On Quiescent Reliable Communication*

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

December 3, 1998

Abstract

We study the problem of achieving reliable communication with *quiescent* algorithms (i.e., algorithms that eventually stop sending messages) in asynchronous systems with process crashes and lossy links. We first show that it is impossible to solve this problem in purely asynchronous systems (with no failure detectors). We then show that, among failure detectors that output lists of suspects, the weakest one that can be used to solve this problem is $\diamond\mathcal{P}$, a failure detector that cannot be implemented. To overcome this difficulty, we introduce an implementable failure detector called *Heartbeat* and show that it can be used to achieve quiescent reliable communication. Heartbeat is novel: in contrast to typical failure detectors, it does not output lists of suspects and it is implementable without timeouts. With Heartbeat, many existing algorithms that tolerate only process crashes can be transformed into quiescent algorithms that tolerate both process crashes and message losses. This can be applied to consensus, atomic broadcast, k-set agreement, atomic commitment, etc.

1 Introduction

1.1 Motivation

We focus on the problem of *quiescent* reliable communication in asynchronous message-passing systems with process crashes and lossy links. To illustrate this problem consider a system of two processes, a sender s and a receiver r, connected by an asynchronous bidirectional link. Process s wishes to send some message m to r. Suppose first that no process may crash, but the link between s and r may lose messages (in both directions). If we put no restrictions on message losses it is obviously impossible to ensure that r receives m. An assumption commonly made to circumvent this problem is that the link is *fair*: if a message is sent infinitely often then it is received infinitely often.

With such a link, s could repeatedly send copies of m forever, and r is guaranteed to eventually receive m. This is impractical, since s never stops sending messages. The obvious fix is the following protocol: (a) s sends a copy of m repeatedly until it receives $ack(m)$ from r, and (b) upon each receipt of m, r sends $ack(m)$ back to s. Note that this protocol is *quiescent*: eventually no process sends or receives messages.

*Research partially supported by NSF grant CCR-9402896 and CCR-9711403, by ARPA/ONR grant N00014-96-1-1014, and by an Olin Fellowship. An extended abstract of part of this paper appeared in the *Proceedings of the 11th International Workshop on Distributed Algorithms*, September 1997.

The situation changes if, in addition to message losses, process crashes may also occur. The protocol above still works, but it is not quiescent anymore: for example, if r crashes before sending $ack(m)$, then s will send copies of m forever. Is there a *quiescent* protocol ensuring that if neither s nor r crashes then r eventually receives m? It turns out that the answer is no, even if one assumes that the link can only lose a finite number of messages.

Since process crashes and message losses are common types of failures, this negative result is an obstacle to the design of fault-tolerant distributed systems. In this paper, we explore the use of *unreliable failure detectors* to circumvent this obstacle. Roughly speaking, unreliable failure detectors provide (possibly erroneous) hints on the operational status of processes. Each process can query a local failure detector module that provides some information about which processes have crashed. This information is typically given in the form of a list of *suspects*.

In general, failure detectors can make mistakes: a process that has crashed is not necessarily suspected and a process may be suspected even though it has not crashed. Moreover, the local lists of suspects dynamically change and lists of different processes do not have to agree (or even eventually agree). Introduced in [CT96], the abstraction of unreliable failure detectors has been used to solve several important problems such as consensus, atomic broadcast, group membership, non-blocking atomic commitment, and leader election [BDM97, Gue95, HMR97, LH94, OGS97, SM95].

Our goal is to use unreliable failure detectors to achieve quiescence, but before we do so we must address the following important question. Note that 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 failure detection service as a tool to achieve this goal.

1.2 Achieving Quiescent Reliable Communication Using Failure Detectors

How can we use an unreliable failure detector to achieve quiescent reliable communication in the presence of process and link failures? This can be done with the *eventually perfect* failure detector $\diamond\mathcal{P}$ [CT96]. Intuitively, $\diamond\mathcal{P}$ satisfies the following two properties: (a) if a process crashes then there is a time after which it is permanently suspected, and (b) if a process does not crash then there is a time after which it is never suspected. Using $\diamond\mathcal{P}$, the following obvious algorithm solves our sender/receiver example: (a) while s has not received $ack(m)$ from r, it periodically does the following: s queries $\diamond\mathcal{P}$ and sends a copy of m to r if r is not currently suspected; (b) upon each receipt of m, r sends $ack(m)$ back to s. Note that this algorithm is *quiescent*: eventually no process sends or receives messages.

