony. The timing assumptions for fault detection, however, are encapsulated by the oracle abstraction. Since oracle-based algorithms depend only on the assertional *properties* of fault detection, they are effectively decoupled from the underlying implementation mechanisms and network timing parameters.

* This work was supported by the Advanced Research Program of the Texas Higher Education Coordinating Board under Project Number 000512-0007-2006.

This separation of concerns has spawned two basic lines of research. One examines the weakest detection properties sufficient for solving fundamental problems using oracles. The other examines increasingly weaker models of computation for implementing oracles in real systems. These two trajectories are complementary. The former extends our knowledge of relative solvability, while the latter addresses the practical implications of implementing such oracles, typically for providing fault-detection capabilities as a system service.

We contribute to the second line of research by defining a novel implementation of the *eventually perfect failure detector* $\diamond\mathcal{P}$ from the original Chandra-Toueg hierarchy [1]. Informally, $\diamond\mathcal{P}$ can give arbitrarily unreliable information about process crashes for a finite computation prefix of unknown length. Eventually, however, it provides perfect information about crash faults. Unfortunately, the time to convergence is not known, so it is not generally decidable whether the output of $\diamond\mathcal{P}$ during a given segment of computation is reliable or not. More precisely, $\diamond\mathcal{P}$ satisfies the following two properties [1]:

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

The oracle $\diamond\mathcal{P}$ is of interest for two primary reasons: it is relatively powerful, and yet realistically implementable. First, $\diamond\mathcal{P}$ is sufficiently powerful to solve many fundamental problems which are otherwise unsolvable without some recourse to fault detection. $\diamond\mathcal{P}$ is more than sufficient to solve fault-tolerant consensus [1], but its computational power is better illustrated by its sufficiency for harder problems like stable leader election [3], quiescent reliable communication [4], wait-free non-blocking contention management [5], wait-free eventual weak exclusion [6], crash-locality-1 dining philosophers [7], and wait-free eventually k-bounded schedulers under eventual weak exclusion [8].

Beyond its theoretical significance, $\diamond\mathcal{P}$ is also realistically implementable. Among other relatively powerful oracles — such as the Perfect ($\mathcal{P}$) [1], Strong ($\mathcal{S}$) [1], and Marabout [9] detectors — $\diamond\mathcal{P}$ is the only oracle implementable in partially synchronous systems. This result is from [10], where Larrea, *et al.*, prove that failure detectors with perpetual accuracy (including $\mathcal{P}$, $\mathcal{S}$, and Marabout) cannot be implemented in classical models of partial synchrony [1, 11]. As such, $\diamond\mathcal{P}$ has both theoretical as well as practical importance, insofar as it can solve important fault-tolerant problems, while being implementable in systems subject to timing uncertainties characterized by partial synchrony.

Ideally, we would like to know the weakest system model for implementing $\diamond\mathcal{P}$. This challenge has dual significance, because it underpins the fundamental solvability and portability of $\diamond\mathcal{P}$-based algorithms. Accordingly, the contributions of this paper are twofold: (1) we articulate a new, weaker model of partially synchronous communication called Average Delayed/Dropped (ADD) Channels, and (2) we implement $\diamond\mathcal{P}$ in the ADD model, which permits unbounded message loss and unbounded point-to-point message delay. Our results characterize one of the weakest models to date for implementing $\diamond\mathcal{P}$.

2 Motivation

Classical models of partial synchrony ([1, 3, 12–17]) for implementing $\diamond\mathcal{P}$ make certain assumptions about the reliability and timeliness of the underlying communication channels. The models cited above assume that the communication channels are either always reliable[1], or eventually reliable (i.e., can lose at most finitely-many messages over some prefix, followed by an infinite reliable suffix). Additionally, these models assume the existence of upper-bounds (known or unknown) on point-to-point message delay, or average message delay.

