Crash-Quiescent Failure Detection*

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. A distributed algorithm is *crash quiescent* if it eventually stops sending messages to crashed processes. An algorithm can be made crash quiescent by providing it with either a crash notification service or a reliable communication service. Both services can be implemented in practical environments with *failure detectors*. Therefore, crash-quiescent failure detection is fundamental to system-wide crash quiescence. We establish necessary and sufficient conditions for crash-quiescent failure detection in partially synchronous environments where a bounded, but unknown, number of consecutive messages can be arbitrarily late or lost. Without a correct majority of processes, not even the weakest oracle for fault-tolerant consensus, $\diamond\mathcal{W}$, can be implemented crash quiescently. With a correct majority, however, the eventually perfect failure detector, $\diamond\mathcal{P}$, is possible. Our $\diamond\mathcal{P}$ algorithm is correct in all runs, but improves performance via crash quiescence in any run with a correct majority. We also present a refinement of our $\diamond\mathcal{P}$ algorithm to mitigate the overhead of achieving crash quiescence; the resulting bit complexity per utilized link is asymptotically better than or equal to that of non-crash-quiescent counterparts.

1 Introduction

A distributed algorithm is called *crash quiescent*, if, in all runs, correct processes eventually stop sending messages to crashed processes. Depending on the system model, crash quiescence may be straightforward, non-trivial, or even impossible. For example, crash quiescence is straightforward with reliable communication, even for purely asynchronous systems: every message received generates an ack, and each process can have at most k unacknowledged messages per process at any time. Crashed processes — which permanently halt without warning — stop sending acks, so after the final such ack is delivered, each correct process will become crash quiescent once k subsequent messages go unacknowledged.

By contrast, crash quiescence with unreliable communication is far more challenging due to inherent limitations on process coordination in the presence of both crash faults and message loss. For example, consider any application

* This work was supported in part by Texas Higher Education Coordinating Board grants ARP-00512-0007-2006 and ARP-00512-0130-2007, and by NSF grant 0500265.

where some correct process i requires acknowledged delivery of a message m to each correct process. With lossy communication, i must re-send m sufficiently many times until the corresponding acks are received from each correct process; otherwise, the application program will be incorrect. For each crashed process, however, i must eventually stop re-sending m; otherwise, crash quiescence will be violated. In such systems, correct processes are committed to distinctly different (and contradictory) communication behaviors, depending on whether messages are sent to correct or faulty processes.

Since each message can be dropped (due to message loss), or never sent (due to process crashes), or just late (due to asynchrony), crash-quiescent algorithms must navigate an intersection of uncertainties. Fortunately, in systems with both crashes and native message loss, application-layer algorithms can be made crash quiescent relative to underlying system services for crash detection. As such, crash-quiescent failure detection is fundamental to system-wide crash quiescence.

A *failure detector* [1] can be viewed as a distributed oracle that can be queried for (potentially unreliable) information about process crash faults. Despite such unreliability, failure detectors can solve many problems that are *not* solvable in pure asynchrony [2]: most notably, crash-tolerant consensus [1]. As system services, failure detection oracles decouple distributed algorithms from explicit commitments to lower-level timing parameters; more theoretically, such oracles function as proxies for various degrees of partial or even full synchrony.

One oracle — the eventually perfect failure detector $\diamond\mathcal{P}$ — is particularly useful for enabling crash-quiescent applications. Informally, $\diamond\mathcal{P}$ suspects all crashed processes and eventually trusts all correct processes permanently. As such, $\diamond\mathcal{P}$ can suspect correct processes only finitely many times. In an asynchronous system augmented with $\diamond\mathcal{P}$, applications can become crash quiescent (despite message loss) as follows: so long as the recipient remains trusted by $\diamond\mathcal{P}$, re-send each (new or buffered) message sufficiently many times until an acknowledgment is received; otherwise, buffer outbound messages while the recipient is suspected.

The foregoing protocol essentially provides quiescent reliable communication among correct processes, which is the approach taken by [3] as well. The same paper proves that — among oracles that output a list of suspected processes — $\diamond\mathcal{P}$ is actually the weakest failure detector for quiescent reliable communication. Nonetheless, a fundamental problem remains: *for system-wide crash quiescence, it is essential that any underlying oracles are crash quiescent as well.*

Contribution. We prove necessary and sufficient conditions for crash-quiescent failure detection in partially synchronous environments where a bounded, but unknown, number of consecutive messages can be arbitrarily late or lost. Without a correct majority of processes, not even the weakest oracle for fault-tolerant consensus, $\diamond\mathcal{W}$, can be implemented crash quiescently. With a correct majority, however, $\diamond\mathcal{P}$, is possible. Our $\diamond\mathcal{P}$ algorithm is correct in all runs, but improves performance via crash quiescence in any run with a correct majority. We also present a refinement of our $\diamond\mathcal{P}$ algorithm to mitigate the overhead of achieving crash quiescence; the resulting bit complexity per utilized link is asymptotically better than or equal to that of non-crash-quiescent counterparts.

2 Definitions

System Model. We consider partially-synchronous systems subject to bounded intervals of message loss and delay. We start with the canonical model $\mathcal{M}_1$ from [1,4], and we weaken channel reliability and synchrony guarantees to allow an infinite number of messages to be lost or arbitrarily delayed. Specifically, we assume that communication takes place on ADD channels [5]. We informally describe the system model, which we denote E_{CLPS} (for Communication-Lossy Partially-Synchronous Environment — pronounced "eclipse"), next.