So $\diamond\mathcal{P}$ is sufficient to achieve quiescent reliable communication. But is it necessary? In the first part of the paper, we show that among all failure detectors that output lists of suspects, $\diamond\mathcal{P}$ is indeed the *weakest* one that can be used to solve this problem. Unfortunately, $\diamond\mathcal{P}$ is not implementable (this would violate a known impossibility result [FLP85, CT96]). Thus, at a first glance, it seems that achieving quiescent reliable communication requires a failure detector that cannot be implemented. In the second part of the paper, we show that this is not so.

In fact, we show that quiescent reliable communication can be achieved with a failure detector that *is implementable* in systems with process crashes and lossy links. This new failure detector, called *heartbeat* and denoted $\mathcal{HB}$, is very simple. Roughly speaking, the failure detector module of $\mathcal{HB}$ at a process p outputs a vector of counters, one for each neighbor q of p. If neighbor q does not crash, its counter at p increases with no bound. If q crashes, its counter eventually stops increasing. The basic idea behind an implementation of $\mathcal{HB}$ is the obvious one: each process periodically sends an *I-am-alive* message (a "heartbeat") and every process receiving a heartbeat increases the corresponding counter.[1]

$\mathcal{HB}$ should not be confused with existing failure detectors (some of which, such as those in Ensemble and Phoenix, have modules that are also called *heartbeat* [vR97, Cha97]): $\mathcal{HB}$ does not output lists of suspects and its implementation does not use any timeout mechanism. Even though existing failure detectors are also based on the repeated sending of a heartbeat, *they use timeouts* on these heartbeats in order to output *lists of processes* considered to be up or down; applications can only see these lists. In contrast, $\mathcal{HB}$ simply counts the total number of heartbeats received from each process, and shows these "raw" counters to applications without any further processing or interpretation.

A remark is now in order regarding the practicality of $\mathcal{HB}$. As we mentioned above, $\mathcal{HB}$ outputs a vector of unbounded counters. In practice, these unbounded counters are not a problem for the following reasons. First, they are in *local memory* and not in messages — our $\mathcal{HB}$ implementations use 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.

1.3 Detailed Outline of the Results

We focus on two types of reliable communication mechanisms: *quasi reliable send and receive*, and *reliable broadcast*. Roughly speaking, a pair of send/receive primitives is quasi reliable if it satisfies the following property: if processes s and r are *correct* (i.e., they do not crash), then r receives a message from s exactly as many times as s sent that message to r. Reliable broadcast [HT93] ensures that if a correct process broadcasts a message m then all correct processes deliver m; moreover, all correct processes deliver the same set of messages. Our goal is to obtain quiescent implementations of these primitives in networks that do not partition permanently. More precisely, we consider networks in which processes may crash and links may lose messages but every pair of correct processes are connected through some *fair path*, i.e., a path containing only fair links and correct processes.

We first show that, in purely asynchronous systems (with no failure detectors), there is no quiescent implementation of quasi reliable send/receive or of reliable broadcast in such networks (even if we assume that links can lose only a finite number of messages). We then show that the weakest failure detector *with bounded output size*[2] that can be used to solve these problems is $\diamond\mathcal{P}$ — which is not implementable.

To overcome this difficulty, we introduce $\mathcal{HB}$, a failure detector that outputs unbounded counters, and show that $\mathcal{HB}$ is strong enough to achieve quiescent reliable communication, but weak enough to be implementable. We consider two types of networks. In the first type, all links are bidirectional and fair. In the second one, some links are unidirectional, and some links have no restrictions on message losses, i.e., they are not fair. Examples of such networks are unidirectional rings that intersect. For the first type of networks, a common one in practice, the implementation of $\mathcal{HB}$ and the reliable communication algorithms are very simple and efficient. The algorithms for the second type are significantly more complex.

We then consider two stronger types of communication primitives, namely, *reliable send and receive*, and *uniform reliable broadcast*, and give quiescent implementations that use $\mathcal{HB}$. These implementations

[1] As we will see, however, in some types of networks the actual implementation is not as easy.

