Using the Heartbeat Failure Detector for Quiescent Reliable Communication and Consensus in Partitionable Networks*

Marcos Kawazoe Aguilera Wei Chen Sam Toueg

Department of Computer Science
Upson Hall, Cornell University
Ithaca, NY 14853-7501, USA.
aguilera,weichen,sam@cs.cornell.edu

July 1997

Abstract

We consider *partitionable* networks with process crashes and lossy links, and focus on the problems of *reliable communication* and *consensus* for such networks. For both problems we seek algorithms that are *quiescent*, i.e., algorithms that eventually stop sending messages. We first tackle the problem of reliable communication for partitionable networks by extending the results of [ACT97a]. In particular, we generalize the specification of the heartbeat failure detector $\mathcal{HB}$, show how to implement it, and show how to use it to achieve quiescent reliable communication. We then turn our attention to the problem of consensus for partitionable networks. We first show that, even though this problem can be solved using a natural extension of failure detector $\diamond\mathcal{S}$, such solutions are not quiescent — in other words, $\diamond\mathcal{S}$ alone is not sufficient to achieve quiescent consensus in partitionable networks. We then solve this problem using $\diamond\mathcal{S}$ and the quiescent reliable communication primitives that we developed in the first part of the paper.

1 Introduction

We focus on the problems of *reliable communication* and *consensus* for asynchronous networks that may partition. For both problems we seek algorithms that are *quiescent*, i.e., algorithms that eventually stop sending messages.

We consider networks where processes may crash and communication links may lose messages. We assume that a lossy link is either *fair* or *eventually down*. Roughly speaking, a fair link may lose an infinite number of messages, but if a message is repeatedly sent then it is eventually received. A link is eventually down (we also say that it *eventually crashes*) if it eventually stops transporting messages. Links are unidirectional and the network is not necessarily completely connected. The network is partitionable: there may be two correct

*Research partially supported by NSF grants CCR-9402896 and CCR-9711403, by ARPA/ONR grant N00014-96-1-1014, and by an Olin Fellowship.

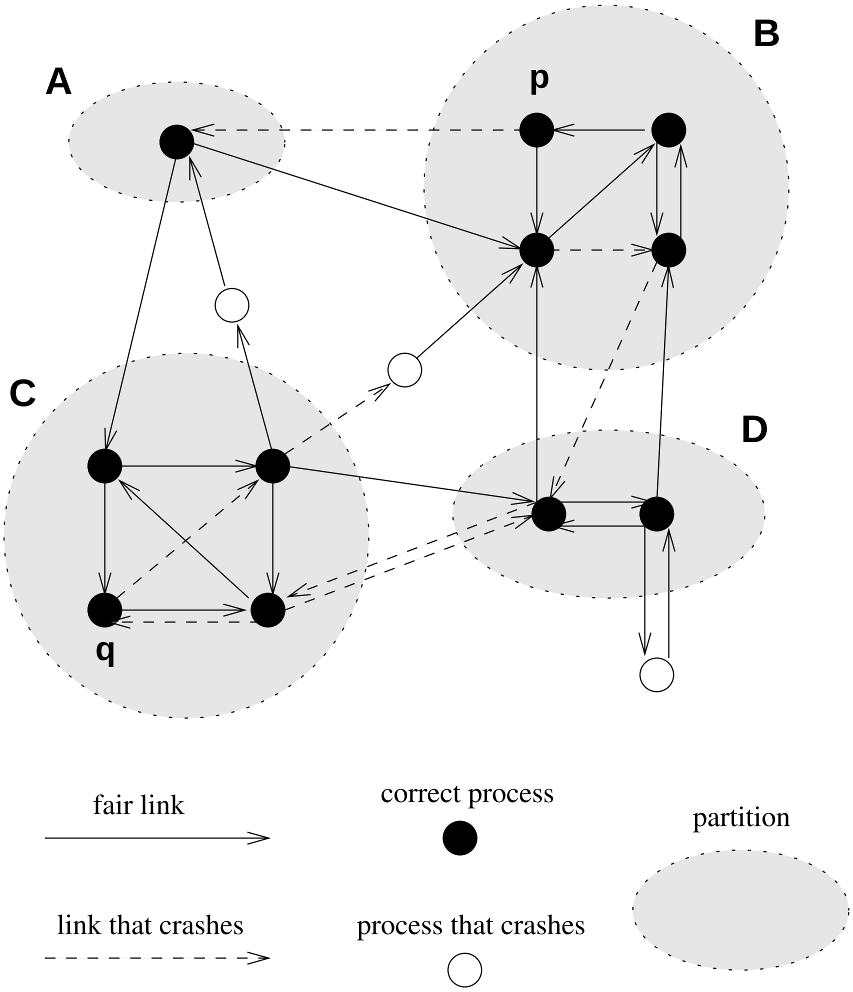

Figure 1: A network that partitions

processes p and q such that q is not reachable from p, i.e., there is no fair path from p to q.[1] A *partition* is a maximal set of processes that are mutually reachable from each other. We do not assume that partitions are eventually isolated: one partition may be able to receive messages from another, or to successfully send messages to another partition, forever.

An example of a network that partitions is given in Fig. 1. The processes that do not crash (black disks) are eventually divided into four partitions, A, B, C and D. Each partition is strongly connected through fair links (solid arrows). So processes in each partition can communicate with each other (but message losses can occur infinitely often). None of the partitions is isolated. For example, processes in D may continue to receive messages from processes in C (but not vice-versa); processes in D are able to send messages to processes in B; there is no fair path from C to A, or from D to C, etc.

[ACT97a] shows that without the help of failure detectors it is impossible to achieve quiescent reliable communication in the presence of process crashes and lossy links — even if one assumes that the network never partitions. In order to overcome this problem, [ACT97a] introduces the *heartbeat failure detector*

[1] A fair path is one consisting of correct processes and fair links.

(denoted $\mathcal{HB}$), and shows how it can be implemented, and how it can be used to achieve quiescent reliable communication. All these results are for networks that do not partition.

In this paper, we extend the above results to partitionable networks. In particular, we: (a) generalize the definitions of reliable communication primitives, (b) generalize the definition of the heartbeat failure detector $\mathcal{HB}$, (c) show how to implement $\mathcal{HB}$, and (d) use $\mathcal{HB}$ to achieve quiescent reliable communication.

We next consider the problem of consensus for partitionable networks, and focus on solving this problem with a quiescent algorithm.[2] In order to do so, we first generalize the traditional definition of consensus to partitionable networks. We also generalize the definition of $\diamond\mathcal{S}$ — the weakest failure detector for solving consensus in networks that do not partition [CHT96b].

We show that, although $\diamond\mathcal{S}$ can be used to solve consensus for partitionable networks, any such solution is not quiescent: Thus, *$\diamond\mathcal{S}$ alone is not sufficient to solve quiescent consensus for partitionable networks*. We then show that this problem can be solved using $\diamond\mathcal{S}$ together with $\mathcal{HB}$. In fact, our quiescent consensus algorithm for partitionable networks is identical to the one given in [CT96] for non-partitionable networks with reliable links: we simply replace the communication primitives used by the algorithm in [CT96] with the quiescent reliable communication primitives that we derive in this paper (the proof of correctness, however, is different).

An important remark on the use of failure detectors to achieve quiescence is now in order. Any reasonable implementation of a failure detector in a message-passing system is itself *not* quiescent: A process being monitored by a failure detector must periodically send a message to indicate that it is still alive, and it must do so forever (if it stops sending messages it cannot be distinguished from a process that has crashed). Given that failure detectors are not quiescent, does it still make sense to use them as a tool to achieve quiescent applications (such as quiescent reliable broadcast, consensus, or group membership)?

The answer is yes, for two reasons. First, a failure detector is intended to be a basic system service that is *shared* by many applications during the lifetime of the system, and so its cost is amortized over all these applications. Second, failure detection is a service that needs to be active forever — and so it is natural that it sends messages forever. In contrast, many applications (such as a single RPC call or the reliable broadcast of a single message) should not send messages forever, i.e., they should be quiescent. Thus, there is no conflict between the goal of building quiescent applications and the use of a (non-quiescent) shared failure detection service as a tool to achieve this goal.

Organization of the Paper

The rest of the paper is organized as follows. In Section 2, we explain our model of partitionable networks, and of failure detection for such networks. In Section 3, we extend the definition of the failure detector $\mathcal{HB}$ to partitionable networks. In Section 4, we define reliable communication primitives for partitionable networks, and give quiescent implementations that use $\mathcal{HB}$. We then turn our attention to the consensus problem in Section 5. We first define this problem for partitionable networks (Section 5.1), and extend the definition of the failure detector $\diamond\mathcal{S}$ (Section 5.2). We then show that $\diamond\mathcal{S}$ is not sufficient to achieve quiescent consensus in partitionable networks (Section 5.3), and give a quiescent implementation that uses both $\diamond\mathcal{S}$ and $\mathcal{HB}$ (Section 5.4). In Section 6, we show how to implement $\mathcal{HB}$ in partitionable networks. Some practical issues are briefly addressed in Section 7. We conclude with a short discussion of related work (Section 8) and a comparison with other models (Section 9).

[2] The consensus algorithms for partitionable networks given in [FKM$^+$95, CHT96a, DFKM96] are *not* quiescent.

2 Model

We consider asynchronous message-passing distributed systems in which there are no timing assumptions. In particular, we make no assumptions on the time it takes to deliver a message, or on relative process speeds. Processes can communicate with each other by sending messages through unidirectional links. The system can experience both process failures and link failures. Processes can fail by crashing, and links can fail by crashing or by intermittently dropping messages (while remaining fair). Failures may cause permanent network partitions. The model, based on the one in [CHT96b], is described next.

A network is a directed graph $G = (\Pi, \Lambda)$ where $\Pi = \{1, \ldots, n\}$ is the set of processes, and $\Lambda \subseteq \Pi \times \Pi$ is the set of links. If there is a link from process p to process q, we denote this link by $p \rightarrow q$, and if, in addition, $q \neq p$ we say that q is a *neighbor* of p. The set of neighbors of p is denoted by $neighbor(p)$.