The system consists of a finite fixed set Π of n processes. We assume that the set Π is known to all processes. Each process's local program is represented as an action system consisting of a finite set of guarded commands. At each step of a process, the process can receive at most one message from one of its incoming message buffers, update its local state, and send at most one message to each process. Each process' action system includes a special *crash* action. The crash action can be executed at most once and permanently disables the guards of all the program actions, thereby halting the process.

Processes communicate with each other by sending messages over a fully connected communication topology. A send statement by process i causes the indicated message to be added to the channel from i to the recipient process j. When a message m is in the channel from i to j, a *deliver* action is enabled whose effect is to remove m from the channel and place it in the incoming message buffer at j for sender i.[1]

Starting from a system state in which channels are empty and local program variables have specified initial values, a *run* of an algorithm consists of a potentially infinite sequence of enabled actions (or steps). Each action in the run is either a local program action of a process, a crash action, or a deliver action of a channel. If the run is finite, then no program action should be enabled at the end of the run. A process that has not (yet) crashed is called *live*. Processes that never crash are called *correct*, and processes that crash are called *faulty*.

In a given run, each step is associated with a non-negative integer, which is the real time when it occurs; times assigned to steps in a run must be non-decreasing, but no two steps by the same process may have the same time. This common way of modeling runs enforces an upper bound on *absolute* process speed[2], but processes can decelerate indefinitely subject to the following restriction on *relative* process speeds: there exists $\Phi \in \mathbb{N}$ such that if (1) processes i and j are both live during a time interval, and (2) i takes at least Φ steps in the interval, then j takes at least one step in the interval. The bound Φ is not necessarily known to the processes and can vary for different runs of the system.

We assume that the actions of each process are locally scheduled by a First-Come-First-Serve (FCFS) scheduler which executes program actions in the order

[1] Our impossibility result holds even if incoming message buffers are infinite, while our algorithm works with a one-slot buffer whose contents are overwritten by the execution of each deliver event.

[2] This assumption is necessary to implement eventually reliable timeouts using only action-time clocks [6].

in which they were enabled. Note that such scheduling fairness applies only to program actions and not to crash actions, which are a modeling device. Thus, the crash action is always enabled at a correct process but never executed, while it is continuously enabled at a faulty process until the action is executed.

Each channel guarantees that some subset of the messages sent on it will be delivered in a timely manner and such messages are not too sparsely distributed in time, i.e., it is an ADD channel [5]. In more detail, consider the channel from process i to process j. The (real-time) *delay* of a message is the time elapsed between the step in which the message is sent and the deliver event for the message; if there is no deliver event, then the delay is infinite. For each run, there exist constants $\Delta \in \mathbb{N}$ and $B \in \mathbb{N}$ and a subset S_p of the set of messages sent over the channel (the *privileged* messages) satisfying the following: (1) The delay of each message in S_p is at most Δ. (2) For all intervals of time in which i sends at least B messages to j, at least one of the messages sent over the channel in that interval is in S_p. The bounds Δ and B and the set S_p are not necessarily known to the processes and can be different in different runs of the system.[3]

As consequences of our model definition, the following properties hold:

Property 1. The maximum number of steps taken by a process during the time that a privileged message is in transit in a channel is Δ.

Property 2. For every pair of processes i and j, the maximum number of steps taken by i during a time period in which j takes s steps is $(\Phi \cdot s)$.

Eventually Perfect Failure Detector. The *eventually perfect failure detector* $\diamond\mathcal{P}$ satisfies the following two properties in each run [1]:

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

$\diamond\mathcal{P}$ is a particularly attractive oracle. First, it is sufficiently powerful to solve many fundamental problems including fault-tolerant consensus [1], stable leader election [7], and wait-free dining [8]. Additionally, it is realistically implementable: in contrast to other relatively powerful oracles — such as Perfect [1], Strong [1], and Trusting [9] — $\diamond\mathcal{P}$ is actually implementable in classic models of partial synchrony[4].

Crash Quiescence. Algorithm $\mathcal{A}$ is said to be *crash quiescent* if, for every run of $\mathcal{A}$, there exists a time after which no correct process sends messages to any crashed process.

[3] If the bound B is known, then implementing $\diamond\mathcal{P}$ in E_{CLPS} with an arbitrary number of crash faults is straightforward. Simply take a standard ping-ack implementation of $\diamond\mathcal{P}$ for reliable channels, and instead of sending a single ping or a single ack, send B pings or B acks, respectively.

[4] This claim is based on: (a) the many implementations of $\diamond\mathcal{P}$ in recent works (cf. [1, 7, 10–15]), and (b) the results from [16], where Larrea, *et al.*, prove that failure detectors with perpetual accuracy (including $\mathcal{P}$, and $\mathcal{S}$) cannot be implemented in classic models of partial synchrony [1, 4].

3 Impossibility of Crash-Quiescent $\diamond\mathcal{P}$ in E_{CLPS}

In this section we show that it is impossible to implement $\diamond\mathcal{P}$ crash quiescently in E_{CLPS} without a correct majority of processes. We start by showing that it is impossible to implement the *eventually weak failure detector* ($\diamond\mathcal{W}$) [1], a weaker failure detector than $\diamond\mathcal{P}$, in E_{CLPS} crash-quiescently if at most $\lceil \frac{n}{2} \rceil$ processes may crash. The oracle $\diamond\mathcal{W}$ satisfies *weak completeness*, which states that every crashed process is eventually and permanently suspected by *some* correct process, and *eventual weak accuracy*, which states that *some* correct process is eventually and permanently trusted by all correct processes. Since every implementation of $\diamond\mathcal{P}$ is also an implementation of $\diamond\mathcal{W}$, the impossibility result for $\diamond\mathcal{W}$ holds for $\diamond\mathcal{P}$ as well.