[2] Note that a list of suspects has bounded size.

assume that a majority of processes are correct (a result in [BCBT96] shows that this assumption is necessary).

We conclude the paper by showing how $\mathcal{HB}$ can be used to extend previous work in order to solve problems with algorithms that are both quiescent and tolerant of process crashes and messages losses. First, we explain how $\mathcal{HB}$ can be used to transform many existing algorithms that tolerate process crashes into quiescent algorithms that tolerate both process crashes and message losses (fair links). This transformation can be applied to the algorithms for consensus in [AT96, Ben83, BT85, CT96, CMS89, FM90, Rab83], for atomic broadcast in [CT96], for k-set agreement in [Cha93], for atomic commitment in [Gue95], for approximate agreement in [DLP$^+$86], etc. Next, we show that $\mathcal{HB}$ can be used to extend the work in [BCBT96] to obtain the following result. Let P be a problem. Suppose P is correct-restricted (i.e., its specification refers only to the behavior of correct processes) or a majority of processes are correct. If P is solvable with a quiescent protocol that tolerates only process crashes, then P is also solvable with a quiescent protocol that tolerates process crashes and message losses.[3]

To summarize, in this paper:

1. We explore the use of unreliable failure detectors to achieve *quiescent* reliable communication in the presence of process crashes and lossy links — a problem that cannot be solved without failure detection.

2. We show that the weakest failure detector with bounded output size that can be used to solve this problem is $\diamond\mathcal{P}$ — which is not implementable.

3. To overcome this obstacle, we introduce $\mathcal{HB}$: this failure detector can be used to achieve quiescent reliable communication and it *is* implementable. In contrast to common failure detectors [BDM97, CT96, Gue95, GLS95, LH94, SM95], $\mathcal{HB}$ does not output a list of suspects, and it can be implemented without timeouts.

4. We show that $\mathcal{HB}$ can be used to extend existing algorithms for many fundamental problems (e.g., consensus, atomic broadcast, k-set agreement, atomic commitment, approximate agreement) to tolerate message losses. It can also be used to extend the results of [BCBT96].

Result (2) above implies that failure detectors with bounded output size are either (a) too weak to achieve quiescent reliable communication, or (b) not implementable. Thus, failure detectors that output lists of processes, which are commonly used in practice, are not always the best ones to solve a problem: their power or applicability is limited. To the best of our knowledge, this is the first work that shows that failure detectors with bounded output size have inherent limitations.

The problem of achieving reliable communication despite failures has been extensively studied, especially in the context of data link protocols (see Chapter 22 of [Lyn96] for a compendium). Our work differs from previous results because we seek *quiescent* algorithms in systems where processes *and* links can fail (and this requires the use of unreliable failure detectors). The works that are the closest to ours are due to Moses *et al.* [MR89] and Basu *et al.* [BCBT96]. The main goal of [MR89] is to achieve quiescent reliable communication with algorithms that garbage collect old messages in systems with lossy links (the issue of garbage collection is only briefly considered here). The algorithms in [MR89], however, are not resilient to process crashes. The protocols in [BCBT96] tolerate both process crashes and lossy links but they are not quiescent (and they do not use failure detectors). In Section 10, we use $\mathcal{HB}$ to extend the results of [BCBT96] and obtain quiescent protocols.

The paper is organized as follows. Our model is given in Section 2. Section 3 defines the reliable communication primitives that we focus on. In Section 4, we show that, without failure detectors, quiescent

[3]The link failure model in [BCBT96] is slightly different from the one used here (cf. Section 10).

reliable communication is impossible. In Section 5, we prove that $\diamond\mathcal{P}$ is the weakest failure detector with bounded output size that can be used to solve this problem (this proof is under a simplifying assumption; the proof without this assumption is given in the Appendix). We then define the heartbeat failure detector $\mathcal{HB}$ in Section 6. In Section 7, we show how to use $\mathcal{HB}$ to achieve quiescent reliable communication. In Section 8, we show how to implement $\mathcal{HB}$. In Section 9, we consider two stronger types of communication primitives and give quiescent implementations that use $\mathcal{HB}$. In Section 10, we explain how $\mathcal{HB}$ can be used to extend several previous results. We conclude the paper with some remarks about message buffering, quiescence versus termination, models of lossy links, and the generalization of our results to partitionable networks.