We assume the existence of a discrete global clock — this is merely a fictional device to simplify the presentation and processes do not have access to it. We take the range $\mathcal{T}$ of the clock's ticks to be the set of natural numbers.

2.1 Failures and Failure Patterns

Processes can fail by crashing, i.e., by halting prematurely. A *process failure pattern* F_P is a function from $\mathcal{T}$ to 2^{Π}. Intuitively, $F_P(t)$ denotes the set of processes that have crashed through time t. Once a process crashes, it does not "recover", i.e., $\forall t : F_P(t) \subseteq F_P(t+1)$. We define $crashed(F_P) = \bigcup_{t \in \mathcal{T}} F_P(t)$ and $correct(F_P) = \Pi \setminus crashed(F_P)$. If $p \in crashed(F_P)$ we say p *crashes (or is faulty) in* F_P and if $p \in correct(F_P)$ we say p is *correct in* F_P.

We assume that the network has two types of links: links that are fair and links that crash. Roughly speaking, a fair link $p \rightarrow q$ may intermittently drop messages, and do so infinitely often, but if p repeatedly sends some message to q and q does not crash, then q eventually receives that message. If link $p \rightarrow q$ crashes, then it eventually stops transporting messages. Link properties are made precise in Section 2.5.

A *link failure pattern* F_L is a function from $\mathcal{T}$ to 2^{Λ}. Intuitively, $F_L(t)$ is the set of links that have crashed through time t. Once a link crashes, it does not "recover", i.e., $\forall t : F_L(t) \subseteq F_L(t+1)$. We define $crashed(F_L) = \bigcup_{t \in \mathcal{T}} F_L(t)$. If $p \rightarrow q \in crashed(F_L)$, we say that $p \rightarrow q$ *crashes (or is eventually down) in* F_L. If $p \rightarrow q \notin crashed(F_L)$, we say that $p \rightarrow q$ is *fair in* F_L.

A *failure pattern* $F = (F_P, F_L)$ combines a process failure pattern and a link failure pattern.

2.2 Connectivity

In contrast to [ACT97a], the network is *partitionable*: there may be two correct processes p and q such that q is not reachable from p (Fig. 1). Intuitively, a partition is a maximal set of processes that are mutually reachable from each other. We do not assume that partitions are eventually isolated: one partition may be able to receive messages from another, or to successfully send messages to another partition, forever. This is made more precise below.

The following definitions are with respect to a given failure pattern $F = (F_P, F_L)$. We say that a path $(p_1, \ldots, p_k)$ in the network is *fair* if processes $p_1, \ldots, p_k$ are correct and links $p_1 \rightarrow p_2, \ldots, p_{k-1} \rightarrow p_k$ are fair. We say process q is *reachable from process* p if there is a fair path from p to q.[3] If p and q are

[3] We allow singleton paths of the form (p). Since fair paths contain only correct processes, p is reachable from itself if and only

both reachable from each other, we write $p \rightleftharpoons_F q$. Note that $\rightleftharpoons_F$ is an equivalence relation on the set of correct processes. The equivalence classes are called *partitions*. The partition of a process p (with respect to F) is denoted $partition_F(p)$. For convenience, if p is faulty we define $partition_F(p) = \emptyset$. The set of all non-empty partitions is denoted by $Partitions_F$. The subscript F in the above definitions is omitted whenever it is clear from the context.

2.3 Failure Detectors

Each process has access to a local failure detector module that provides (possibly incorrect) information about the failure pattern that occurs in an execution. A *failure detector history H with range $\mathcal{R}$* is a function from $\Pi \times \mathcal{T}$ to $\mathcal{R}$. $H(p, t)$ is the output value of the failure detector module of process p at time t. *A failure detector $\mathcal{D}$* is a function that maps each failure pattern F to a *set* of failure detector histories with range $\mathcal{R}_\mathcal{D}$ (where $\mathcal{R}_\mathcal{D}$ denotes the range of the failure detector output of $\mathcal{D}$). $\mathcal{D}(F)$ denotes the set of possible failure detector histories permitted by $\mathcal{D}$ for the failure pattern F.

2.4 Algorithms and Runs

An algorithm A is a collection of n deterministic automata, one for each process in the system. Computation proceeds in atomic *steps* of A. In each step, a process may: receive a message from a process, get an external input, query its failure detector module, undergo a state transition, send a message to a neighbor, and issue an external output.

A *run of algorithm A using failure detector $\mathcal{D}$* is a tuple $R = (F, H_\mathcal{D}, I, S, T)$ where $F = (F_P, F_L)$ is a failure pattern, $H_\mathcal{D} \in \mathcal{D}(F)$ is a history of failure detector $\mathcal{D}$ for failure pattern F, I is an initial configuration of A, S is an infinite sequence of steps of A, and T is an infinite list of strictly increasing time values indicating when each step in S occurs.

A run must satisfy some properties for every process p: If p has crashed by time t, i.e., $p \in F_P(t)$, then p does not take a step at any time $t' \geq t$; if p is correct, i.e., $p \in correct(F_P)$, then p takes an infinite number of steps; if p takes a step at time t and queries its failure detector, then p gets $H_\mathcal{D}(p, t)$ as a response.

The correctness of an algorithm may depend on certain assumptions on the "environment", e.g., the maximum number of processes and/or links that may crash. For example, in Section 5.4, we give a consensus algorithm that assumes that a majority of processes are in the same network partition. Formally, an *environment* $\mathcal{E}$ is a set of failure patterns.

A problem P is defined by properties that sets of runs must satisfy. An algorithm A solves problem P using a failure detector $\mathcal{D}$ in environment $\mathcal{E}$ if the set of all runs $R = (F, H_\mathcal{D}, I, S, T)$ of A using $\mathcal{D}$ where $F \in \mathcal{E}$ satisfies the properties required by P. Let $\mathcal{C}$ be a class of failure detectors. An algorithm A solves a problem P using $\mathcal{C}$ in environment $\mathcal{E}$ if for all $\mathcal{D} \in \mathcal{C}$, A solves P using $\mathcal{D}$ in $\mathcal{E}$. An algorithm implements $\mathcal{C}$ in environment $\mathcal{E}$ if it implements some $\mathcal{D} \in \mathcal{C}$ in $\mathcal{E}$. Unless otherwise stated, we put no restrictions on the environment (i.e., $\mathcal{E}$ is the set of all possible failure patterns) and we do not refer to it.

if it is correct.

2.5 Link Properties

So far we have put no restrictions on how links behave in a run (e.g., processes may receive messages that were never sent, etc.). As we mentioned before, we want to model networks that have two types of links: links that are fair and links that crash. We therefore require that in each run $R = (F, H_{\mathcal{D}}, I, S, T)$ the following properties hold for every link $p \to q \in \Lambda$:

- [*Uniform Integrity*] for all $k \geq 1$, if q receives a message m from p exactly k times by time t, then p sent m to q at least k times before time t;
- If $p \to q \notin \mathit{crashed}(F_L)$: [*Fairness*] if p sends a message m to q an infinite number of times and q is correct, then q receives m from p an infinite number of times.

  If $p \to q \in \mathit{crashed}(F_L)$: [*Finite Receipt*] q receives messages from p only a finite number of times.[4]

Uniform Integrity ensures that a link does not create or duplicate messages. Fairness ensures that if a link does not crash then it eventually transports any message that is repeatedly sent through it. Finite Receipt implies that if a link crashes then it eventually stops transporting messages.

3 The Heartbeat Failure Detector $\mathcal{HB}$ for Partitionable Networks

One of our goals is to achieve quiescent reliable communication in partitionable networks with process crashes and message losses. In [ACT97a] it is shown that without failure detectors this is impossible, even if one assumes that the network does not partition. In order to circumvent this impossibility result, [ACT97a] introduces the heartbeat failure detector, denoted $\mathcal{HB}$, for non-partitionable networks. In this section, we generalize the definition of $\mathcal{HB}$ to partitionable networks. We then show how to implement it in Section 6.

$\mathcal{HB}$ is different from the failure detectors defined in [CT96], or those currently in use in many systems (even though some existing systems, such as Ensemble and Phoenix, use the same name *heartbeat* in their failure detector implementations [vR97, Cha97]). In contrast to existing failure detectors, $\mathcal{HB}$ is implementable in asynchronous systems, without the use of timeouts (see Section 6).

A *heartbeat failure detector $\mathcal{D}$ (for partitionable networks)* has the following features. The output of $\mathcal{D}$ at each process p is an array $(v_1, v_2, \ldots, v_n)$ with one nonnegative integer for each process in Π.[5] Intuitively, v_q increases if process q is in the partition of p, and stops increasing otherwise. We say that v_q is *the heartbeat value of process q at p*. The *heartbeat sequence of q at p* is the sequence of the heartbeat values of q at p as time increases. $\mathcal{D}$ satisfies the following properties:

- *$\mathcal{HB}$-Completeness*: At each correct process p, the heartbeat sequence of every process not in the partition of p is bounded. Formally:

$$\forall F = (F_P, F_L), \forall H \in \mathcal{D}(F), \forall p \in \mathit{correct}(F_P), \forall q \in \Pi \setminus \mathit{partition}_F(p),$$
$$\exists K \in \mathbb{N}, \forall t \in \mathcal{T} : H(p,t)[q] \leq K$$

[4]We could have required a stronger property: if $p \to q$ has crashed by time t, i.e., $p \to q \in F_L(t)$, then q does not receive any message sent by p at time $t' \geq t$. This stronger property is not necessary in this paper.

[5]In [ACT97a], the output of $\mathcal{D}$ at p is an array with one nonnegative integer for each *neighbor* of p.