In many systems, however, message loss and/or delays occur intermittently throughout the computation. For instance, consider a communication system with (1) an unknown upper-bound on channel delay, (2) bounded buffers at each in-bound channel interface, and (3) a load-shedding policy (such as *milk* or *wine*) for congestion control. Suppose process p persistently sends messages to q at a rate faster than q can process them. In finite time, the in-bound buffer at q becomes full and activates the load-shedding policy: for milk, buffered messages are dropped in favor of (fresh) arriving messages; for wine, arriving messages are dropped in favor of (aged) buffered messages. For every message that arrives when the buffer at q is full, some message is lost. However, q processes messages infinitely often, so infinitely many messages are also delivered reliably. Upper-bounds on channel delay and buffer delay translate into an upper-bound on ene-to-end message delay. Therefore, an infinite subset of messages are delivered within some bounded delay, but an infinite subset of them are also dropped.

Existing implementations of $\diamond\mathcal{P}$ can be trivially adapted to withstand certain subsets of messages being arbitrarily delayed or dropped. For instance, consider a system E where only the odd-numbered messages may be delayed or dropped, but all the even-numbered messages are delivered reliably within some unknown bound on delay. Implementing $\diamond\mathcal{P}$ in such a system is trivial, because the infinite pattern of potentially delayed and dropped messages in known, and hence can be used to advantage. Such applications can simply send dummy information in the odd-numbered messages and use the even-numbered messages to communicate. Effectively, the applications have access to a reliable sub-channel consisting of the even-numbered messages. However, consider a system T where, during every prefix of computation, at most 50% of the messages sent may be delayed or dropped, but all other messages are delivered within some (unknown) bound on delay. Implementing $\diamond\mathcal{P}$ in such a system becomes non-trivial.

In this paper, we consider systems in which an infinite subset of messages can be *non-privileged*, insofar as they may be arbitrarily delayed or even dropped. We assume that non-privileged messages follow some distribution, about which we have only limited knowledge. All we assume is that each sufficiently long window of communication contains at least one message that is neither dropped, nor arbitrarily delayed. Our system model and implementation of $\diamond\mathcal{P}$ follow next.

[1] A communication channel is said to be reliable if every message sent to any correct process is delivered in finite time, and is neither lost, duplicated, nor corrupted.

3 System Model

We introduce a new model of partial synchrony based on unreliable communication links called *Average Delayed or Dropped* (ADD) Channels, and show how to implement $\diamond\mathcal{P}$. To our knowledge, the ADD system model is weaker than peer models of partial synchrony for implementing $\diamond\mathcal{P}$ using only bounded space.

3.1 Communication Model – ADD Channels

Every *Average Delayed or Dropped* (ADD) channel is a unidirectional communication link connecting two process. All messages sent on an ADD channel can be logically partitioned into two disjoint sets: *privileged* and *non-privileged*. This distinction is merely a modeling device which is known neither to the channel, nor to the application processes using such channels. Non-privileged messages have no timing or reliability guarantees. As such, infinitely many messages may be arbitrarily delayed or even dropped. By contrast, ADD channels provide the following guarantees for privileged messages:

1. If a process p sends infinitely-many messages to a correct process q on an ADD channel, then some infinite subset of those messages will be privileged. All such privileged messages will be delivered reliably to q.
2. For every execution of an ADD channel, there exists an unknown window size $w \in \mathbb{N}^+$, an unknown message delay $d \in \mathbb{N}^+$, and an unknown message ratio $r \in \mathbb{N}^+$, such that for every sending interval containing a subsequence S of exactly w privileged messages:
   (a) The average delay of all privileged messages in S is at most d.
   (b) The average number of non-privileged messages sent between any consecutive pair of privileged messages in S is at most r.

Intuitively, privileged messages are delivered reliably, and, on average, are neither too late nor too sparse. Privileged and non-privileged messages can be interleaved, but any interval containing w privileged messages is subject to the bounds d and r, which restrict the average delay of privileged messages, and the average ratio to non-privileged messages, respectively. Although the bounds w, d, and r exist for each ADD channel, they are unknown and may vary per run.