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. We do not assume that the network is completely connected or that the links are bidirectional. The system can experience both process failures and link failures. Processes can fail by crashing, and links can fail by dropping messages. The model, based on the one in [CHT96], is described next.

A network is a directed graph $G = (\Pi, \Lambda)$ where $\Pi = \{1, \dots, n\}$ 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 $p \to q$, 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 say *$p$ crashes in F_P* if $p \in F_P(t)$ for some t; otherwise we say *p is correct in F_P*.

Some links in the network are fair. Roughly speaking, a fair link $p \to q$ may intermittently drop messages, and may do so infinitely often, but it must satisfy the following "fairness" property: if p repeatedly sends some message to q and q does not crash, then q eventually receives that message. Link properties are made precise in Section 2.5.

A *link failure pattern* F_L is a subset of the set of links Λ. Intuitively, F_L is the set of links that may fail to satisfy the above fairness property. If $p \to q \notin F_L$, we say that *$p \to q$ is fair in F_L*.

A *failure pattern* $F = (F_P, F_L)$ combines a process failure pattern and a link failure pattern, and $correct\_proc(F)$ and $crashed\_proc(F)$ denote the set of processes that are correct and crashed in F_P, respectively.

2.2 Network Connectivity

The following definitions are with respect to a given failure pattern $F = (F_P, F_L)$. We say that a path $(p_1, \dots, p_k)$ in the network is *fair* if processes $p_1, \dots, p_k$ are correct and links $p_1 \to p_2, \dots, p_{k-1} \to p_k$ are fair. We assume that every pair of distinct correct processes is connected through a fair path. This precludes permanent network partitions.

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 output of $\mathcal{D}$). $\mathcal{D}(F)$ denotes the set of possible failure detector histories permitted by $\mathcal{D}$ for the failure pattern F.

We now define the *eventually perfect failure detector* $\diamond\mathcal{P}$ [CT96].[4] Each failure detector module of $\diamond\mathcal{P}$ outputs a *set of processes* that are suspected to have crashed, i.e., $\mathcal{R}_{\diamond\mathcal{P}} = 2^\Pi$. For each failure pattern F, $\diamond\mathcal{P}(F)$ is the set of all failure detector histories H with range $\mathcal{R}_{\diamond\mathcal{P}}$ that satisfy the following properties:

- *Strong Completeness*: Eventually every process that crashes is permanently suspected by every correct process. More precisely:

$$\exists t \in \mathcal{T}, \forall p \in \mathit{crashed\_proc}(F), \forall q \in \mathit{correct\_proc}(F), \forall t' \geq t : p \in H(q,t')$$

- *Eventual Strong Accuracy*: There is a time after which correct processes are not suspected by any correct process. More precisely:

$$\exists t \in \mathcal{T}, \forall t' \geq t, \forall p,q \in \mathit{correct\_proc}(F) : p \notin H(q,t')$$

Sometimes we need to consider systems without failure detectors. For convenience, we model such systems by assuming that their failure detectors always output nil. More precisely, the *nil failure detector* $\mathcal{D}_\perp$ is the one where the failure detector modules of all processes always output $\perp$, independent of the failure pattern. A *system without failure detectors* is one whose failure detector is $\mathcal{D}_\perp$.

2.4 Runs of Algorithms

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 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 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., $p \in \mathit{correct\_proc}(F)$, 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.

2.5 Link Properties

Each run $R = (F, H_\mathcal{D}, I, S, T)$ must also satisfy some "link properties". First, no link creates or duplicates messages. More precisely, 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;

[4] In [CT96], $\diamond\mathcal{P}$ denotes a *class* of failure detectors.

Moreover, every fair link transports any message that is repeatedly sent through it. More precisely, for every link $p \to q \notin F_L$:

- *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.

Note that any link, whether fair or not, may lose (or not lose) messages arbitrarily during any finite period of time. In particular, a fair link may lose all the messages sent during any finite period of time, while a link that is not fair may behave perfectly during that time.

2.6 Environments and Problem Solving

The correctness of an algorithm may depend on certain assumptions on the "environment", e.g., the maximum number of processes that may crash. For example, a consensus algorithm may need the assumption that a majority of processes is correct. Formally, an *environment* $\mathcal{E}$ is a set of failure patterns. Unless otherwise stated, the only restriction that we put on the environment in this paper is that every pair of distinct correct processes is connected through a fair path.