- $\mathcal{HB}$-*Accuracy*:
  - At each process p, the heartbeat sequence of every process is nondecreasing. Formally:

    $$\forall F, \forall H \in \mathcal{D}(F), \forall p \in \Pi, \forall q \in \Pi, \forall t \in \mathcal{T} : H(p,t)[q] \leq H(p,t+1)[q]$$

  - At each correct process p, the heartbeat sequence of every process in the partition of p is unbounded. Formally:

    $$\forall F = (F_P, F_L), \forall H \in \mathcal{D}(F), \forall p \in \mathit{correct}(F_P), \forall q \in \mathit{partition}_F(p),$$
    $$\forall K \in \mathbb{N}, \exists t \in \mathcal{T} : H(p,t)[q] > K$$

The class of all heartbeat failure detectors is denoted $\mathcal{HB}$. By a slight abuse of notation, we sometimes use $\mathcal{HB}$ to denote a (generic) member of that class.

The output of $\mathcal{HB}$ is a vector of unbounded counters. In contrast, the output of failure detectors that are commonly used in practice has bounded size: it is just a list of processes suspected to have crashed. Some remarks are now in order regarding the necessity and practicality of $\mathcal{HB}$'s unbounded output.

$\mathcal{HB}$ can be used to solve the problem of quiescent reliable communication and it is implementable in asynchronous systems, but its counters are unbounded. Can we solve this problem using a failure detector that is both implementable and has bounded output? The answer is no: in [ACT97b] we show that a failure detector with bounded output size is either (a) too weak to achieve quiescent reliable communication, or (b) not implementable. This shows that failure detectors that are commonly used in practice, i.e., those that output only lists of suspects, are not always the best ones to solve a problem: their power or applicability is limited. Thus, the difference between $\mathcal{HB}$ and existing failure detectors is more than "skin deep".

In practice, the unbounded counters of $\mathcal{HB}$ are not a problem for the following reasons. First, they are in *local memory* and not in messages — the implementation of $\mathcal{HB}$ shown in Section 6 uses bounded messages. Second, if we bound each local counter to 64 bits, and assume a rate of one heartbeat per nanosecond, which is orders of magnitude higher than currently used in practice, then $\mathcal{HB}$ will work for more than 500 years.

4 Reliable Communication for Partitionable Networks

There are two types of basic communication primitives: point-to-point and broadcast. We first define reliable broadcast for partitionable networks, and give a quiescent implementation that uses $\mathcal{HB}$. We then consider point-to-point reliable communication.

4.1 Reliable Broadcast: Specification

Reliable broadcast for partitionable networks is defined in terms of two primitives: broadcast(m) and deliver(m). We say that process *p broadcasts message* m if p invokes broadcast(m). We assume that every broadcast message m includes the following fields: the identity of its sender, denoted $\mathit{sender}(m)$, and a sequence number, denoted $\mathit{seq}(m)$. These fields make every message unique. We say that *q delivers message* m if q returns from the invocation of deliver(m). Primitives broadcast and deliver satisfy the following properties:[6]

[6]This specification is a generalization of the one for non-partitionable networks given in [HT94].

- *Validity*: If a correct process broadcasts a message m, then it eventually delivers m.
- *Agreement*: If a correct process p delivers a message m, then all processes *in the partition of* p eventually deliver m.
- *Uniform Integrity*: For every message m, every process delivers m at most once, and only if m was previously broadcast by $sender(m)$.
- *Partition Integrity*: If a process q delivers an infinite number of messages broadcast by a process p, then q is reachable from p.

Validity and Agreement imply that if a correct process p broadcasts a message m, then all processes in the partition of p eventually deliver m.

We want to implement broadcast and deliver using the communication service provided by the network links (which are described in Section 2.5). Informally, an implementation of reliable broadcast is *quiescent* if it sends only a finite number of messages when broadcast is invoked a finite number of times.[7]

4.2 Reliable Broadcast: Algorithm Using $\mathcal{HB}$

The quiescent implementation of reliable broadcast for partitionable network that we give here is identical to the one given in [ACT97a] for non-partitionable networks. However, the network assumptions, the reliable broadcast requirements, and the failure detector properties are different, and so its proof of correctness and quiescence changes.

This implementation, which uses $\mathcal{HB}$, has the following desirable feature: processes do not need to know the entire network topology or the number of processes in the system; they only need to know the identity of their neighbors. Moreover, each process only needs to know the heartbeats of its neighbors.

The implementation of reliable broadcast is shown in Fig. 2. $\mathcal{D}_p$ denotes the current output of the failure detector $\mathcal{D}$ at process p. All variables are local to each process. In the following, when ambiguities may arise, a variable local to process p is subscripted by p. For each message m that is reliably broadcast, each process p maintains a variable $got_p[m]$ containing a set of processes. Intuitively, a process q is in $got_p[m]$ if p has evidence that q has delivered m. All the messages sent by a process p in the reliable broadcast algorithm are of the form $(m, got\_msg, path)$ where got_msg is the current value of $got_p[m]$, and *path* is the sequence of processes that this copy of $(m, got\_msg, path)$ has traversed so far.

In order to reliably broadcast a message m, p first delivers m; then p initializes variable $got_p[m]$ to $\{p\}$ and forks task $diffuse(m)$; finally p returns from the invocation of broadcast(m). The task $diffuse(m)$ runs in the background. In this task, p periodically checks if, for some neighbor $q \notin got_p[m]$, the heartbeat of q at p has increased and, if so, p sends $(m, got_p[m], p)$ to all neighbors whose heartbeat increased — even to those who are already in $got_p[m]$.[8] The task terminates when all neighbors of p are contained in $got_p[m]$.

Upon the receipt of a message $(m, got\_msg, path)$, process p first checks if it has already delivered m and, if not, it delivers m and forks task $diffuse(m)$. Then p adds the contents of got_msg to $got_p[m]$ and appends

[7] A quiescent implementation is allowed to send a finite number of messages even if no broadcast is invoked at all (e.g., some messages may be sent as part of an "initialization phase").

[8] It may appear that p does not need to send this message to processes in $got_p[m]$, since they already got m! But with this "optimization" the algorithm is no longer quiescent; we will indicate exactly where the sending to *every* neighbor whose heartbeat increased is necessary in the proof of Lemma 9.

```
1  For every process p:
2
3      To execute broadcast(m):
4          deliver(m)
5          got[m] ← {p}
6          fork task diffuse(m)
7          return
8
9      task diffuse(m):
10         for all q ∈ neighbor(p) do prev_hb[q] ← −1
11         repeat periodically
12             hb ← D_p                                                        {query HB}
13             if for some q ∈ neighbor(p), q ∉ got[m] and prev_hb[q] < hb[q] then
14                 for all q ∈ neighbor(p) such that prev_hb[q] < hb[q] do send (m, got[m], p) to q
15                 prev_hb ← hb
16         until neighbor(p) ⊆ got[m]
17
18     upon receive (m, got_msg, path) from q do
19         if p has not previously executed deliver(m) then
20             deliver(m)
21             got[m] ← {p}
22             fork task diffuse(m)
23         got[m] ← got[m] ∪ got_msg
24         path ← path · p
25         for all q such that q ∈ neighbor(p) and q appears at most once in path do
26             send (m, got[m], path) to q
```

Figure 2: Quiescent implementation of broadcast and deliver using $\mathcal{HB}$

itself to $path$. Finally, p forwards the new message $(m, got_p[m], path)$ to all its neighbors that appear at most once in $path$.

The code consisting of lines 18–26 is executed atomically.[9] Moreover, if there are several concurrent executions of the *diffuse* task (lines 9 to 16), then each execution must have its own private copy of all the local variables in this task, namely m, *hb*, and *prev_hb*.

We now show that this implementation is correct and quiescent. The proofs of the first few lemmata are obvious and therefore omitted.

Lemma 1 (Uniform Integrity) *For every message m, every process delivers m at most once, and only if m was previously broadcast by sender(m).*

Lemma 2 (Validity) *If a correct process broadcasts a message m, then it eventually delivers m.*

Lemma 3 (Partition Integrity) *If a process q delivers an infinite number of messages broadcast by a process p, then q is reachable from p.*

[9] A process p executes a region of code atomically if at any time there is at most one thread of p in this region.

Lemma 4 *For any processes p and q, (1) if at some time t, $q \in got_p[m]$, then at every time $t' \geq t$, $q \in got_p[m]$; (2) When $got_p[m]$ is initialized, $p \in got_p[m]$; (3) if $q \in got_p[m]$ then q delivered m.*

Lemma 5 *For every m and path, there is a finite number of distinct messages of the form $(m, *, path)$.*

Lemma 6 *If some process sends a message of the form $(m, *, path)$, then no process appears more than twice in path.*

Lemma 7 *Suppose link $p \to q$ is fair, and p and q are in the same partition. If p delivers a message m, then q eventually delivers m.*

Proof. Suppose for a contradiction that p delivers m and q never delivers m. Since p and q are in the same partition, they are both correct. Therefore, p forks task $\mathit{diffuse}(m)$. Since q does not deliver m, by Lemma 4 part (3) q never belongs to $got_p[m]$. Because p is correct and q is a neighbor of p, this implies that p executes the loop in lines 11–16 an infinite number of times. Since q is in the partition of p, the $\mathcal{HB}$-Accuracy property guarantees that the heartbeat sequence of q at p is nondecreasing and unbounded. Thus, the condition in line 13 evaluates to true an infinite number of times. Therefore, p executes line 14 infinitely often. So p sends a message of the form $(m, *, p)$ to q infinitely often. By Lemma 5, there exists a subset $g_0 \subseteq \Pi$ such that p sends message (m, g_0, p) infinitely often to q. Since q is correct and link $p \to q$ is fair, q eventually receives (m, g_0, p). Therefore, q delivers m, a contradiction. $\Box$

Lemma 8 (Agreement) *If a correct process p delivers a message m, then all processes in the partition of p eventually deliver m.*

Proof (Sketch). For every process q in the partition of p, there is a fair path from p to q. The result follows from successive applications of Lemma 7 over the links of this path. $\Box$

We now show that the implementation in Fig. 2 is quiescent. In order to do so, we focus on a single invocation of broadcast and show that it causes the sending of only a finite number of messages in the network. This implies that a finite number of invocations of broadcast cause the sending of only a finite number of messages.