Theorem 3. *It is impossible to implement a deterministic crash-quiescent $\diamond\mathcal{W}$ in E_{CLPS} if up to $\lceil \frac{n}{2} \rceil$ processes may crash.*

Proof. For the purpose of contradiction, assume there is an algorithm $\mathcal{A}$ that implements crash-quiescent $\diamond\mathcal{W}$ in E_{CLPS}. Partition the set of processes into two sets X and Y such that $|X| = \lfloor \frac{n}{2} \rfloor$ and $|Y| = \lceil \frac{n}{2} \rceil$.

Consider a run α_X in which all processes in X are correct and execute in synchronous rounds, all processes in Y crash initially, and all messages are received (by correct processes) in the next round after they are sent. By the assumed correctness of $\mathcal{A}$, there exists a round r_X after which each process in Y is permanently suspected by some process in X, and all processes in X stop sending messages to processes in Y. Let m_X denote the maximum number of messages sent by any process $x \in X$ to any process $y \in Y$ during execution α_X.

Let α_Y be a run that is the same as α_X except that the roles of X and Y are reversed; define r_Y and m_Y analogously to r_X and m_X.

Let L_{part} be the set of communication links that go between processes in X and processes in Y.

Now consider a run α in which all processes are correct and execute in synchronous rounds. All messages sent over links in L_{part} through round $r = \max(r_X, r_Y)$ are lost, and all other messages (those sent over the other links and those sent over L_{part} after round r, if any) are delivered with delay of one round. It can be shown using standard arguments that α_X and α are indistinguishable to all processes in X through round r, and thus each process in X is quiescent with respect to all processes in Y at the end of round r of α. Similarly, α_Y and α are indistinguishable to all processes in Y through round r, and thus each process in Y is quiescent with respect to all processes in X at the end of round r of α. The processes in X remain quiescent with respect to the processes in Y and each process in Y is permanently suspected by at least one process in X. Similarly, the processes in Y remain quiescent with respect to the processes in X and each process in X is permanently suspected by at least one process in Y.

The link behavior in α conforms to the ADD channel specification with $B = \max(m_X, m_Y)$. Since $\mathcal{A}$ is supposed to work correctly in α but every correct process is permanently suspected by at least one other correct process (violating the specification of $\diamond\mathcal{W}$), we have a contradiction. □

$\diamond\mathcal{P}$ *implementation for each process* $i \in \Pi$

constant n		*Total number of processes in* Π
constant $intermission_i \in \mathbb{N}^+$		*Min. interval between sending heartbeat pulses*
integer $next\text{-}pulse_i$	$:= 0$	*Countdown timer to send next heartbeat pulse*
integer $estimate_{ij}$	$:= 1$	*Predicted interval between heartbeats from* j
integer $deadline_{ij}$	$:= 1$	*Countdown timer to receive next heartbeat from* j
$\Pi \times \Pi$ **boolean matrix** S_i		*Suspect matrix:* $S_i(j,k) = \text{true}$ *implies* j *suspected* k

$\diamond\mathcal{P}_i \stackrel{\text{def}}{=} \{\forall j \in \Pi : S_i(i,j) = true : j\}$ — *$\diamond\mathcal{P}$ output when queried by i*

$Q_i \stackrel{\text{def}}{=} \{\forall j,k \in \Pi : S_i(i,k) \wedge count(\neg S_i(i,j) \wedge S_i(j,k)) > \lfloor \frac{n}{2} \rfloor : k\} \cup \{i\}$ *Quiescence Set*

---

1 : $\{next\text{-}pulse_i = 0\} \longrightarrow$ — Action 1: Send Pulse
2 : **foreach** $(j \notin Q_i)$ **send** $\langle \diamond\mathcal{P}_i \rangle$ **to** j — *Send suspected ids in pulse*
3 : **foreach** $(k \in \Pi)$ **do** $S_i(i,k) := (deadline_{ik} = 0)$ — *Update local suspect list*
4 : $next\text{-}pulse_i := intermission_i - 1$ — *Schedule next heartbeat pulse*

---

5 : $\{$**receive** $\langle hb \rangle$ **from** $j\} \longrightarrow$ — Action 2: Receive Heartbeats
6 : **if**$(S_i(i,j) =$ **true**$)$ — *Detect a false-positive mistake*
7 : $S_i(i,j) :=$ **false** — *Remove j from local suspect list*
8 : $estimate_{ij} := estimate_{ij} + 1$ — *Increase predicted interval for j*
9 : **foreach** $(k \in \Pi)$ **do** $S_i(j,k) := (k \in hb)$ — *Update suspect matrix row for j*
10 : $deadline_{ij} := estimate_{ij}$ — *Set next heartbeat deadline for j*

---

11 : $\{$**true**$\} \longrightarrow$ — Action 3: Decrement Timers
12 : $next\text{-}pulse_i := max(0, next\text{-}pulse_i - 1)$
13 : **foreach** $(j \in \Pi)$ **where** $(j \neq i)$ **do** $deadline_{ij} := max(0, deadline_{ij} - 1)$

Alg 1.1. Implementation of $\diamond\mathcal{P}$ that is correct in all runs and crash quiescent in any run with a correct majority of processes. Initial values for the suspect matrix can be arbitrary. Note that $deadline_{ii}$ is initially positive and is never decremented, so each process i will eventually trust itself permanently.