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 $\mathcal{E}$.

3 Quiescent Reliable Communication

In this paper, we focus on quasi reliable send and receive, and reliable broadcast, because these communication primitives are sufficient to solve many problems (see Section 10.1). Stronger types of communication primitives — reliable send and receive, and uniform reliable broadcast — are briefly considered in Section 9.

3.1 Quasi Reliable Send and Receive

Consider any two distinct processes s and r. We define *quasi reliable send and receive from s to r* 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:

- *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.

- *Quasi No Loss*[5]: For all $k \geq 1$, if both s and r are correct, 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.

[5] A stronger property, called *No Loss*, is used in Section 9.1 to define *reliable* send and receive.

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

We want to implement quasi reliable send/receive primitives 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 qr-send$_{s,r}$ is invoked a finite number of times.[6]

3.2 Reliable Broadcast

Reliable broadcast [BT85] 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 $sender(m)$, and a sequence number, denoted $seq(m)$. 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[HT93]:

- *Validity*: If a correct process broadcasts a message m, then it eventually delivers m.
- *Agreement*: If a correct process delivers a message m, then all correct processes eventually deliver m.
- *Uniform Integrity*: For every message m, every process delivers m at most once, and only if m was previously broadcast by $sender(m)$.

Validity and Agreement imply that if a correct process broadcasts a message m, then all correct processes eventually deliver m.

We want to implement reliable broadcast 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 broadcast is invoked a finite number of times.

3.3 Relating Reliable Broadcast and Quasi Reliable Send and Receive

From a quiescent implementation of quasi reliable send and receive one can easily obtain a quiescent implementation of reliable broadcast, and vice-versa.

Remark 1 *From any quiescent implementation of reliable broadcast, we can obtain a quiescent implementation of the quasi reliable primitives* qr-send$_{p,q}$ *and* qr-receive$_{q,p}$ *for every pair of processes p and q.*

Remark 2 *Suppose that every pair of correct processes is connected through a path of correct processes. If we have a quiescent implementation of quasi reliable primitives* qr-send$_{p,q}$ *and* qr-receive$_{q,p}$ *for all processes p and $q \in neighbor(p)$, then we can obtain a quiescent implementation of reliable broadcast.*

To implement reliable broadcast from qr-send and qr-receive one can use a simple diffusion algorithm (e.g., see [HT93]).

4 Impossibility of Quiescent Reliable Communication

We now show that in a system without failure detectors, quiescent reliable communication cannot be achieved. This holds even if the network is completely connected and only a finite number of messages can be lost.

[6] A quiescent implementation of qr-send$_{s,r}$ and qr-receive$_{r,s}$ is allowed to send a finite number of messages even if no qr-send$_{s,r}$ is invoked at all (e.g., some messages may be sent as part of an "initialization phase").

Theorem 1 *Consider a system without failure detectors where every pair of processes is connected by a fair link and at most one process may crash. Let s and r be any two distinct processes. There is no quiescent implementation of quasi reliable send and receive from s to r. This holds even if we assume that only a finite number of messages can be lost.*

Proof. [7] Assume, by contradiction, that there exists a quiescent implementation I of quasi reliable $\mathsf{qr\text{-}send}_{s,r}$ and $\mathsf{qr\text{-}receive}_{r,s}$. The basic intuition behind the proof is to construct a run R_1 where s qr-sends a message to r, but r crashes. Since the implementation of qr-send and qr-receive is quiescent, only a finite number of messages are sent to r in R_1. We then construct a similar run R_2 where s qr-sends a message to r, r does not crash, but the finite number of messages sent to r are lost. Runs R_1 and R_2 are indistinguishable from the point of view of r, so r never qr-receives the message — a contradiction. It turns out that to construct run R_1, we need another run R_0. This is because we allow the quiescent implementation of qr-send and qr-receive to send a finite number of "initialization" messages (see Footnote 6). We now describe runs R_0, R_1 and R_2 in more details.

In run R_0, s qr-sends no messages, all processes are correct, processes take steps in round-robin fashion and every time a process takes a step it receives the earliest message sent to it that it did not yet receive. Since I is quiescent, there is a time t_0 after which no messages are sent or received. By the Uniform Integrity property of qr-send and qr-receive, process r never qr-receives any message.