Let m be a message and consider an invocation of broadcast(m). This invocation can only cause the sending of messages of form $(m, *, *)$. Thus, all we need to show is that every process eventually stops sending messages of this form.

Lemma 9 *Let p be a process and q be a neighbor of p with $q \in partition(p)$. If p forks task $\mathit{diffuse}(m)$, then eventually condition $q \in got_p[m]$ holds forever.*

Proof. By Lemma 4 part (1), we only need to show that eventually q belongs to $got_p[m]$. Suppose, for a contradiction, that q never belongs to $got_p[m]$. Since p and q are in the same partition, they are correct and there exist both a simple fair path[10] $(p_1, p_2, \ldots, p_{k'})$ from p to q with $p_1 = p$ and $p_{k'} = q$, and a simple fair path $(p_{k'}, p_{k'+1}, \ldots, p_k)$ from q to p with $p_k = p$. For $1 \leq j < k$, let $P_j = (p_1, p_2, \ldots, p_j)$. Note that a process can appear at most twice in P_k. Thus, for $1 \leq j < k$, process p_{j+1} appears at most once in P_j. Moreover, for every $j \in \{1, \ldots, k\}$, $p_j \in partition(p)$.

We claim that for every $j \in \{1, \ldots, k-1\}$, there is a set g_j containing $\{p_1, p_2, \ldots, p_j\}$ such that p_j sends (m, g_j, P_j) to p_{j+1} an infinite number of times. For $j = k-1$, this claim together with the Fairness property

[10] A path is *simple* if all processes in that path are distinct.

of link $p_{k-1} \rightarrow p_k$ immediately implies that $p_k = p$ eventually receives (m, g_{k-1}, P_{k-1}). Upon the receipt of such a message, p adds the contents of g_{k-1} to its variable $got_p[m]$. Since g_{k-1} contains $p_{k'} = q$, this contradicts the fact that q never belongs to $got_p[m]$.

We show the claim by induction on j. For the base case, note that q never belongs to $got_p[m]$ and q is a neighbor of $p_1 = p$, and so p_1 executes the loop in lines 11–16 an infinite number of times. Furthermore, since q is in the partition of p_1, the $\mathcal{HB}$-Accuracy property guarantees that the heartbeat sequence of q at p_1 is nondecreasing and unbounded. This implies that the condition in line 13 evaluates to true an infinite number of times. So p_1 executes line 14 infinitely often. Since p_2 is in the partition of p_1, its heartbeat sequence is nondecreasing and unbounded. Together with the fact that p_2 is a neighbor of p_1, this implies that p_1 sends messages of the form $(m, *, p_1)$ to p_2 an infinite number of times.[11] By Lemma 5, there is some g_1 such that p_1 sends (m, g_1, p_1) to p_2 an infinite number of times. Parts (1) and (2) of Lemma 4 imply that $p_1 \in g_1$. This shows the base case.

For the induction step, suppose that for $j < k - 1$, p_j sends (m, g_j, P_j) to p_{j+1} an infinite number of times, for some set g_j containing $\{p_1, p_2, \ldots, p_j\}$. By the Fairness property of the link $p_j \rightarrow p_{j+1}$, p_{j+1} receives (m, g_j, P_j) from p_j an infinite number of times. Since p_{j+2} is a neighbor of p_{j+1} and appears at most once in P_{j+1}, each time p_{j+1} receives (m, g_j, P_j), it sends a message of the form $(m, *, P_{j+1})$ to p_{j+2}. It is easy to see that each such message is (m, g, P_{j+1}) for some g that contains both g_j and $\{p_{j+1}\}$. By Lemma 5, there exists $g_{j+1} \subseteq \Pi$ such that g_{j+1} contains $\{p_1, p_2, \ldots, p_{j+1}\}$ and p_{j+1} sends (m, g_{j+1}, P_{j+1}) to p_{j+2} an infinite number of times. ◻

Corollary 10 *If a correct process p forks task diffuse(m), then eventually p stops sending messages in task diffuse(m).*

Proof. For every neighbor q of p, there are two cases. If q is in the partition of p then eventually condition $q \in got_p[m]$ holds forever by Lemma 9. If q is not in the partition of p, then the $\mathcal{HB}$-Completeness property guarantees that the heartbeat sequence of q at p is bounded, and so eventually condition $prev\_hb_p[q] \geq hb_p[q]$ holds forever. Therefore, there is a time after which the guard in line 13 is always false. Hence, p eventually stops sending messages in task $diffuse(m)$. ◻

Lemma 11 (Quiescence) *Eventually every process stops sending messages of the form $(m, *, *)$.*

Proof. Suppose, for a contradiction, that the lemma is not true. Then there exists a process p such that p never stops sending messages of the form $(m, *, *)$. By Lemma 6, the third component of a message of the form $(m, *, *)$ ranges over a finite set of values. Therefore, there is some fixed $path$ such that p sends an infinite number of messages of the form $(m, *, path)$.

Now let $path_0$ to be the shortest path such that there exists some process p_0 that sends messages of the form $(m, *, path_0)$ an infinite number of times. Note that p_0 must be correct. Corollary 10 shows that there is a time after which p_0 stops sending messages in its task $diffuse(m)$. Since p_0 only sends a message in task $diffuse(m)$ or in line 26, then p_0 sends messages of the form $(m, *, path_0)$ in line 26 an infinite number of times. For each $(m, *, path_0)$ that p_0 sends in line 26, p_0 must have previously received a message of the form $(m, *, path_1)$ such that $path_0 = path_1 \cdot p_0$. So p_0 receives a message of the form $(m, *, path_1)$ an infinite number of times. By the Uniform Integrity property of the links, some process p_1 sends a message of form $(m, *, path_1)$ to p_0 an infinite number of times. But $path_1$ is shorter than $path_0$ — a contradiction to the minimality of $path_0$. ◻

From Lemmata 1, 2, 3, 8, and 11 we have:

[11] This is where the proof uses the fact that p sends a message containing m to all its neighbors whose heartbeat increased — even to those (such as p_2) that may already be in $got_p[m]$ (cf. line 14 of the algorithm).

Theorem 12 *For partitionable networks, Fig. 2 shows a quiescent implementation of reliable broadcast that uses $\mathcal{HB}$.*

We next consider point-to-point reliable communication for partitionable networks.

4.3 Quasi Reliable Send and Receive for Partitionable Networks

Consider any two distinct processes s and r. We define *quasi reliable send and receive from s to r (for partitionable networks)* in terms of two primitives: $\textsf{qr-send}_{s,r}$ and $\textsf{qr-receive}_{r,s}$. We say that process s qr-sends *message m to process r* if s invokes $\textsf{qr-send}_{s,r}(m)$. We assume that if s is correct, it eventually returns from this invocation. We allow process s to qr-send the same message m more than once through the same link. We say that process r qr-receives *message m from process s* if r returns from the invocation of $\textsf{qr-receive}_{r,s}(m)$. Primitives $\textsf{qr-send}_{s,r}$ and $\textsf{qr-receive}_{r,s}$ satisfy the following properties:[12]

- *Quasi No Loss*: For all $k \geq 1$, if s and r are *in the same partition*, and s qr-sends m to r exactly k times by time t, then r eventually qr-receives m from s at least k times.
- *Uniform Integrity*: For all $k \geq 1$, if r qr-receives m from s exactly k times by time t, then s qr-sent m to r at least k times before time t.
- *Partition Integrity*: If r qr-receives messages from s an infinite number of times then r is reachable from s.

Intuitively, Quasi No Loss together with Uniform Integrity implies that if s and r are in the same partition, then r qr-receives m from s exactly as many times as s qr-sends m to r.

We want to implement $\textsf{qr-send}_{s,r}$ and $\textsf{qr-receive}_{r,s}$ using the communication service provided by the network links. Informally, such an implementation is *quiescent* if it sends only a finite number of messages when $\textsf{qr-send}_{s,r}$ is invoked a finite number of times.

Given any quiescent implementation of reliable broadcast (such as the one given in the previous section), we can obtain a quiescent implementation of $\textsf{qr-send}_{p,q}$ and $\textsf{qr-receive}_{q,p}$ for every pair of processes p and q. The implementation works as follows: to qr-send a message m to q, p simply broadcasts the message $M = (m, p, q, k)$ using the given quiescent implementation of reliable broadcast, where $sender(M) = p$ and $seq(M) = k$, a sequence number that p has not used before. Upon the delivery of $M = (m, p, q, k)$, a process r qr-receives m from p if $r = q$, and discards m otherwise. This implementation of $\textsf{qr-send}_{p,q}$ and $\textsf{qr-receive}_{q,p}$ is clearly correct and quiescent. Thus, from Theorem 12, we have:

Corollary 13 *For partitionable networks, quasi reliable send and receive between every pair of processes can be implemented with a quiescent algorithm that uses $\mathcal{HB}$.*

5 Consensus for Partitionable Networks

5.1 Specification

We now define the problem of consensus for partitionable networks as a generalization of the standard definition for non-partitionable networks. Roughly speaking, some processes propose a value and must

[12]This specification is a generalization of the one for non-partitionable networks given in [ACT97a].

decide on one of the proposed values [FLP85]. More precisely, consensus is defined in terms of two primitives, $\mathsf{propose}(v)$ and $\mathsf{decide}(v)$, where v is a value drawn from a set of possible proposed values. When a process invokes $\mathsf{propose}(v)$, we say that it *proposes* v. When a process returns from the invocation of $\mathsf{decide}(v)$, we say that it *decides* v.