Corollary 4. *It is impossible to implement a crash-quiescent $\diamond\mathcal{P}$ in E_{CLPS} if up to $\lceil \frac{n}{2} \rceil$ processes may crash.*

4 Crash-Quiescent $\diamond\mathcal{P}$ in E_{CLPS} with Majority Correct

In contrast to the previous result, we now show that it is possible to implement crash-quiescent $\diamond\mathcal{P}$ in E_{CLPS} if a majority of the processes are correct; furthermore, the implementation is correct, although not crash-quiescent, without a correct majority of processes.

Alg. 1.1 presents one such $\diamond\mathcal{P}$ implementation in E_{CLPS}. It is a heartbeat-based implementation that gains extra information by exchanging suspect lists with other processes. This extra information is used to achieve crash quiescence. Specifically, each process i relays its entire suspect list by including it in heartbeat messages sent to other processes at regular intervals (Action 1, line 2). The

intervals are measured by a step timer $next\text{-}pulse_i$ which is decremented in Action 3 to send the heartbeat messages. Every time $next\text{-}pulse_i$ expires (counts down to zero), the process sends heartbeats with its current suspect list to a subset of processes (Action 1, line 2), determined by a method explained later.

Process i expects to receive heartbeats from each live neighbor at regular intervals. The upper bound on the inter-arrival time of the heartbeats may be unknown. Hence, i has an adaptive step timer $deadline_{ij}$ with respect to each process j that is initialized to the value of $estimate_{ij}$ and is decremented in Action 3. If the timer $deadline_{ij}$ expires (counts down to zero) before i receives a heartbeat from j (Action 1, line 3), then i suspects j. Every time i receives a heartbeat from a process j, i trusts j (Action 2) and restarts the timer $deadline_{ij}$ (Action 2, line 10). However, if j was previously suspected by i, then i also increases the timer value $estimate_{ij}$ (Action 2, line 8).

Recall that processes send their suspect lists in each heartbeat. When a process i receives a heartbeat from process j, it records the list of processes suspected by j, as communicated in that heartbeat (Action 2, line 9), in the j^{th} row of the suspect matrix S_i. Every time $next\text{-}pulse_i$ expires, process i determines the set of processes that it will not send a heartbeat to as follows: If a process j is currently suspected by i, and among the processes that i trusts, more than $\lfloor \frac{n}{2} \rfloor$ suspect j (as communicated through the latest heartbeats received by i), then i adds j to the quiescence set Q_i (as per the definition of Q_i in Alg. 1.1), and i does not send a heartbeat to j. Also, note that i is always in Q_i. The set Q_i is dynamically defined every time $next\text{-}pulse_i$ expires, so it is possible for i to send a heartbeat to j in some instances of Action 1 in a run and not in others.

4.1 Proof of Correctness

Theorem 5. *Alg. 1.1 satisfies strong completeness: every crashed process is eventually and permanently suspected by all correct processes.*

Proof. Upon crashing, each faulty process j stops sending heartbeats. Thus, each correct process i stops receiving heartbeats from j. Since Action 3 at i is always enabled and executed infinitely often, eventually $estimate_{ij}$ is permanently 0. After such time, all executions of Action 1 at i suspect j, and j is never trusted again because no heartbeats from j are received. □

We prove eventual strong accuracy (eventually no correct process is suspected by any correct process) through the following lemmas:

Lemma 6. *The values taken on by variable $estimate_{ij}$ are non-decreasing, and every time a process j is taken off a process i's suspect list, the value of $estimate_{ij}$ is increased.*

Proof. Inspection of Alg. 1.1 reveals that $estimate_{ij}$ is never decremented. Furthermore, the only action at i that takes a process j off the suspect list is Action 2 (in line 6–9). However, the same action also increments the value of $estimate_{ij}$ by 1 (in line 9) after j is taken off the suspect list. □

Since the action system at each process contains $n+1$ actions and the local scheduler is FCFS, we get:

Lemma 7. *For each correct process i, the maximum number of steps executed by i between the time that an action a is enabled at i and the earliest time thereafter that the action a is executed is n.*

Let INT denote the largest $intermission_i$ over all processes $i \in \Pi$.

Lemma 8. *Within every interval in which process i takes $n \cdot INT$ steps, i executes Action 1 at least once.*

Proof. Inspection of Alg. 1.1 shows that $next\text{-}pulse_i$ is always non-negative, it is set to a value not exceeding $INT-1$ in Action 1, and it is decremented by 1 in Action 3. If Action 3 is executed $INT-1$ times, then $next\text{-}pulse_i$ is guaranteed to be decremented to 0, enabling Action 1. Since Action 3 is always enabled, by Lemma 7, we know that within $n \cdot (INT-1)$ steps by i, Action 1 is enabled at i. Applying Lemma 7 again, we know that Action 1 will be executed within the next n steps at i. □

Lemma 9. *If processes i and j are correct and i sends a heartbeat to j infinitely often, then j sends a heartbeat to i infinitely often.*

Proof. If process i sends heartbeats to process j infinitely often, then j receives heartbeats from i infinitely often. From Action 2, we know that j trusts i upon receiving a heartbeat from i. Process j continues to trust i until the next execution of Action 1, guaranteed to occur by Lemma 8. This execution of Action 1 will send a heartbeat to i. □

Lemma 10. *If processes i and j are correct and i sends a heartbeat to j infinitely often, then i and j eventually trust each other permanently.*

Proof. If i sends heartbeats to j infinitely often, then by Lemma 9 j sends heartbeats to i infinitely often as well. Consequently, i and j trust each other infinitely often. By Lemma 6 we know that every time i (falsely) suspects j, the value of $estimate_{ij}$ increases in the future when i trusts j again. Similarly, every time j (falsely) suspects i, the value of $estimate_{ji}$ increases in the future when j trusts i again.