Run R_1 is identical to run R_0 up to time t_0; at time $t_0 + 1$, s qr-sends M to r, and r crashes; after time $t_0 + 1$, no processes crash, and every time a process takes a step, it receives the earliest message sent to it that it did not yet receive. Since I is quiescent, there is a time $t_1 > t_0$ after which no messages are sent or received.

In run R_2, r behaves exactly as in run R_0 (in particular, r does not crash and r receives a message m in R_2 whenever it receives m in R_0); all other processes behave exactly as in run R_1 (in particular, a process $p \neq r$ receives a message m in R_2 whenever it receives m in R_1). Note that, in R_2, if messages are sent to or from r after time t_0, then they are never received.

We now show that in R_2 all links satisfy the Uniform Integrity property. Assume that for some $k \geq 1$, some process q receives m from some process p k times by time t. There are several cases. (1) If $q = r$ then r receives m from p k times in R_0 by time t (since r behaves in the same way in R_0 and R_2). In R_0, by the Uniform Integrity property of the links p sent m to r at least k times before time t. This happens by time t_0, since there are no sends in R_0 after time t_0. Note that by time t_0, p behaves exactly in the same way in R_0, R_1 and R_2. Thus p sent m to r at least k times before time t in R_2. (2) If $q \neq r$ and $p = r$, then q receives m from r k times in R_1 by time t (since q behaves in the same way in R_1 and R_2). In R_1, by the Uniform Integrity property of the links, r sent m to q at least k times before time t. This happens by time t_0, since r crashes at time $t_0 + 1$ in R_1. By time t_0, r behaves exactly in the same way in R_0, R_1 and R_2. Thus r sent m to q at least k times before time t in R_2. (3) If $q \neq r$ and $p \neq r$, then q receives m from p k times in R_1 by time t (since q behaves in the same way in R_1 and R_2). In R_1, by the Uniform Integrity property of the links, p sent m to q at least k times before time t. Note that p behaves exactly in the same way in R_1 and R_2. Thus p sent m to q at least k times in R_2 before time t. Therefore, in R_2 all links satisfy the Uniform Integrity property.

We next show that in R_2 all links satisfy the Fairness property, and in fact only a finite number of messages are lost. Note that r sends only a finite number of messages in R_0 (since it does not send messages after time t_0), and every process $p \neq r$ sends only a finite number of messages in R_1 (since it does not send messages after time t_1). So, by construction of R_2, all processes send only a finite number of messages in R_2. Therefore, only a finite number of messages are lost, and in R_2 all links satisfy the Fairness property.

[7] This theorem is actually a corollary of Theorem 32 and the fact that the eventually perfect failure detector $\diamond\mathcal{P}$ cannot be implemented. The proof of Theorem 32, however, uses some complex arguments that obscure the intuition behind Theorem 1. We prefer to give a self-contained and direct proof that does not use Theorem 32.

We conclude that R_2 is a possible run of I in a network with fair links that lose only a finite number of messages. Note that in R_2: (a) both s and r are correct; (b) s $\mathsf{qr\text{-}send}$s M to r; and (c) r does not $\mathsf{qr\text{-}receive}$ M. This violates the Quasi No Loss property of $\mathsf{qr\text{-}send}_{s,r}$ and $\mathsf{qr\text{-}receive}_{r,s}$, and so I is not an implementation of $\mathsf{qr\text{-}send}_{s,r}$ and $\mathsf{qr\text{-}receive}_{r,s}$ — a contradiction. □

Theorem 1 and Remark 1 immediately imply:

Corollary 2 *There is no quiescent implementation of reliable broadcast in a network where a process may crash and links may lose a finite number of messages.*

The above results show that quiescent reliable communication cannot be achieved in a system without failure detectors. The rest of this paper explores the use of failure detectors to solve this problem.

5 The Weakest Failure Detector with Bounded Output Size for Quiescent Reliable Communication

In practice, and in much of the previous literature, the output of a failure detector is just a set of processes suspected to have failed. One such failure detector, namely $\diamond\mathcal{P}$, can be used to achieve quiescent reliable communication. However, $\diamond\mathcal{P}$ is not implementable in asynchronous systems. Can we achieve quiescent reliable communication with a failure detector that outputs a set of suspects and is implementable?