The *largest partition* is defined to be the one with the largest number of processes (if more than one such partition exists, pick the one containing the process with the largest process id). The *consensus problem (for partitionable networks)* is specified as follows:

- *Agreement*: No two processes in the same partition decide differently.
- *Uniform Validity*: A process can only decide a value that was previously proposed by some process.
- *Uniform Integrity*: Every process decides at most once.
- *Termination*: If all processes in the largest partition propose a value, then they all eventually decide.

Stronger versions of consensus may also require one or both of the following properties:

- *Uniform Agreement*: No two processes (whether in the same partition or not) decide differently.
- *Partition Termination*: If a process decides then every process in its partition decides.

The consensus algorithm given in Section 5.4 satisfies the above two properties, while the impossibility result in Section 5.3 holds for the weaker version of consensus.

Informally, an implementation of consensus is *quiescent* if each execution of consensus causes the sending of only a finite number of messages throughout the network. This should hold even for executions where only a subset of the correct processes actually propose a value (the others may not wish to run consensus).

5.2 $\diamond\mathcal{S}$ for Partitionable Networks

It is well known that consensus cannot be solved in asynchronous systems, even if at most one process may crash and the network is completely connected with reliable links [FLP85]. To overcome this problem, Chandra and Toueg introduced unreliable failure detectors in [CT96]. In this paper, we focus on the class of eventually strong failure detectors (the weakest one for solving consensus in non-partitionable networks [CHT96b]), and extend it to partitionable networks.[13]

At each process p, an eventually strong failure detector outputs a set of processes. In [CT96], these are the processes that p suspects to have crashed. In our case, these are the processes that p suspects to be outside its partition. More precisely, an *eventually strong failure detector* $\mathcal{D}$ *(for partitionable networks)* satisfies the following properties (in the following, we say that a process p *trusts* process q, if its failure detector does not suspect q):

- *Strong Completeness*: For every partition P, there is a time after which every process that is not in P is permanently suspected by every process in P. Formally:

$$\forall F, \forall H \in \mathcal{D}(F), \forall P \in \mathit{Partitions}_F, \exists t \in \mathcal{T}, \forall p \notin P, \forall q \in P, \forall t' \geq t : p \in H(q, t')$$

[13] The other classes of eventual failure detectors introduced in [CT96] can be generalized in a similar way.

- *Eventual Weak Accuracy*: For every partition P, there is a time after which some process in P is permanently trusted by every process in P. Formally:

$$\forall F, \forall H \in \mathcal{D}(F), \forall P \in \mathit{Partitions}_F, \exists t \in \mathcal{T}, \exists p \in P, \forall t' \geq t, \forall q \in P : p \notin H(q, t')$$

The class of all failure detectors that satisfy the above two properties is denoted $\Diamond\mathcal{S}$.

A weaker class of failure detectors, denoted $\Diamond\mathcal{S}_{LP}$, is obtained by defining the *largest partition* as in Section 5.1, and replacing "For every partition P" with "For the largest partition P" in the two properties above (this definition is similar to one given in [DFKM96]). Note that $\Diamond\mathcal{S}_{LP}$ does not impose *any* requirement on the failure detector modules of processes in "small" partitions. To strengthen our results, we use $\Diamond\mathcal{S}$ for the impossibility result (Section 5.3), and $\Diamond\mathcal{S}_{LP}$ for the consensus algorithm (Section 5.4).

By a slight abuse of notation, we sometimes use $\Diamond\mathcal{S}$ and $\Diamond\mathcal{S}_{LP}$ to refer to an arbitrary member of the respective class.

5.3 Quiescent Consensus for Partitionable Networks Cannot be Achieved using $\Diamond\mathcal{S}$

Although consensus for partitionable networks can be solved using $\Diamond\mathcal{S}$, we now show that any such solution is *not* quiescent (the consensus algorithms in [CHT96a, DFKM96] do not contradict this result because they are not quiescent).

Theorem 14 *In partitionable networks with 5 or more processes, consensus has no quiescent implementation using $\Diamond\mathcal{S}$. This holds even if we assume that no process crashes, there is a link between every pair of processes, each link is eventually up or down,*[14] *a majority of processes are in the same partition, and all processes initially propose a value.*

Proof (Sketch). The proof is by contradiction. Suppose there is a quiescent algorithm $\mathcal{A}$ that uses $\Diamond\mathcal{S}$ to solve consensus for partitionable networks. We consider a network with $n \geq 5$ processes, and construct three runs of $\mathcal{A}$ using $\Diamond\mathcal{S}$ in this network, such that the last run violates the specification of consensus. In each of these three runs no process crashes, and every process executes $\mathcal{A}$ by initially proposing 0.

- *Run* R_0. There are two permanent partitions: $\{1, 2\}$ and $\{3, 4, \ldots, n\}$. Within each partition no messages are lost, and all messages sent across the partitions are lost. At all times, each process $p \in \{1, 2\}$ trusts only itself and process 2, and each process $p \in \{3, 4, \ldots, n\}$ trusts only itself and process 3. We can easily show that processes 1 and 2 cannot decide any value in this run.[15] Since $\mathcal{A}$ is quiescent, there is a time t_0 after which no messages are sent or received in R_0.

- *Run* R_1. Up to time t_0, R_1 is identical to run R_0. At time $t_0 + 1$, the network partitions permanently into $\{1\}$ and $\{2, 3, \ldots, n\}$. From this time on, within each partition no messages are lost, and all messages sent across partitions are lost. Moreover, from time $t_0 + 1$, process 1 trusts only itself, and each process $p \in \{2, 3, \ldots, n\}$ trusts only itself and process 2. Since $\mathcal{A}$ is quiescent, there is a time t_1 after which no messages are sent or received in R_1.

[14] I.e., for each link there is a time after which either all the messages sent are received or no message sent is received.

[15] In a minority partition that does not receive messages from the outside, such as partition $\{1, 2\}$ above, processes can never decide. Otherwise, we construct another run in which, after they decide, the minority partition merges with a majority partition where processes have decided differently.

- *Run* R_2. There is a single partition: $\{1, 2, \ldots, n\}$. Throughout the whole run, process 1 and its failure detector module behaves as in R_0, and all other processes and their failure detector modules behave as in R_1. In particular, up to time t_0, R_2 is identical to R_0, and from time $t_0 + 1$ to t_1, all messages sent to and from process 1 are lost. We conclude that, as in R_0, process 1 does not decide in R_2. This violates the Termination property of consensus, since all processes in partition $\{1, 2, \ldots, n\}$ propose a value.

Note that the behavior of the failure detector in each of the above three runs is compatible with $\diamond\mathcal{S}$. □

5.4 Quiescent Consensus for Partitionable Networks using $\diamond\mathcal{S}_{LP}$ and $\mathcal{HB}$

To solve consensus using $\diamond\mathcal{S}_{LP}$ and $\mathcal{HB}$ in partitionable networks, we take the rotating coordinator consensus algorithm of [CT96], we replace its communication primitives with the corresponding ones defined in Sections 4.3 and 4.1, namely, qr-send, qr-receive, broadcast and deliver, and then we plug in the quiescent implementations of these primitives given in Section 4.2 (these implementations use $\mathcal{HB}$). The resulting algorithm satisfies all the properties of consensus for partitionable networks, including Uniform Agreement and Partition Termination, under the assumption that the largest partition contains a majority of processes (this assumption is only necessary for the Termination property of consensus).[16] Moreover, this algorithm is quiescent.

Although this algorithm is almost identical to the one given in [CT96] for non-partitionable networks, the network assumptions, the consensus requirements, and the failure detector properties are different, and so its proof of correctness and quiescence changes.

The rotating coordinator algorithm is shown in Fig. 3 (the code consisting of lines 39–41 is executed atomically). Processes proceed in asynchronous "rounds". During round r, the coordinator is process $c = (r \mod n) + 1$. Each round is divided into four asynchronous phases. In Phase 1, every process qr-sends its current estimate of the decision value timestamped with the round number in which it adopted this estimate, to the current coordinator c. In Phase 2, c waits to qr-receive $\lceil (n + 1)/2 \rceil$ such estimates, selects one with the largest timestamp, and qr-sends it to all the processes as its new estimate $estimate_c$. In Phase 3, for each process p there are two possibilities: (1) p qr-receives $estimate_c$ from c, it adopts $estimate_c$ as its own estimate, and then qr-sends an *ack* to c; or (2) upon consulting its failure detector module, p suspects c, and qr-sends a *nack* to c. In Phase 4, c waits to qr-receive $\lceil (n+1)/2 \rceil$ replies (*ack* or *nack*). If all replies are *ack*s, then c knows that a majority of processes changed their estimates to $estimate_c$, and thus $estimate_c$ is locked (i.e., no other decision value is possible). Consequently, c reliably broadcasts a request to decide $estimate_c$. At any time, if a process delivers such a request, it decides accordingly.

We next prove that the algorithm is correct and quiescent. Our proof is similar to the one in [CT96], except for the proofs of Termination and Quiescence. The main difficulty in these proofs stems from the fact that we do not assume that partitions are eventually isolated: it is possible for processes in one partition to receive messages from outside this partition, forever. The following is an example of why this is problematic. The failure detector $\diamond\mathcal{S}_{LP}$ guarantees that in the largest partition there is some process c that is trusted by all processes *in that partition*. However, c may be permanently suspected of being faulty by processes outside the largest partition. Thus, it is conceivable that c receives *nacks* from these processes in Phase 4 of *every* round in which it acts as the coordinator. These *nacks* would prevent c from ever broadcasting a request to decide. In such a scenario, processes in the largest partition never decide, and they qr-send messages

[16] A standard partitioning argument shows that consensus for partitionable networks cannot be solved using $\diamond\mathcal{S}$ and $\mathcal{HB}$ if we do not make this assumption.