We now show that after i and j suspect each other finitely many times, either i and j trust each other permanently (thus vacuously satisfying eventual strong accuracy), or the values of $estimate_{ij}$ and $estimate_{ji}$ grow sufficiently large such that: in an infinite suffix, i and j always receive heartbeats from each other before timers $deadline_{ij}$ and $deadline_{ji}$ (which are reset to $estimate_{ij}$ and $estimate_{ji}$, respectively) expire. Therefore, i and j eventually and permanently trust each other.

Let $M \stackrel{\text{def}}{=} B \cdot n \cdot INT + \Delta + \Phi(n + B \cdot n \cdot INT) + \Delta + n$. Consider a time t_{suf} at which: (a) either i permanently trusts j or $estimate_{ij}$ exceeds M, and (b) either j permanently trusts i or $estimate_{ij}$ exceeds M.

Consider any time t_f (subscript f for *final*) after t_{suf} at which i receives a heartbeat from j by executing Action 2. Thus, i trusts j at time t_f and $deadline_{ij}$ is reset to $estimate_{ij}$. By Lemma 8 we know that in the next $B \cdot n \cdot INT$ steps at i, process i executes Action 1 at least B times. Since $estimate_{ij}$ exceeds M, which is greater than $B \cdot n \cdot INT$, i continues to trust j for B executions of Action 1, and therefore, B heartbeats are sent to j. Note that at least one heartbeat among the B is privileged. Let m_1 be one such heartbeat.

From the system model definitions, the message delay for m_1 is at most Δ time ticks. By Property 1, the maximum number of steps taken by i while m_1 is in transit is Δ. Delivery of m_1 at j enables Action 2 at j (if it had not been enabled already).

By Lemma 7, the maximum number of steps by j between the delivery of m_1 and the receipt of a heartbeat from i is n. The same argument as above shows that after j receives a heartbeat from i, j trusts i and within $B \cdot n \cdot INT$ steps (at j) process j sends a privileged message, m_2, to i. Thus, by Property 2 the maximum number of steps taken by i during the time that j is waiting to take delivery of a heartbeat from i and to send the B heartbeats (including m_2) to i is $\Phi(n + B \cdot n \cdot INT)$.

A symmetric argument shows that the maximum number of steps taken by i while m_2 is in transit to i and some heartbeat from j is received by i is $\Delta + n$.

In aggregate, we see that within $(B \cdot n \cdot INT + \Delta + \Phi(n + B \cdot n \cdot INT) + \Delta + n) = M$ steps of i after time t_f, i gets another heartbeat from j. Since $estimate_{ij} > M$, i has been continuing to trust j throughout this interval. Applying the same argument iteratively, it follows that i never suspects j after time t_f.

Reversing the roles of i and j shows that j never suspects i after time t_f. □

Lemma 11. *If processes i and j are correct and i sends only finitely many heartbeats to j, then j sends only finitely many heartbeats to i, and i and j suspect each other eventually and permanently.*

Proof. Let i and j be two correct processes such that i sends only finitely many heartbeats to j. By the contra-positive of Lemma 9, j sends only finitely many heartbeats to i as well.

Let t_x be the latest time at which a heartbeat from i to j, or from j to i, is received. After t_x, process i never executes Action 2 with respect to j, and j never executes Action 2 with respect to i. Consequently, timers $deadline_{ij}$ and $deadline_{ji}$ are never increased, but since Action 3 at both i and j is continuously enabled, the timers are decremented infinitely often. Eventually, these timers reach 0 and when i and j execute their respective Action 1 after such time, i suspects j, and *vice versa*. Since no more heartbeats are received by i or j from each other, processes i and j suspect each other permanently. □

Lemma 12. *If process i is correct, then its suspect list stops changing.*

Proof. From Theorem 5 we know that eventually all crashed processes are permanently suspected. From Lemmas 10 and 11 we know that i either eventually and permanently trusts a correct process j, or i eventually and permanently suspects a correct process j. That is, i's suspect list eventually stops changing. □

Theorem 13. *Alg. 1.1 satisfies eventual strong accuracy whereby every correct process is eventually and permanently trusted by all correct processes.*

Proof. From Lemma 12 we know that the suspect list at each correct process stops changing eventually. Consider a run α of Alg. 1.1. Let t_{stable} be the time after which the suspect list at each correct process stops changing and all faulty processes have crashed.

Let i and j be two correct processes in run α. If i sends infinitely many heartbeats to j, then from Lemma 10 we know that i eventually and permanently trusts j. However, if i sends only finitely many heartbeats to j, then from Lemma 11 we know i and j eventually and permanently suspect each other. We will now show that the latter is impossible.