In this section we show that the answer is no. In fact, we prove a stronger result: Among all failure detectors *with bounded output size* (these include all failure detectors that output a set of suspects) the weakest one for achieving quiescent reliable communication is $\diamond\mathcal{P}$ — which is not implementable. In contrast, if we do not bound the output size, quiescent reliable communication can be solved with $\mathcal{HB}$ — which is implementable. This shows that failure detectors with bounded output size have some inherent limitations.

We prove our result with respect to a problem that we call *Single-Shot Reliable Send and Receive*. This problem is weaker than quasi reliable send and receive, and reliable broadcast, and thus our result immediately applies to those problems as well.

In Section 5.1, we explain what it means for a failure detector to be weaker than another one. Section 5.2 defines the Single-Shot Reliable Send and Receive problem. We then proceed to prove our main result under some reasonable simplifying assumption. We first give a rough outline of this proof (Section 5.3), and then the proof itself (Sections 5.4 and 5.5). In the Appendix, we give the full proof without the simplifying assumption.

5.1 Failure Detector Transformations

Failure detectors can be compared via algorithmic transformations [CT96, CHT96]. A transformation algorithm $T_{\mathcal{D}\to\mathcal{D}'}$ uses failure detector $\mathcal{D}$ to emulate $\mathcal{D}'$, as we now explain. At each process p, the algorithm $T_{\mathcal{D}\to\mathcal{D}'}$ maintains a variable $\mathcal{D}'_p$ that emulates the output of $\mathcal{D}'$ at p. Let $H_{\mathcal{D}'}$ be the history of all the $\mathcal{D}'$ variables in a run R of $T_{\mathcal{D}\to\mathcal{D}'}$, i.e., $H_{\mathcal{D}'}(p,t)$ is the value of $\mathcal{D}'_p$ at time t in run R. Algorithm $T_{\mathcal{D}\to\mathcal{D}'}$ *transforms* $\mathcal{D}$ *into* $\mathcal{D}'$ *in environment* $\mathcal{E}$ if and only if for every $F \in \mathcal{E}$ and every run $R = (F, H_{\mathcal{D}}, I, S, T)$ of $T_{\mathcal{D}\to\mathcal{D}'}$ using $\mathcal{D}$, we have $H_{\mathcal{D}'} \in \mathcal{D}'(F)$. Intuitively, since $T_{\mathcal{D}\to\mathcal{D}'}$ is able to use $\mathcal{D}$ to emulate $\mathcal{D}'$, $\mathcal{D}$ provides at least as much information about process failures as $\mathcal{D}'$ does, and we say that $\mathcal{D}'$ *is weaker than* $\mathcal{D}$ *in* $\mathcal{E}$.

Note that, in general, $T_{\mathcal{D}\to\mathcal{D}'}$ need not emulate *all* the failure detector histories of $\mathcal{D}'$ (in environment $\mathcal{E}$); what we do require is that all the failure detector histories it emulates be histories of $\mathcal{D}'$ (in that environment).

5.2 Single-Shot Reliable Send and Receive

The Single-Shot Reliable Send and Receive problem is defined in terms of two communication primitives, called s-send and s-receive. Each process can s-send a single bit once to one process of its choice, if it wishes to do so (but it is also possible that no process in the system ever s-sends any bit). The s-send and s-receive primitives must satisfy the following property. For any two correct processes p and q, and any $b \in \{0, 1\}$: p s-sends b to q if and only if q s-receives b from p.

An implementation $\mathcal{I}$ of s-send and s-receive is quiescent if it sends only a finite number of messages throughout the network.

5.3 Intuitive Overview of the Simple Proof

Let $\mathcal{D}$ be a failure detector with bounded output size, i.e., the range of $\mathcal{D}$ is finite. Suppose $\mathcal{D}$ can be used to solve the Single-Shot Reliable Send and Receive problem with a quiescent algorithm $\mathcal{I}$ ($\mathcal{I}$ is also called the implementation of s-send and s-receive). We show that $\mathcal{D}$ can be transformed to $\diamond\mathcal{P}$.