For every process p:

To execute $\textsf{propose}(v_p)$:

$estimate_p \leftarrow v_p$ $\{estimate_p$ is p's estimate of the decision value$\}$

$state_p \leftarrow undecided$

$r_p \leftarrow 0$ $\{r_p$ is p's current round number$\}$

$ts_p \leftarrow 0$ $\{ts_p$ is the last round in which p updated $estimate_p$, initially $0\}$

repeat $\{$Rotate through coordinators until decision is reached$\}$

$r_p \leftarrow r_p + 1$

$c_p \leftarrow (r_p \bmod n) + 1$ $\{c_p$ is the current coordinator$\}$

Phase 1:

qr-send $(p, r_p, estimate_p, ts_p)$ to c_p

Phase 2:

if $p = c_p$ **then**

wait until [for $\lceil (n+1)/2 \rceil$ processes q: qr-received $(q, r_p, estimate_q, ts_q)$ from q]

$msgs_p[r_p] \leftarrow \{(q, r_p, estimate_q, ts_q) \mid p$ qr-received $(q, r_p, estimate_q, ts_q)$ from $q\}$

$t \leftarrow$ largest ts_q such that $(q, r_p, estimate_q, ts_q) \in msgs_p[r_p]$

$estimate_p \leftarrow$ select one $estimate_q$ such that $(q, r_p, estimate_q, t) \in msgs_p[r_p]$

qr-send $(p, r_p, estimate_p)$ to all

Phase 3:

wait until [qr-received $(c_p, r_p, estimate_{c_p})$ from c_p **or** $\mathcal{D}_p$ suspects c_p] $\{$query $\diamond\mathcal{S}_{LP}\}$

if [qr-received $(c_p, r_p, estimate_{c_p})$ from c_p] **then**

$estimate_p \leftarrow estimate_{c_p}$

$ts_p \leftarrow r_p$

qr-send (p, r_p, ack) to c_p

else qr-send $(p, r_p, nack)$ to c_p

Phase 4:

if $p = c_p$ **then**

wait until [for $\lceil (n+1)/2 \rceil$ processes q: qr-received (q, r_p, ack) **or** $(q, r_p, nack)$]

if [for $\lceil (n+1)/2 \rceil$ processes q: qr-received (q, r_p, ack)] **then**

broadcast $(p, r_p, estimate_p, decide)$ $\{$reliable broadcast the decision value$\}$

until $state_p = decided$

upon $\textsf{deliver}(q, r_q, estimate_q, decide)$

if $state_p = undecided$ **then**

$\textsf{decide}(estimate_q)$

$state_p \leftarrow decided$

Figure 3: Consensus for partitionable networks using $\diamond\mathcal{S}_{LP}$ and reliable communication primitives

forever. Similar scenarios in which processes in the minority partitions qr-send messages forever are also conceivable. To show that all such undesirable scenarios cannot occur, we use a partial order on the set of partitions.

Lemma 15 (Uniform Integrity) *Every process decides at most once.*

Proof. Immediate from the algorithm. □

Lemma 16 (Uniform Validity) *A process can only decide a value that was previously proposed by some process.*

Proof. Immediate from the algorithm, the Uniform Integrity property of qr-send and qr-receive and the Uniform Integrity property of reliable broadcast. □

Lemma 17 (Partition Termination) *If a process decides then every process in its partition decides.*

Proof. If p is faulty then $partition(p) = \emptyset$, so the result is vacuously true. If p is correct then the result follows from the Agreement property of reliable broadcast. □

We omit the proof of the next lemma because it is almost identical to the one of Lemma 6.2.1 in [CT96].

Lemma 18 (Uniform Agreement) *No two processes (whether in the same partition or not) decide differently.*

We now show the termination and quiescence properties of the implementation. For any partition P, we say that $QuiescentDecision(P)$ holds if:

1. all processes in P eventually stop qr-sending messages, and
2. if $|P| > \lfloor n/2 \rfloor$ and all processes in P propose a value, then all processes in P eventually decide.

Lemma 19 *For every partition P, if there is a time after which no process in P* qr-receives *messages from processes in $\Pi \setminus P$, then QuiescentDecision(P) holds.*

Proof (Sketch). Let t be the time after which no process in P qr-receives messages from processes in $\Pi \setminus P$. We first show that all processes in P eventually stop qr-sending messages. There are several possible cases.

Case 1: Some process in P decides. Then by Lemma 17 all processes in P decide. A process that decides stops qr-sending messages after it reaches the end of its current round, so all processes in P eventually stop qr-sending messages.

Case 2: No process in P decides. There are now two subcases:

Case 2.1: Each process in P that proposes a value blocks at a **wait** *statement.* Then all processes in P eventually stop qr-sending messages.

Case 2.2: Some process p in P that proposes a value does not block at any of the **wait** *statements*. Then, since p does not decide, it starts every round $r > 0$. There are now two subcases:

Case 2.2.1: $|P| \le \lfloor n/2 \rfloor$. Let r_0 be the round of process p at time t and let r_1 be the first round after r_0 in which p is the coordinator. In Phase 2 of round r_1, p waits to qr-receive estimates from $\lceil (n+1)/2 \rceil$ processes. It can only qr-receive messages from processes in P, and since $|P| \le \lfloor n/2 \rfloor$, it blocks at the **wait** statement of Phase 2 — a contradiction.

Case 2.2.2: $|P| > \lfloor n/2 \rfloor$. By the Eventual Weak Accuracy property of $\diamond\mathcal{S}_{LP}$, there exists a process $c \in P$ and a time t' such that after t', all processes in P trust c. Let $t'' = \max\{t, t'\}$ and let r_0 be the largest round number among all processes at time t''. Let r_1 and r_2 be, respectively, the first and second rounds greater than r_0 in which c is the designated coordinator. Since p trusts c after time t'', and it completes Phase 3 of round r_2, p must have qr-received a message of the form $(c, r_2, estimate_c)$ from c in that phase. Therefore, c starts round r_2, and thus c completes round r_1. So c qr-receives messages from $\lceil (n+1)/2 \rceil$ processes in Phase 4 of round r_1. These processes are all in P because, after time t'', c qr-receives no messages from processes in $\Pi \setminus P$. All such messages are ack's because all processes in P start round r_1 after time t'', and so they trust c while in round r_1. Therefore, c reliably broadcasts a decision value at the end of Phase 4 of round r_1, and so it delivers that value and decides — a contradiction to the assumption that no process in P decides.

We now show that if $|P| > \lfloor n/2 \rfloor$ and all processes in P propose a value, then all processes in P eventually decide. By Lemma 17, we only need to show that some process in P decides. For contradiction, suppose that no process in P decides. We claim that no process in P remains blocked forever at one of the **wait** statements. This claim implies that every process in P starts every round $r > 0$, and thus qr-sends an infinite number of messages, which contradicts what we have shown above. We prove the claim by contradiction. Let r_0 be the smallest round number in which some process in P blocks forever at one of the **wait** statements. Since all processes in P propose and do not decide, they all reach the end of Phase 1 of round r_0: they all qr-send a message of the type $(*, r_0, estimate, *)$ to the coordinator $c = (r_0 \bmod n) + 1$ of round r_0. Thus, at least $\lceil (n+1)/2 \rceil$ such messages are qr-sent to c. There are now two cases: (1) $c \in P$. Then c qr-receives those messages and replies by qr-sending $(c, r_0, estimate_c)$. Thus c completes Phase 2 of round r_0. Moreover, every process in P qr-receives this message, and so every process in P completes Phase 3 of round r_0. Thus every process in P qr-sends a message of the type $(*, r_0, ack)$ or $(*, r_0, nack)$ to c, and so c completes Phase 4 of round r_0. We conclude that every process in P completes round r_0 — a contradiction. (2) $c \notin P$. Then, by the Strong Completeness property of $\diamond\mathcal{S}_{LP}$, all processes in P eventually suspect c forever, and thus they do not block at the **wait** statement in Phase 3 of round r_0. Therefore, all processes in P complete round r_0 — a contradiction. □

Lemma 20 *For every partition P, QuiescentDecision(P) holds.*

Proof (Sketch). Define a binary relation $\rightsquigarrow$ on the set $Partitions$ as follows: for every $P, Q \in Partitions$, $P \rightsquigarrow Q$ if and only if $P \neq Q$ and there is a fair path from some process in P to some process in Q. Clearly $\rightsquigarrow$ is an irreflexive partial order. The lemma is shown by induction on $\rightsquigarrow$. Let P be any partition and assume that, for every Q such that $Q \rightsquigarrow P$, $QuiescentDecision(Q)$ holds. We must show that $QuiescentDecision(P)$ also holds.

Let Q be any partition such that $Q \rightsquigarrow P$. Since $QuiescentDecision(Q)$ holds, every process $q \in Q$ eventually stops qr-sending messages. So, by the Uniform Integrity property of qr-send and qr-receive, there is a time after which no process in P qr-receives messages from processes in Q.

Now let Q be any partition such that $Q \not\rightsquigarrow P$ and $Q \neq P$. For all processes $q \in Q$ and $p \in P$, there is no fair path from q to p, and so p is not reachable from q. By the Partition Integrity property of qr-send and qr-receive, eventually p does not qr-receive messages from q. So, eventually no process in P qr-receives messages from processes in Q.