3.2 Simplified Reduction of ADD Channel Properties

Consider any run of an ADD channel with window size w and average privileged delay d. For contradiction, suppose that some privileged message m is delayed for more than $w \times d$ time units in this run. Consider any sequence S containing w privileged messages including m. The average delay of privileged messages in S must exceed d. Thus, the unknown window size w and the unknown bound d on *average* privileged delay actually induce an unknown bound on the *absolute* delay of privileged messages; specifically, no privileged message can be delayed more than $w \times d = D$ time units.

Similarly, let r bound the average number of non-privileged messages between any consecutive pair of privileged messages in windows of size w. Again, suppose for contradiction that more than $w \times r$ non-privileged messages are sent between some consecutive pair of privileged messages, say, m_i and m_j. Consider any sequence S containing w privileged messages including m_i and m_j. The average ratio of non-privileged to privileged messages in S must exceed r. Thus, the unknown window size w and the unknown bound r on *average* message ratio actually induce an unknown bound on the maximum number of non-privileged messages which can be sent between any consecutive pair of privileged messages; specifically, at most $w \times r = R$ non-privileged messages can be sent between any consecutive pair of privileged messages.

The foregoing analysis yields a simplified specification of the timeliness and reliability properties of ADD channels. This equivalent characterization will be used throughout the remainder of the paper in our analysis and proofs.

1. If a process p sends infinitely-many messages to a correct process q on an ADD channel, then some infinite subset of those messages will be privileged. All such privileged messages will be delivered reliably to q.
2. For every execution of an ADD channel, there exist two unknown bounds $D \in \mathbb{N}^+$ and $R \in \mathbb{N}^+$ such that:
   (a) The absolute delay of all privileged messages is at most D.
   (b) The maximum number of non-privileged messages sent between any consecutive pair of privileged messages is at most R.

3.3 ADD System Model

An ADD system consists of a finite set of processes Π where:

1. 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*.
2. Each pair of processes is connected via two reciprocal ADD channels, and message passing is the only means of interprocess communication.
3. There exists an unknown *lower-bound* on the absolute speed of live processes. No bounds on relative process speeds are assumed.
4. Each process has access to a local clock which generates ticks at a constant rate. Different clocks can tick at different rates and be unsynchronized. In physical systems, this is typically realized by a crystal oscillator which generates clock ticks at a constant frequency.

4 Implementation

In this section, we describe a heartbeat-based implementation of $\diamond\mathcal{P}$ in the ADD model for a system S with n processes. Our implementation has two types of modules: *heartbeat generators* and *heartbeat witnesses*. Each process p has a

single heartbeat generator, and n heartbeat witnesses (one for each process in S). Note that p has a heartbeat witness for itself as well. Thus, there are n generators and n^2 witnesses globally in the system.

Each heartbeat generator periodically sends n heartbeats, one to each process in S. The heartbeat frequency at each process is not necessarily known; moreover, it may vary from process to process due to local differences in hardware, server loads, and operating system scheduling policies. Nonetheless, we will show that each generator satisfies a *lower-bound* on heartbeat frequency. In conjunction with ADD channels, such heartbeat generators will yield an *upper bound* on the heartbeat inter-arrival times at each recipient. Although such bounds are not necessarily known, they can be adaptively estimated by the witness modules to provide an eventually reliable time-out mechanism.

Each heartbeat witness monitors the heartbeat traffic received on a given ADD channel. The role of each witness is to maintain a time-out variable that eventually converges to the de facto upper-bound on heartbeat inter-arrival time. Each witness starts with an initial estimate of the inter-arrival bound. If no heartbeat is received within the estimated bound, then the process sending heartbeats on this channel is suspected. False-positive suspicions may result if the estimated bound is too low, but such mistakes will be detected whenever a subsequent heartbeat is received. If so, the witness exonerates the previously suspected process, and increases its estimate of the inter-arrival bound.

The adaptive time-out mechanism just described is a common approach to implementing $\diamond\mathcal{P}$. The informal basis for correctness is as follows. Consider two processes p and q. If q crashes, then p receives at most finitely many messages from q. After the final such message, p will eventually time-out and permanently suspect q thereafter. This satisfies the strong completeness requirement of $\diamond\mathcal{P}$. By contrast, if p and q are both correct, then p can falsely suspect q at most finitely many times. Since q sends infinitely many heartbeats, each false suspicion of q will be detected by p, thereby causing p to increase its estimated bound on inter-arrival time. This estimate can only increase finitely many times before eventually exceeding the actual bound on inter-arrival time. Thereafter, p never suspects q again. This satisfies the eventual strong accuracy requirement of $\diamond\mathcal{P}$.