For the purposes of contradiction, let us assume that i sends only finitely many heartbeats to j. By Lemma 11, i and j eventually and permanently suspect each other and stop sending heartbeats to each other. By Lemma 12, the suspect lists of all correct processes eventually stop changing. Hence, eventually, i and j receive unchanging heartbeats (if at all) from other correct processes in the system. In other words, eventually, the suspect matrices S_i and S_j stay constant.

Since i eventually stops sending heartbeats to j, it implies that $j \in Q_i$ eventually and permanently. That is, i trusts a majority of processes, and therefore, a majority of processes trust i (follows from Lemma 10 and the fact that i sends heartbeats in Action 1 to such trusted processes infinitely often). Also, a majority of processes suspect j. This suspicion information is relayed to i in the contents of the heartbeats from the trusted processes.

Reversing the roles of i and j in the arguments above, we know that a majority of processes suspect i for j to stop sending heartbeats to i.

The above arguments establish that a correct majority of processes trust i permanently for i to stop sending heartbeats to j, but a correct majority of processes also suspect i permanently for j to stop sending heartbeats to i. This is a contradiction! Hence, it follows that i and j send heartbeats to each other infinitely often. Lemma 10 implies that i eventually and permanently trusts j.

Thus, every correct process is eventually and permanently trusted by all correct processes. □

Theorem 14. *Alg. 1.1 is crash quiescent if a majority of processes are correct.*

Proof. From Theorem 5, we know that every crashed process is suspected by every correct process. From Theorem 13, we know that every correct process eventually and permanently trusts every correct process. Hence, every correct process receives suspect lists from all correct processes infinitely often. Since eventually every correct process permanently suspects every crashed process, the following is eventually and permanently true: for all pairs of processes (i, k) where i and k are correct, $S_k(i, j)$ is true for all crashed processes j.

Therefore, in all runs where a majority of processes are correct, eventually for every correct process i, every crashed process j is permanently in the set Q_i. In other words, every correct process i eventually and permanently stops sending heartbeats to any crashed process. □

Communication Complexity. Next, we analyze the communication complexity of our algorithm, both in terms of the number of messages and the number of bits sent. Since processes send messages periodically, and correct processes never cease doing so, we focus on the number of messages (and bits) sent in each period (the same approach is used in, for instance, [12]). In every run of a non-crash-quiescent implementation of $\diamond\mathcal{P}$ using heartbeats, eventually all faulty processes have crashed and every correct process periodically sends heartbeats to all the processes, resulting in $\mathcal{O}(c \cdot n)$ messages per period, where c is the number of correct processes. By contrast, in runs of Alg. 1.1 where $c > \lfloor \frac{n}{2} \rfloor$, eventually $\mathcal{O}(c^2)$ messages are sent per period. Thus, Alg. 1.1 offers improved message complexity in majority-correct runs, and this improvement has no penalty in message complexity for runs where half or more processes crash.

Ironically, the bit complexity of Alg. 1.1 is greater than that of its non-crash-quiescent counterparts in all runs. Since the receipt of a "dummy" heartbeat message devoid of any payload may be sufficient to establish the liveness of the sender, each heartbeat message in a non-crash-quiescent algorithm requires just $\mathcal{O}(\log n)$ bits (for encoding the sender and recipient). But each message in Alg. 1.1 requires $\Theta(n)$ bits (to encode the suspect list and the sender and recipient addresses). Thus the total periodic bit complexity of Alg. 1.1 is $\Theta(c^2 \cdot n)$ as compared to $\mathcal{O}(c \cdot n \cdot \log n)$ for the non-crash-quiescent version. Thus for instance, if c is a constant fraction of n, as will be the case if processes have a fixed probability of failure, the bit complexity of Alg. 1.1 is asymptotically worse than that of the non-crash-quiescent version. Yet the purpose of crash-quiescence is to reduce the overall communication complexity. We address this next.

5 Improving the Communication Bit Complexity

Algorithm. We improve the communication bit complexity of Alg. 1.1 by inserting a communication sub-layer between Alg. 1.1 and the communication infrastructure. This communication sub-layer sends and receives two types of heartbeats: *heavyweight heartbeats* and *lightweight heartbeats*. The heavyweight heartbeats contain the entire suspect list sent by the process, whereas the lightweight heartbeats merely contain 'i-am-alive' information.

Alg. 1.2 implements such a communication sub-layer. In the action system, each process maintains, in the variable $prev\_hb_{ij}$, the suspect list that the $\diamond\mathcal{P}$ module sent to process j in the previous heartbeat. If the contents of the current heartbeat to be sent are different from the contents of the previous heartbeat sent, then the action system generates a new sequence number (Action 1, line 3) and constructs a heavyweight heartbeat (Action 1, line 5). However, it sends the heavyweight heartbeat only if the recipient is not currently suspected (Action 1, line 6)[5]; otherwise it sends a lightweight heartbeat (Action 1, line 7). Alg. 1.2 stores the sequence number of the latest heartbeat received from each process