We conclude that eventually no process in P qr-receives messages from processes in any partition $Q \neq P$. Moreover, eventually no process in P qr-receives messages from faulty processes. Thus, there is a time after which no process in P qr-receives messages from processes in $\Pi \setminus P$. Therefore, by Lemma 19, $QuiescentDecision(P)$ holds. □

Corollary 21 (Termination) *Assume that the largest partition contains a majority of processes. If all processes in the largest partition propose a value, then they all eventually decide.*

Proof. Let P be the largest partition. By assumption, $|P| > \lfloor n/2 \rfloor$. Apply Lemma 20. □

Corollary 22 (Quiescence) *By plugging the quiescent implementations of* qr-send*,* qr-receive*,* broadcast*, and* deliver *of Section 4.2 into the algorithm of Fig. 3, we obtain a quiescent algorithm.*

Proof. First note that every process p invokes only a finite number of broadcasts: if p crashes, this is obvious; if p is correct and broadcasts at least once, it eventually delivers its first broadcast, and then stops broadcasting soon after this delivery. Furthermore, each process also invokes only a finite number of qr-sends: for a process that crashes, this is obvious, and for a correct process, this is a consequence of Lemma 20. The result now follows since the implementations of broadcast and qr-send in Section 4.2 are quiescent. □

From Lemmata 15, 16, 17 and 18, and Corollaries 21 and 22, we have:

Theorem 23 *Consider the algorithm obtained by plugging the implementations of* qr-send*,* qr-receive*,* broadcast *and* deliver *of Section 4.2 into the algorithm of Fig. 3. This algorithm is quiescent, and satisfies the following properties of consensus: Uniform Agreement, Uniform Validity, Uniform Integrity, and Partition Termination. Moreover, if the largest partition contains a majority of processes, then it also satisfies Termination.*

6 Implementation of $\mathcal{HB}$ for Partitionable Networks

We now show how to implement $\mathcal{HB}$ for partitionable networks. Our implementation (Fig. 4) is a minor modification of the one given in [ACT97a] for non-partitionable networks. Every process p executes two concurrent tasks. In the first task, p periodically increments its own heartbeat value, and sends the message $(\mathsf{HEARTBEAT}, p)$ to all its neighbors. The second task handles the receipt of messages of the form $(\mathsf{HEARTBEAT}, path)$. Upon the receipt of such a message from process q, p increases the heartbeat values of all the processes that appear after p in $path$. Then p appends itself to $path$ and forwards message $(\mathsf{HEARTBEAT}, path)$ to all its neighbors that appear at most once in *path*.

Note that $\mathcal{HB}$ does *not* use timeouts on the heartbeats of a process in order to determine whether this process has failed or not. $\mathcal{HB}$ just counts the *total number of heartbeats* received from each process, and outputs these "raw" counters without any further processing or interpretation.

Thus, $\mathcal{HB}$ should not be confused with existing implementations of failure detectors (some of which, such as those in Ensemble and Phoenix, have modules that are also called *heartbeat* [vR97, Cha97]). Even though existing failure detectors are also based on the repeated sending of a heartbeat, they use timeouts on

```
1   For every process p:
2
3       Initialization:
4           for all q ∈ Π do D_p[q] ← 0                                   {D_p is the output of HB at p}
5
6       cobegin
7           || Task 1:
8               repeat periodically
9                   D_p[p] ← D_p[p] + 1                                   {increment p's own heartbeat}
10                  for all q ∈ neighbor(p) do send (HEARTBEAT, p) to q
11
12          || Task 2:
13              upon receive (HEARTBEAT, path) from q do
14                  for all q ∈ Π such that q appears after p in path do
15                      D_p[q] ← D_p[q] + 1
16                  path ← path · p
17                  for all q such that q ∈ neighbor(p) and q appears at most once in path do
18                      send (HEARTBEAT, path) to q
19      coend
```

Figure 4: Implementation of $\mathcal{HB}$ for partitionable networks

heartbeats in order to derive lists of processes considered to be up or down; applications can only see these lists. In contrast, $\mathcal{HB}$ simply counts heartbeats, and shows these counts to applications.

We now proceed to prove the correctness of the implementation.

Lemma 24 *At each process p, the heartbeat sequence of every process q is nondecreasing.*

Proof. This is clear since $\mathcal{D}_p[q]$ can only be changed in lines 9 and 15. $\square$

Lemma 25 *At each correct process p, the heartbeat sequence of every process in the partition of p is unbounded.*

Proof. Let q be a process in the partition of p. If $q = p$ then line 9 is executed infinitely many times (since p is correct), and so the heartbeat sequence of p at p is unbounded. Now assume $q \neq p$ and let $(p_1, p_2, \dots, p_i)$ be a simple fair path from p to q, and $(p_i, p_{i+1}, \dots, p_k)$ be a simple fair path from q to p, so that $p_1 = p_k = p$ and $p_i = q$. For $j = 1, \dots, k$, let $P_j = (p_1, \dots, p_j)$. For each $j = 1, \dots, k-1$, we claim that p_j sends $(\mathsf{HEARTBEAT}, P_j)$ to p_{j+1} an infinite number of times. We show this by induction on j. For the base case ($j = 1$), note that $p_1 = p$ is correct, so its Task 1 executes forever and therefore p_1 sends $(\mathsf{HEARTBEAT}, p_1)$ to all its neighbors, and thus to p_2, an infinite number of times. For the induction step, let $j < k-1$ and assume that p_j sends $(\mathsf{HEARTBEAT}, P_j)$ to p_{j+1} an infinite number of times. Since p_{j+1} is correct and the link $p_j \rightarrow p_{j+1}$ is fair, p_{j+1} receives $(\mathsf{HEARTBEAT}, P_j)$ an infinite number of times. Moreover, p_{j+2} appears at most once in P_{j+1} and p_{j+2} is a neighbor of p_{j+1}, so each time p_{j+1} receives $(\mathsf{HEARTBEAT}, P_j)$, it sends $(\mathsf{HEARTBEAT}, P_{j+1})$ to p_{j+2} in line 18. Therefore, p_{j+1} sends $(\mathsf{HEARTBEAT}, P_{j+1})$ to p_{j+2} an infinite number of times. This shows the claim.

For $j = k-1$ this claim shows that p_{k-1} sends $(\mathsf{HEARTBEAT}, P_{k-1})$ to p_k an infinite number of times. Process p_k is correct and link $p_{k-1} \rightarrow p_k$ is fair, so p_k receives $(\mathsf{HEARTBEAT}, P_{k-1})$ an infinite number of times. Note that q appears after p in P_{k-1}. So every time p_k receives $(\mathsf{HEARTBEAT}, P_{k-1})$, it increments $\mathcal{D}_{p_k}[q]$ in line 15. So $\mathcal{D}_{p_k}[q]$ is incremented an infinite number of times. Note that, by Lemma 24, $\mathcal{D}_{p_k}[q]$ can never be decremented. So, the heartbeat sequence of q at $p_k = p$ is unbounded. □

Corollary 26 ($\mathcal{HB}$-Accuracy) *At each process p, the heartbeat sequence of every process is nondecreasing, and at each correct process p, the heartbeat sequence of every process in the partition of p is unbounded.*

Proof. From Lemmata 24 and 25. □

Lemma 27 *If some process p sends* $(\mathsf{HEARTBEAT}, path)$ *then (1) p is the last process in path and (2) no process appears more than twice in path.*

Proof. Obvious. □

Lemma 28 *Let p and q be processes, and path be a sequence of processes. Suppose that p receives message* $(\mathsf{HEARTBEAT}, path \cdot q)$ *an infinite number of times. Then q is correct and link* $q \rightarrow p$ *is fair. Moreover, if path is non-empty, then q receives message* $(\mathsf{HEARTBEAT}, path)$ *an infinite number of times.*

Proof. Obvious. □

Lemma 29 ($\mathcal{HB}$-Completeness) *At each correct process p, the heartbeat sequence of every process not in the partition of p is bounded.*

Proof (Sketch). Let q be a process that is not in the partition of p. Note that $q \neq p$. For a contradiction, suppose that the heartbeat sequence of q at p is not bounded. Then p increments $\mathcal{D}_p[q]$ an infinite number of times in line 15. So, for an infinite number of times, p receives messages of the form $(\mathsf{HEARTBEAT}, *)$ with a second component that contains q after p. Lemma 27 part (2) implies that the second component of a message of the form $(\mathsf{HEARTBEAT}, *)$ ranges over a finite set of values. Thus there exists a $path$ containing q after p such that p receives $(\mathsf{HEARTBEAT}, path)$ an infinite number of times. Let $path = (p_1, \ldots, p_k)$. For convenience, let $p = p_{k+1}$. By repeated applications of Lemma 28, we conclude that for each $j = k, k-1, \ldots, 1$, p_j is correct and link $p_j \rightarrow p_{j+1}$ is fair. Let $i, i' \in \{1, \ldots, k\}$ be such that $p_i = p$, $p_{i'} = q$ and $i < i'$. Thus $(p_i, p_{i+1}, \ldots, p_{i'})$ is a fair path from p to q and $(p_{i'}, p_{i'+1}, \ldots, p_k, p)$ is a fair path from q to p. Therefore p and q are in the same partition — a contradiction. □

By Corollary 26 and the above lemma, we have:

Theorem 30 *Figure 4 implements* $\mathcal{HB}$ *for partitionable networks.*

7 Some Practical Considerations

In contrast to several previous works on network partitions, we did not assume here that all partitions are isolated. In other words, there can be two partitions P and P' such that processes in P can continuously receive messages from processes in P' (but processes in P' eventually stop receiving messages from P). Dealing with non-isolated partitions complicates the task of designing and/or proving the algorithms (e.g., in the proof of our Consensus algorithm, we had to define a partial order on the set of partitions, and argue by induction on this partial order). The completeness properties of $\mathcal{HB}$ and $\diamond\mathcal{S}$ helped us deal with non-isolated partitions, as we now explain.

Let P and P' be two partitions such that $p \in P$ receives every message that $p' \in P'$ sends. The completeness property of $\mathcal{HB}$ requires that the heartbeat of p' at p must eventually stop. Similarly, the completeness property of $\diamond\mathcal{S}$ requires that p permanently suspectes p'. In other words, even though p receives all the messages of p', $\mathcal{HB}$ and $\diamond\mathcal{S}$ must behave as if all the processes in P were actually isolated from those in P'. Thus, $\mathcal{HB}$ and $\diamond\mathcal{S}$ help algorithms by "restoring" the isolation of partitions to some extent. At this point, it may seem that we dealt with problem of non-isolated partitions by simply "postulating it away" in the definitions of $\mathcal{HB}$ and $\diamond\mathcal{S}$. This is not the case, since we gave an implementation of $\mathcal{HB}$ (Section 6), and by incorporating a timeout mechanism to this implementation, one can also obtain $\diamond\mathcal{S}$: if the heartbeat of p' at p does not increase within a certain timeout period, p suspects p' (of course, timeout mechanisms make sense only in partially synchronous systems).