The foregoing argument for correctness depends on the critical assumption that there exists an upper bound on heartbeat inter-arrival times. The primary contribution of our work is to demonstrate that inter-arrival times can be bounded in the ADD system model, despite the fact that (1) there is no upper bound on relative process speeds, (2) there is no upper bound on message delay, and (3) infinitely many messages can be dropped. Nevertheless, we will prove that the ADD system model is sufficient to implement heartbeat generators with a lower bound on heartbeat frequency, and that ADD channels are sufficiently timely and reliable to convert lower bounds on heartbeat frequency into upper bounds on heartbeat inter-arrival times.

The remainder of this section describes the heartbeat generator and heartbeat witness modules, and defines a scheduler for fairly interleaving the actions of each module as threads within a single process.

```
class timer()
1 :   method timer.start(integer countervalue)
2 :       counter := countervalue
3 :       while (counter > 0)
4 :             upon event clock_tick() decrement counter
5 :       send notification
6 :   end method
7 :   method timer.stop()
8 :       counter := 0
9 :   end method
```

Fig. 1.1. Implementation of a timer using the local clock

4.1 Communication Patterns

Sending pattern and receiving pattern. In an execution, we refer to the set of messages sent by one process to another as a *sending pattern*. Some messages in the sending pattern may be dropped. Therefore, only a subset of the sending pattern is delivered. We refer to the set of messages (sent by some process) that is delivered to a process as a *receiving pattern*.

Bounded persistent sending pattern. Consider a sending pattern s consisting of an infinite number of messages. If there exists an upper-bound on the duration between every pair of consecutive messages, then s is referred to as a *bounded persistent* sending pattern.

4.2 Timer

In order to send heartbeats at regular intervals, and to implement an adaptive timeout mechanism, we need a mechanism to measure time. We accomplish this through a countdown timer. A class called *timer* implements such a countdown timer using the local clock available at each process. The pseudo-code for the class is show in Fig. 1.1.

The class *timer* uses the local clock primitive *clock_tick*() to measure time. The primitive *clock_tick*() is the output of the local clock provided by the system that generates ticks at a constant rate. *clock_tick*() generates an event (tick) that is used by *timer* to count down.

The class *timer* has two methods associated with it: *start*, and *stop*. The method *timer.start* accepts a countdown value, and starts counting down from the given value to zero. The counter is decremented for each clock tick. When the counter reaches zero, and a notification is sent to the process that called that timer. The method *timer.stop* simply sets the counter value to zero, and no notification is sent.

```
function generator(process p)
1 :   param:
2 :       p ∈ Π /* p is the client process */
3 :   var:
4 :       const inter_hb_time /* timer for sending heartbeats periodically */
5 :   primitive:
6 :       timer hb_tmr /* timer is the class from Fig. 1.1 */
7 :   begin
8 :       loop forever
9 :           hb_tmr.start(inter_hb_time)
10 :              wait until notification from hb_tmr
11 :              foreach q ∈ Π
12 :                  send a heartbeat to q
13 : end
```

Fig. 1.2. Pseudo-code *generator* implements the heartbeat generator at process p, periodically sending heartbeats to all process

4.3 Heartbeat Generator

The pseudo-code in Figure 1.2 implements the heartbeat generator. The parameter p is the local client process of the heartbeat generator. The variable *hb_tmr* is an instance of the *timer* class in Fig. 1.1. The variable *inter_hb_time* determines the interval between two heartbeats sent to a process. This variable is constant, and is determined before the execution begins.