[5] The condition for trusting a recipient to send a heavyweight heartbeat ensures that in non-crash-quiescent runs, a correct process does not send an infinite number of heavyweight heartbeats to a crashed process.

```
set prev_hb_ij := ∅      The previous heartbeat sent by the local ◇P-module to process j
message msg_ij := ⟨0, null, 0⟩                    The actual heartbeat (HB) sent to j
integer seq_num_ij := 0                  The current sequence number for heartbeats to j
integer max_seq_ij := 0                    The highest sequence number received from j
integer latest_ack_ij := 0                                  The latest ack sent to j
message hb_ij := ∅              The heartbeat (suspect list) sent to the local ◇P module
---------------------------------------------------------------------------------------
1 : {upon exec(◇P−send ⟨hb⟩ to j)} ⟶                              Action 1: Send a heartbeat
2 :     if (hb ≠ prev_hb_ij)                          Check if the suspect list has changed
3 :       increment seq_num_ij by 1        A new sequence number for new suspect list
4 :       prev_hb_ij := hb                    Update local record of the latest heartbeat
5 :       msg_ij := ⟨seq_num_ij, hb, latest_ack_ij⟩      Construct HB (piggybacked ack)
6 :     if (j ∉ hb) send ⟨msg_ij⟩ to j              Send constructed HB if j is trusted
7 :     else send ⟨0, null, 0⟩ to j                              Else send a lightweight HB
---------------------------------------------------------------------------------------
8 : {upon receive ⟨seq, hb, ack⟩ from j} ⟶                        Action 2: Receive a heartbeat
9 :     if (ack = seq_num_ij)                 Check if the ack is for latest local suspect list
10 :      msg_ij := ⟨0, null, seq⟩     Construct a lightweight HB (with piggybacked ack)
11 :    latest_ack_ij := seq                      Record the ack to be sent in the next HB
12 :    if (seq > max_seq_ij)                    Check if the suspect list from j is newer
13 :      hb_ij := hb                          Update hb_j with the new suspect list
14 :      max_seq_ij := seq              Update the max sequence number received from j
15 :    exec(◇P−receive ⟨hb_ij⟩ from j)    Send suspect list to the local ◇P-module
```

Alg 1.2. Bit-complexity optimizer for the $\diamond\mathcal{P}$ algorithm in Alg. 1.1

j in the variable $latest\_seq_{ij}$ (Action 2, line 11) and piggybacks, in the next heartbeat sent, the sequence number as the ack for the latest heartbeat received (Action 1, line 5 and Action 2, line 10). The action system continues to send heavyweight heartbeats until it receives an ack from the recipient process for the new heavyweight heartbeat (Action 2, lines 9–10). After the ack for the new heavyweight heartbeat is received, the action system starts sending lightweight heartbeats until the suspect list changes again.

At the receiver, the communication sub-layer maintains the latest suspect list received so far from each process j (in the variable hb_{ij}). Upon receiving a heavyweight heartbeat with a suspect list that is newer than the latest one on record (Action 2, line 12), the communication sub-layer updates its local information (Action 2, lines 13–14). It then sends the latest heartbeat on record (stored in the variable hb_{ij}) to the local $\diamond\mathcal{P}$ module (Action 2, line 15).

Correctness. We show that the proof of correctness in Sect. 4.1 applies to Alg. 1.1 + 1.2 as well. Inspection of the action system in Alg. 1.2 shows that the communication sub-layer does not change the number of messages sent or received in the system. It also ensures that the end-to-end communication delay for privileged messages is bounded in the number of action clock ticks (because messages are sent or received in a single atomic step).

Additionally, the heartbeat that is sent to the local $\diamond\mathcal{P}$-module is always a valid suspect list. This fact follows from the observation that the value of hb_{ij} is initialized to a valid suspect list, and the only modification of hb_{ij} is in lines 12–13. This change to hb_{ij} could result in an invalid suspect list (*viz.*, the value *null*) only when a lightweight heartbeat is received. However, all lightweight heartbeats are sent with sequence number 0, and hence, do not overwrite the existing (valid) suspect list. Consequently, the heartbeat sent to the local $\diamond\mathcal{P}$-module is always a valid suspect list.

Strong Completeness. The proof for Theorem 5 is agnostic to the contents of the heartbeats and is therefore applicable to Alg. 1.1 + 1.2 as well. Thus, by Theorem 5, Alg. 1.1 + 1.2 satisfies *strong completeness*.

Eventual Strong Accuracy. The proofs for Lemmas 6 through 12 are agnostic to the heartbeat content. Hence, these lemmas are applicable to Alg. 1.1 + 1.2 too.

Inspection of the proof for Theorem 13 reveals that the argument for eventual strong accuracy is made in the suffix in which all faulty processes have crashed and the suspect lists at all correct processes have stopped changing (by Lemma 12). Since Lemma 12 holds for Alg. 1.1 + 1.2 as well, every run of Alg. 1.1 + 1.2 has a suffix in which all faulty processes have crashed and the suspect lists at all correct processes have stopped changing. In such a suffix, each pair (i, j) of correct processes either (a) trust each other permanently, or (b) suspect each other permanently (by Lemmas 10 and 11).

In the former case, processes i and j send heavyweight heartbeats for the final change in their suspect lists until the acks for the reception of such a heavyweight heartbeat are received; after the reception of these acks, the j^{th} row in i's suspect matrix is the same as j's suspect list, and *vice versa*. In the latter case, i and j stop sending heartbeats to each other (Lemma 11). Thus, in this suffix, the suspect matrices S_i and S_j stay constant. The same arguments in the proof for Theorem 13 show that processes i and j always send heartbeats to each other infinitely often. Then Lemma 10 implies *eventual strong accuracy*.

Crash Quiescence. From the eventual strong accuracy property we know that every correct process is eventually and permanently trusted by all correct processes. Thus, the last change in the suspect list at each correct process is successfully communicated to all other correct processes in the system by Alg. 1.2. This allows us to apply the proof for Theorem 14 to Alg. 1.1 + 1.2, thus showing that Alg. 1.1 + 1.2 is crash quiescent in majority-correct runs.