We now address the issue of message buffering. Soon after a process p crashes its heartbeat ceases everywhere and processes stop sending messages to p. However, they do have to keep the messages they intended to send to p, just in case p is merely very slow, and the heartbeat of p resumes later on. In theory, they have to keep these messages forever, and this requires unbounded buffers. In practice, however, the system will eventually decide that p is indeed useless and will "remove" p (e.g. via a Group Membership protocol). All the stored messages addressed to p can then be discarded. The removal of p may take a long time,[17] but the heartbeat mechanism ensures that processes stop sending messages to p soon after p actually crashes, and much before its removal. The same considerations apply if, instead of crashing, p is partitioned away from its current partition P, and the (Partitionable) Group Membership eventually removes it from P.

8 Related Work

Regarding reliable communication, the works that are closest to ours are [BCBT96, ACT97a]. Both of these works, however, consider only non-partitionable networks. In [BCBT96], Basu *et al.* pose the following question: given a problem that can be solved in asynchronous systems with process crashes only, can this problem still be solved if links can also fail by losing messages? They show that the answer is "yes" if the problem is correct-restricted [BN92, Gop92][18] or if more than half of the processes do not crash. However, the communication algorithms that they give are not quiescent (and do not use failure detectors). [ACT97a] was the first paper to study the problem of achieving quiescent reliable communication by using failure detectors in a system with process crashes and lossy links.

Regarding consensus, the works that are closest to ours are [FKM$^+$95, CHT96a, DFKM96, GS96]. In [GS96], as a first step towards partitionable networks, Guerraoui and Schiper define Γ-*accurate* failure detectors. Roughly speaking, only a subset Γ of the processes are required to satisfy some accuracy property. However, their model assumes that the network is completely connected and links between correct processes do not lose messages — thus, no permanent partition is possible.

The first paper to consider the consensus problem in partitionable networks is [FKM$^+$95], but the algorithms described in that paper had errors [CHT96a]. Correct algorithms can be found in [CHT96a, DFKM96].[19] All these algorithms use a variant of $\diamond\mathcal{S}$, but in contrast to the one given in this paper they do not use $\mathcal{HB}$ and are not quiescent: processes in minority partitions may send messages forever. Moreover, these

[17] In some group membership protocols, the timeout used to remove a process is on the order of minutes: killing a process is expensive and so timeouts are set conservatively.

[18] I.e., its specification refers only to the behavior of non-faulty processes.

[19] Actually, the specification of consensus considered in [FKM$^+$95, CHT96a] only requires that *one* correct process in the largest partition eventually decides. Ensuring that *all* correct processes in the largest partition decide can be subsequently achieved by a (quiescent) reliable broadcast of the decision value.

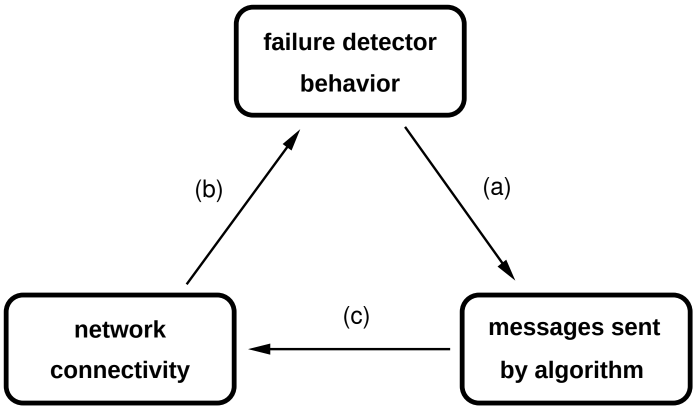

Figure 5: Cycle of dependencies when network connectivity is defined in terms of messages sent

algorithms make the following additional assumptions: (a) the largest partition is eventually isolated from the rest of the system: there is a time after which messages do not go in or out of this partition, and (b) links in the largest partition can lose only a finite number of messages (recall that in our case, all links may lose an infinite number of messages). The underlying model of failures and failure detectors is also significantly different from the one proposed in this paper. Another model of failure detectors for partitionable networks is given in [BDM97]. We compare models in the next section.

9 Comparison with other Models

In [DFKM96, BDM97], network connectivity is defined in terms of the messages exchanged in a run — in particular, it depends on whether the algorithm being executed sends a message or not, on the times these messages are sent, and on whether these messages are received. This way of defining network connectivity, which is fundamentally different from ours, has two drawbacks. First, it creates the following cycle of dependencies (Fig. 5): (a) The messages that an algorithm sends in a particular run depend on the algorithm itself and on the behavior of the failure detector it is using, (b) the behavior of the failure detector depends on the network connectivity, and (c) the network connectivity depends on the messages that the algorithm sends. Second, it raises the following issue: are the messages defining network connectivity, those of the applications, those of the failure detection mechanism, or both?

In our model, network connectivity does not depend on messages sent by the algorithm, and so we avoid the above drawbacks. In fact, network connectivity is determined by the (process and link) failure pattern which is defined independently of the messages sent by the algorithm. The link failure pattern is intended to model the physical condition of each link independent of the particular messages sent by the algorithm being executed.

In [DFKM96], two processes p and q are *permanently connected* in a given run if they do not crash and there is a time after which every message that p sends to q is received by q, and vice-versa. Clearly, network connectivity depends on the messages of the run.

In [BDM97], process q *is partitioned from* p *at time* t if the last message that p sent to q by time $t' \leq t$ is never received by q. This particular way of defining network connectivity in terms of messages is problematic for our purposes, as the following example shows.

A process p wishes to send a sequence of messages to q. For efficiency, the algorithm of p sends a message to q only when p's failure detector module indicates that q is currently reachable from p (this is not unreasonable: it is the core idea behind the use of failure detector $\mathcal{HB}$ to achieve quiescent reliable communication). Suppose that at time t, p sends m to q, and this message is lost (it is never received by q). By the definition in [BDM97], q is partitioned from p at time t. Suppose that the failure detector module at p now tells p (correctly) that q is partitioned from p. At this point, p stops sending messages to q until the failure detector says that q has become reachable again. However, since p stopped sending messages to q, by definition, q remains partitioned from p forever, and the failure detector oracle (correctly) continues to report that q is unreachable from p, forever. Thus, the loss of a single message discourages p from ever sending messages to q again.

A possible objection to the above example is that the failure detector module at p is not just an oracle with axiomatic properties, but also a process that sends its own messages to determine whether q is reachable or not. Furthermore, these failure detector messages should also be taken into account in the definition of network connectivity (together with the messages exchanged by the algorithms that use those failure detectors). However, this defeats one of the original purpose of introducing failure detection as a clean *abstraction* to reason about fault tolerance. The proof of correctness of an algorithm (such as the one in the simple example above) should refer only to the *abstract properties* of the failure detector that it uses, and not to any aspects of its *implementation*.

Acknowledgments

We would like to thank Anindya Basu, Tushar Deepak Chandra, Francis Chu, Vassos Hadzilacos, and the anonymous referees for their helpful comments.

References

[ACT97a] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Heartbeat: a timeout-free failure detector for quiescent reliable communication. In Marios Mavronicolas and Philippas Tsigas, editors, *Proceedings of the 11th International Workshop on Distributed Algorithms*, Lecture Notes on Computer Science. Springer-Verlag, September 1997. A full version is also available as Technical Report 97-1631, Computer Science Department, Cornell University, Ithaca, New York, May 1997.

[ACT97b] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. On the weakest failure detector for quiescent reliable communication. Technical Report 97-1640, Department of Computer Science, Cornell University, July 1997.

[BCBT96] Anindya Basu, Bernadette Charron-Bost, and Sam Toueg. Simulating reliable links with unreliable links in the presence of process crashes. In Özalp Babaoğlu and Keith Marzullo, editors, *Proceedings of the 10th International Workshop on Distributed Algorithms*, Lecture Notes on Computer Science, pages 105–122. Springer-Verlag, October 1996.

[BDM97] Özalp Babaoğlu, Renzo Davoli, and Alberto Montresor. Partitionable group membership: specification and algorithms. Technical Report UBLCS-97-1, Dept. of Computer Science, University of Bologna, Bologna, Italy, January 1997.

[BN92] Rida Bazzi and Gil Neiger. Simulating crash failures with many faulty processors. In Adrian Segal and Shmuel Zaks, editors, *Proceedings of the 6th International Workshop on Distributed Algorithms*, Lecture Notes on Computer Science, pages 166–184. Springer-Verlag, 1992.

[Cha97] Tushar Deepak Chandra, April 1997. Private Communication.

[CHT96a] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg, March 1996. Private Communication to the authors of [FKM$^+$95].

[CHT96b] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[DFKM96] Danny Dolev, Roy Friedman, Idit Keidar, and Dahlia Malkhi. Failure detectors in omission failure environments. Technical Report TR96-1608, Department of Computer Science, Cornell University, Ithaca, New York, September 1996.

[FKM$^+$95] Roy Friedman, Idit Keidar, Dahlia Malkhi, Ken Birman, and Danny Dolev. Deciding in partitionable networks. Technical Report TR95-1554, Department of Computer Science, Cornell University, Ithaca, New York, November 1995.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[Gop92] Ajei Gopal. *Fault-Tolerant Broadcasts and Multicasts: The Problem of Inconsistency and Contamination*. PhD thesis, Cornell University, January 1992.

[GS96] Rachid Guerraoui and André Schiper. Gamma-Accurate failure detectors. In Özalp Babaoğlu and Keith Marzullo, editors, *Proceedings of the 10th International Workshop on Distributed Algorithms*, Lecture Notes on Computer Science, pages 269–286. Springer-Verlag, October 1996.

[HT94] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR 94-1425, Department of Computer Science, Cornell University, Ithaca, New York, May 1994.

[vR97] Robbert van Renesse, April 1997. Private Communication.