Heartbeat generator sends heartbeats at regular intervals. The regularity of the interval is maintained by the *loop forever* construct in lines 8–12 in Fig. 1.2. The timer *hb_tmr* starts (line 9) in the beginning of each iteration of the loop for a duration of *inter_hb_time* time units. In line 10, the heartbeat generator waits for the timer to expire. Upon timer expiry, the heartbeat generator receives a notification. After receiving the notification, the heartbeat generator sends one heartbeat each to all the processes in the system. When *generator* reenters the loop, it restarts the timer for *inter_hb_time* time units. In other words, *generator* sends heartbeats after every *inter_hb_time* time units.

4.4 Heartbeat Witness

The pseudo-code in Fig. 1.3 implements the heartbeat witness. The function *receiver* (in Fig. 1.3), monitors heartbeats from a single process q. To monitor heartbeats from all processes in the system, a process p runs *receiver* for all processes $q \in \Pi$, concurrently.

The parameter p is the process that is receiving the heartbeats, the parameter q is the process whose heartbeats are being monitored by p. The integer variable *max_inter_arrival* stores the current estimate on the heartbeat inter-arrival time, and initialized to some positive value. The timer *tmr* counts down

```
function receiver(process p, process q)
1 :   param:
2 :       p ∈ Π /* p is the client process */
3 :       q ∈ Π /* q is the monitored process */
4 :   var:
5 :       integer max_inter_arrival := initial estimate
6 :       boolean suspect := false
7 :   primitive:
8 :       timer tmr /* timer is the class from Fig. 1.1 */
9 :   begin
10 :    tmr.start(max_inter_arrival)
11 :      loop forever
12 :          if (receive heartbeat from q)
13 :              if (suspect = true)                    /* wrongful suspicion */
14 :                  suspect := false
15 :                  increment max_inter_arrival        /* increase timeout */
16 :              tmr.start(max_inter_arrival)
17 :          if (receive notification from tmr)
18 :              suspect := true
19 : end
```

Fig. 1.3. Pseudo-code *receiver* implements the heartbeat witness monitoring process q, at process p

from *max_inter_arrival* to zero. The boolean variable *suspect* stores the current suspicion status of process q. If *suspect* is *true*, then p currently suspects q.

In Fig. 1.3, the timer *tmr* starts with some initial estimate of the upper-bound on inter-arrival time of heartbeats from q (line 8). The function *receiver* goes into an infinite loop (lines 11–18) waiting for either a heartbeat from q (line 12), or expiry of *tmr*. If a heartbeat is received from q (line 12), then *receiver* checks to see if q is already in the suspect list (line 13). If q is in the suspect list, then its a wrongful suspicion. Therefore, q is removed from the suspect list (line 14). A wrongful suspicion implies that the current estimate on the upper-bound on the inter-arrival time was less than the de-facto upper-bound. Hence, *receiver* increases the estimate on the the upper-bound (line 15). In line 16, the timer *tmr* is restarted. If the timer *tmr* expires (line 17), then q is added to the suspect list.

4.5 Scheduler

The threads *generator* (in Fig. 1.2) and *receiver* (in Fig. 1.3) are scheduled by a round-robin pre-emptive scheduler $\diamond\mathcal{P}$-exec as shown in Figure 1.4. The set of processes Π is finite and static. Therefore, the number of threads executed by $\diamond\mathcal{P}$-exec are finite and constant. The round-robin scheduling ensures that there is an upper-bound on how long the threads *generator*, and *receiver* have to wait in the queue before being scheduled for execution.

```
1 :  ◇P−exec (process p)
2 :      cobegin
3 :          foreach q ∈ Π
3 :              task receiver(p, q)
4 :          end foreach
5 :          task generator(p)
6 :      coend
```

Fig. 1.4. Eventually Perfect Failure Detector ($\Diamond\mathcal{P}$) in the ADD model

The lower-bound on absolute process speed translates to an upper-bound on the physical time it takes for $\Diamond\mathcal{P}$-exec (in Figure 1.4) to execute each instruction in its code. Round-robin scheduling of threads bounds the physical time that each *generator*, and *receiver* thread waits to be executed. The composition of the two bounds translates to an upper-bound on the physical time it takes for each thread to be scheduled and run on the processor.

Note that an upper-bound on physical time translates to an upper-bound on local time measured as the number of local clock ticks. This follows from the assumption that ADD system model provides a local clock that measures time at a uniform rate.