5.1 Communication Bit Complexity

Alg. 1.2 uses sequence numbers in the heartbeats. Hence, in a finite prefix of the execution, heartbeat size may be unbounded. However, as we show next, eventually the processes send only lightweight heartbeats of size $\mathcal{O}(\log(n))$ bits.

Consider an infinite suffix of any run of Alg. 1.1 + 1.2 that starts after (a) all processes that crash in the run have already crashed, (b) all correct processes have started permanently suspecting crashed processes, and (c) no correct process is suspected by any correct process. In this suffix, the local suspect list at each process stops changing.

The finite number of heartbeats that were sent before the start of the suffix are either dropped or delivered in finite time. Subsequently, all heartbeats in transit in the system are ones that are sent during the suffix. Since a process continues to send heavyweight heartbeats until the sending process receives an ack for the latest change in the suspect list, sufficiently many heavyweight heartbeats for the final change in the suspect list are guaranteed to be sent. This ensures that these heartbeats are delivered to their recipients, and the acks for these heavyweight heartbeats are received by the senders.

Thus, eventually all processes have received acks for the last change in their suspect list from all correct processes. Consequently, eventually all processes send only lightweight heartbeats, and hence the piggyback acks are only for lightweight heartbeats as well. In other words, eventually the heartbeats sent by all correct processes are lightweight heartbeats with sequence number 0 and ack number 0. Such heartbeats require $\mathcal{O}(\log(n))$ bits (including the bits needed to encode the sender and the recipient identifier information).

In majority-correct runs, since the asymptotic message complexity is $\mathcal{O}(c^2)$, the communication bit complexity is $\mathcal{O}(c^2 \log(n))$. Similarly, in runs where half or more processes crash, since the asymptotic message complexity is $\mathcal{O}(n \cdot c)$, the communication bit complexity is $\mathcal{O}(n \cdot c \log(n))$.

The asymptotic communication bit complexity of Alg. 1.1 + 1.2 for majority-correct runs is lower than its non-crash-quiescent counterparts, and in runs where half or more processes crash, the asymptotic bit complexity is no worse than its non-crash-quiescent counterparts. Thus, we have achieved crash-quiescence for $\diamond\mathcal{P}$ in E_{CLPS} in majority-correct runs with improved message complexity and (importantly) improved bit complexity.

6 Conclusion

We have proposed a new property of distributed algorithms called crash quiescence. An algorithm is said to be crash quiescent if all correct processes eventually stop sending messages to any crashed process. We have motivated the importance of crash quiescence in the context of the eventually perfect failure detector $\diamond\mathcal{P}$. We have shown that in some partially-synchronous environments where a bounded, but unknown, number of consecutive messages may be arbitrarily late or lost, it is impossible to achieve crash quiescence for even $\diamond\mathcal{W}$ — the weakest failure detector in the Chandra-Toueg hierarchy. However, in such partially synchronous environments, we have presented an implementation of $\diamond\mathcal{P}$ that is correct in all runs and that is crash quiescent in runs where a majority of processes are correct. Furthermore, we have presented a refinement of our $\diamond\mathcal{P}$ algorithm to optimize the message size so that the resulting bit complexity per utilized link is asymptotically better than or equal to that of non-crash-quiescent counterparts.

References

1. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM **43**(2) (1996) 225–267
2. Mostefaoui, A., Mourgaya, E., Raynal, M.: An introduction to oracles for asynchronous distributed systems. Future Gener. Comput. Syst. **18**(6) (2002) 757–767
3. Aguilera, M.K., Chen, W., Toueg, S.: On quiescent reliable communication. SIAM Journal on Computing **29**(6) (2000) 2040–2073
4. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. Journal of the ACM **35**(2) (1988) 288–323
5. Sastry, S., Pike, S.M.: Eventually perfect failure detection using ADD channels. In: Proceedings of the 5th International Symposium on Parallel and Distributed Processing and Applications. (2007) 483–496
6. Sastry, S., Pike, S.M., Welch, J.L.: Crash fault detection in celerating environments. In: Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium. (2009) 1–12
7. Aguilera, M.K., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: Stable leader election. In: Proceedings of the 15th International Symposium on Distributed Computing. (2001) 108–122
8. Pike, S.M., Song, Y., Sastry, S.: Wait-free dining under eventual weak exclusion. In: Proceedings of the 9th International Conference on Distributed Computing and Networking. (2008) 135–146
9. Delporte-Gallet, C., Fauconnier, H., Guerraoui, R., Kouznetsov, P.: Mutual exclusion in asynchronous systems with failure detectors. Journal of Parallel and Distributed Computing **65**(4) (2005) 492–505
10. Mostéfaoui, A., Mourgaya, E., Raynal, M.: Asynchronous implementation of failure detectors. In: Proceedings of the 33rd International Conference on Dependable Systems and Networks. (2003) 351–360
11. 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. (2002) 354–363
12. 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. (1999) 34–49
13. Fetzer, C., Raynal, M., Tronel, F.: An adaptive failure detection protocol. In: Proceedings of the 7th Pacific Rim International Symposium on Dependable Computing. (2001) 146–153
14. 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. (2005) 271–280
15. 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. (2005) 495–496
16. Larrea, M., Fernández, A., Arévalo, S.: On the implementation of unreliable failure detectors in partially synchronous systems. IEEE Transactions on Computers **53**(7) (2004) 815–828