5 Proof of Correctness

5.1 Proof Outline

To prove correctness, we need to show that the implementation in Sec. 4 satisfies $\Diamond\mathcal{P}$ specifications *viz.*, *strong completeness* and *eventual strong accuracy.*

The proof of correctness is divided into three parts:

- In the first part, we prove that *generator* (in Fig. 1.2) described in Sec. 4.3 always generates a bounded persistent sending pattern.
- In the second part, we show that when a bounded persistent sending pattern is transmitted on an ADD channel, the channel yields a receiving pattern with an upper-bound on inter-arrival time.
- In the third part, we show that if *receiver* (in Fig. 1.3) described in Sec. 4.4 witnesses a receiving pattern with an upper-bound on inter-arrival time, *receiver* satisfies *strong completeness* and *eventual strong accuracy.*

The composition of the above three parts demonstrates that the implementation described in Sec. 4 satisfies $\Diamond\mathcal{P}$ specification.

5.2 Generating Bounded Persistent Sending Patterns

Lemma 1. *The heartbeat generator in Fig. 1.2 always generates a bounded persistent sending pattern.*

Proof. The following three arguments hold:

1. All the lines in the code take bounded local time to execute.
2. The scheduler (Sec. 4.5) guarantees that every time the *generator* thread enters the queue, waiting to be executed on the processor, there is an upper-bound on the duration (measured in local time) that the *generator* thread waits before executing on the processor.
3. Visual inspection of the pseudo-code in Fig. 1.2 reveals that notification for the expiry of timer *hb_tmr* is sent every constant number of clock ticks.

From the above three arguments it follows that there is an upper-bound on time between two consecutive executions of lines 10–12 in Fig. 1.2. In other words, there is an upper-bound on the time between every pair of consecutive heartbeats to each process. Given that the timer *hb_tmr* is started infinitely often, heartbeats are sent infinitely often. In other words, the sending pattern generated by these heartbeats is a *bounded persistent* sending pattern. □

5.3 Upper-bound on Heartbeat Inter-arrival Time

Let a bounded persistent sending pattern, transmitted on an ADD channel, yield a receiving pattern V. We show that there exists an upper bound on the inter-arrival time of the privileged messages in V. We use this result to prove that there exists an upper bound on on the inter-arrival time of all messages in V.

Lemma 2. *If a bounded persistent sending pattern of heartbeats is sent on an ADD channel, then there exists an upper-bound on the inter-arrival time of privileged heartbeats in the receiving pattern.*

Proof. ADD channel properties (Sec. 3.1) guarantee an unknown upper-bound R on the number of non-privileged messages between every pair of consecutive privileged messages. In other words, in every sequence of $2(R+1)$ consecutive heartbeats in an ADD channel, at least two of them are privileged.

The sending pattern is bounded persistent; by definition, there exists some upper-bound on the duration between the send times of every pair of consecutive heartbeats. Let us denote this bound as B time units. This implies that the upper-bound on time taken to send $2(R+1)$ heartbeats in the sending pattern is given by $2B(R+1)$ time units. In other words, the upper-bound between send times of every pair of consecutive privileged heartbeats is $2B(R+1)$ time units.

ADD channels also guarantee an unknown upper-bound D on the message delay of privileged messages. Thus, the upper-bound on the difference between message delays of two consecutive privileged messages is D time units. Therefore, the upper-bound on time elapsed between the arrivals of two consecutive privileged heartbeats is given by $A_p = 2B(R+1) + D$ time units, which is an upper-bound on the inter-arrival times of privileged heartbeats. □

Lemma 3. *A bounded persistent sending pattern of heartbeats, when sent on an ADD channel, yields a receiving pattern V with upper-bound on inter-arrival time.*

Proof. Let m be the earliest arriving privileged heartbeat in V. Let the arrival time of m be t_m. Let $V_m \subset V$ be the set of all heartbeats in V that arrived before, or at time t_m. Let $V_{m+} \subset V$ be the set of all heartbeats in V that arrived after, or at time t_m. Note the following: (1) $V_m \cup V_{m+} = V$, (2) all privileged heartbeats are in the set V_{m+}, and (3) heartbeat m is an element of both V_m and V_{m+}.

The set V_{m+} has all the privileged heartbeats. From Lemma 2, we know that there exists some unknown upper bound A_p on inter-arrival time of privileged heartbeats. Therefore, for every heartbeat $g \in V_{m+}$ there exists some privileged heartbeat $h \in V_{m+}$ that arrives within A_p time units after the arrival of g. Therefore, the upper-bound on inter-arrival time of heartbeats in V_{m+} is A_p.

If $|V_m| > 1$, then the inter-arrival time of heartbeats in V_m is at most t_m. This follows from the fact that $m \in V_m$ and m arrived at time t_m.

From the above argument it follows that the upper-bound on inter-arrival time for all the heartbeats that arrived before time t_m is t_m, and the upper-bound on the inter-arrival time for all the heartbeats that arrived at or after time t_m is A_p. Therefore, the upper-bound on the inter-arrival time for all heartbeats in V is given by $A = max(A_p, t_m)$. □

5.4 Strong Completeness and Eventual Strong Accuracy

If there exists an upper-bound on the heartbeat inter-arrival time, then *receiver* in Figure 1.3 satisfies Strong Completeness and Eventual Strong Accuracy.

Strong completeness. The strong completeness property states that every correct process eventually, and permanently suspects all crashed processes.

Lemma 4. *The heartbeat witness in Fig. 1.3 satisfies* ***strong completeness****.*

Proof. A faulty process crashes after some finite time, and therefore, sends only a finite number of heartbeats. Consider a run where process q crashes at time t_c. Let t_f be the time of the last receipt of heartbeats sent by q to a correct process p. After time t_f, process p does not receive any heartbeat from q, therefore the *if* condition in line 12 of *receiver* (Fig. 1.3) at p will evaluate to $false$ in the infinite suffix. In other words, if q is suspected by p after time t_f, then p will suspect q permanently thereafter.

At time t_f, process p has some finite value of timer tmr. Since tmr eventually expires, let it expire at time $t_s > t_f$. This implies that the *if* condition in line 17 of Fig. 1.3 evaluates to true at time t_s. Therefore the correct process p adds q to its suspect list at t_s.

From the above arguments it follows that all correct processes eventually and permanently suspect q, and hence, all crashed processes. □

Eventual strong accuracy. Eventual strong accuracy states that for each run, there exists a time after which no correct process is suspected by any correct process.

Lemma 5. *If there exists an upper-bound on the heartbeat inter-arrival time, receiver in Fig. 1.3 satisfies* ***eventual strong accuracy****.*

Proof. For a given run, for each pair of correct processes p and q, let the upper-bound on inter-arrival time of heartbeats from q to p be A time units.

The thread *receiver* at process p maintains an estimate on the upper-bound on the inter-arrival time of heartbeats (variable *max_inter_arrival* in Fig. 1.3) from q. If the current estimate on the upper-bound on inter-arrival time, at some time t, is equal to, or greater than A, then in the infinite suffix process p will receive a heartbeat from q before timer *tmr* expires at p. Therefore, process q is never suspected by p after time t.

However, if the current estimate on the upper-bound on inter-arrival time is less than A, then timer *tmr* at process p may expire before p receives a heartbeat from q, resulting in a false suspicion. However, within A time units after such a false suspicion, p will receive a heartbeat from q. Consequently, p will take q off the suspect list and increment the estimate on the upper-bound on inter-arrival time. The estimate on the upper-bound on inter-arrival time can be incremented only finitely many times before it exceeds A. Therefore, p can suspect q only finitely many times before the estimate on the upper-bound on inter-arrival time exceeds A. Therefore, in the infinite suffix following the last instance of p suspecting q, process p never suspects process q.

Since p and q are any two correct processes, the above argument applies to every pair of correct processes. In other words, there exists a time after which no correct process is suspected by any correct process. □

Theorem 6. *The algorithm described in Sec. 4 implements an eventually perfect ($\diamond\mathcal{P}$) failure detector in the ADD model.*

Proof. From Lemmas 1 and 3, it follows that the heartbeat generator module *generator* produces a bounded persistent sending pattern, which when transmitted on an ADD channel, yields a receiving pattern with an unknown upper-bound on the inter-arrival time.

From Lemmas 4 and 5 we conclude that if there exists an upper-bound on the inter-arrival time in the receiving pattern, the heartbeat witness module *receiver* satisfies strong completeness and eventual strong accuracy.

Thus, it follows that the algorithm described in Sec. 4 implements $\diamond\mathcal{P}$ in the ADD model. □

6 Conclusion

We have shown that one can implement eventually perfect failure detector ($\diamond\mathcal{P}$) in the weak, unreliable, partially synchronous ADD model with unbounded message loss as well as unbounded message delay for the majority of messages. Our work demonstrates a proximate but essential understanding of unreliable, partially synchronous communication via ADD Channels, and their sufficiency for implementing $\diamond\mathcal{P}$. This work is a fundamental step toward the ultimate goal

of understanding the minimal assumptions on reliability and partial synchrony necessary to implement this oracle.

References

1. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM **43** (1996) 225–267
2. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. Journal of the ACM **32** (1985) 374–382
3. Aguilera, M.K., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: Stable leader election. In: Proceedings of the 15th International Symposium on Distributed Computing, Springer (2001) 108–122
4. Aguilera, M.K., Chen, W., Toueg, S.: On quiescent reliable communication. SIAM Journal on Computing **29** (2000) 2040–2073
5. Guerraoui, R., Kapałka, M., Kouznetsov, P.: The weakest failure detectors to boost obstruction-freedom. In: Proceedings of the 20th International Symposium on Distributed Computing, Springer (2006) 399–412
6. Pike, S.M., Song, Y., Ghoshal, K.: Wait-free dining under eventual weak exclusion. Technical Report TAMU-CS-TR-2006-5-1, Texas A&M University (2006)
7. Pike, S.M., Sivilotti, P.A.G.: Dining philosophers with crash locality 1. In: Proceedings of the 24th International Conference on Distributed Computing Systems, IEEE (2004) 22–29
8. Song, Y., Pike, S.M.: Eventually k-bounded wait-free distributed daemons. Technical Report TAMU-CS-TR-2007-2-1, Texas A&M University (2007)
9. Guerraoui, R.: On the hardness of failure-sensitive agreement problems. Information Processing Letters **79** (2001) 99–104
10. Larrea, M., Fernández, A., Arévalo, S.: On the implementation of unreliable failure detectors in partially synchronous systems. IEEE Transactions on Computers **53** (2004) 815–828
11. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. Journal of the ACM **35** (1988) 288–323
12. Mostéfaoui, A., Mourgaya, E., Raynal, M.: Asynchronous implementation of failure detectors. In: Proceedings of the 33rd International Conference on Dependable Systems and Networks, IEEE (2003)
13. Bertier, M., Marin, O., Sens, P.: Implementation and performance evaluation of an adaptable failure detector. In: Proceedings of the 32nd International Conference on Dependable Systems and Networks, IEEE (2002) 354–363
14. Larrea, M., Arévalo, S., Fernández, A.: Efficient algorithms to implement unreliable failure detectors in partially synchronous systems. In: Proceedings of the 13th International Symposium on Distributed Computing, Springer (1999) 34–48
15. Fetzer, C., Raynal, M., Tronel, F.: An adaptive failure detection protocol. In: Proceedings of the 7th Pacific Rim International Symposium on Dependable Computing, IEEE (2001) 146–153
16. Fetzer, C., Schmid, U., Süsskraut, M.: On the possibility of consensus in asynchronous systems with finite average response times. In: Proceedings of the 25th International Conference on Distributed Computing Systems, IEEE (2005) 271–280
17. Larrea, M., Lafuente, A.: Communication-efficient implementation of failure detector classes $\diamond\mathcal{P}$ and $\diamond\mathcal{Q}$. In: Proceedings of the 19th International Symposium on Distributed Computing, Springer (2005) 495–496