The proof that follows makes the simplifying assumption that $\mathcal{I}$ does not have an "initialization phase" that requires the sending of messages. In other words, we assume that $\mathcal{I}$ is such that if no process ever s-sends any bit, then no process ever sends any messages. This reasonable assumption allows us to simplify the proof and illustrate the basic ideas. In the appendix, we give the full proof.

Since the range of $\mathcal{D}$ is finite, then for every failure detector history H of $\mathcal{D}$: (a) each failure detector module outputs some values infinitely often (these are the "limit values"), and (b) there is a time after which it outputs only limit values. Let v be a limit value for process p and H. A crucial observation is that with H it is possible to construct runs such that whenever p takes a step it always gets v from its failure detector module. It is easy to generalize the notion of a limit value for p to a limit vector for a set of processes P: A vector f (with a value for every process in the system) is a limit vector for P and H if, for each process p in P, the failure detector module of p outputs $f(p)$ infinitely often in H. Note that with H it is possible to construct runs such that whenever a process p in P takes a step, it obtains $f(p)$ from its failure detector module. We say that vector f hints that P is the set of all correct processes, if f could occur as a limit vector for P when P is the set of correct processes (more precisely, f is a limit vector for P in a history $H \in \mathcal{D}(F)$ where $\mathit{correct\_proc}(F) = P$).

Consider a failure detector history H that can occur when P is the set of all correct processes. Let f be any limit vector for P and H. Clearly, f hints that P is the set of all correct processes. Can f also hint that a proper subset P' of P is the set of all correct processes? The answer is no. As we argue next, this is because with $\mathcal{D}$, a process in P' should be able to s-send a bit to a process q in $P \setminus P'$ and to do so quiescently using $\mathcal{I}$.

Suppose, for contradiction, that f hints that P' is the set of all correct processes. Then we can construct a run R_1 of $\mathcal{I}$ where (a) P' is indeed the set of all correct processes, (b) processes in P' are scheduled such that whenever they take steps they get f from their failure detector module, (c) some process p in P' s-sends a bit b to some process q in $P \setminus P'$, and (d) processes in $P \setminus P'$ never take a step. Because the implementation $\mathcal{I}$ is quiescent, in R_1 eventually all processes in P' (including p) stop sending messages — they give up on trying to transmit b to q.

Since f also hints that P is the set of correct processes, we can create another run R_2 of $\mathcal{I}$ where (a) P is the set of correct processes, (b) processes in P are scheduled such that whenever they take steps they get f from their failure detector module, (c) p s-sends b to q, (d) messages sent between processes in P' and processes in $P \setminus P'$ are lost. Note that from the point of view of processes in P', run R_2 is indistinguishable from run R_1. Thus, in R_2 eventually all processes in P' stop sending messages — they give up on trying to transmit b to q. So, in R_2 process q never receives any messages, and thus it does not s-receive b

from p. Since p and q are correct in R_2, the implementation $\mathcal{I}$ of s-send and s-receive is incorrect — a contradiction. Thus, f cannot hint that P' is the set of all correct processes.

Let E_P be the set of all vectors that hint that P is the set of correct processes (this set is determined by $\mathcal{D}$). The algorithm that transforms $\mathcal{D}$ to $\diamond\mathcal{P}$ uses a predetermined "table of hints" containing, for each possible P, the set E_P.

The transformation algorithm works as follows. Each process p periodically sends its current failure detector output to every process, and maintains two variables: f and $Order$. Vector f stores the last failure detector value received from each process, and $Order$ is an ordered set of processes. Whenever p receives a failure detector value from another process q, it records that value in $f(q)$ and moves q to the front of $Order$. Let P be the set of correct processes in this run. Note that: (a) eventually f is a limit vector for P, and (b) the correct processes percolate to the front of $Order$ (processes that crash end up at the tail), so that eventually P is some prefix of $Order$.

To satisfy the properties of $\diamond\mathcal{P}$, p must eventually output the complement of P. By (b) above, eventually P is the largest prefix of $Order$ that contains correct processes. To find this maximal prefix, p repeatedly uses its current value of f and the predetermined table of hints, as follows. For each prefix P' of $Order$, in order of increasing size, p checks if f hints that P' is the set of all correct processes, i.e., $f \in E_{P'}$, and if so p outputs the complement of P'. This works because, as we argued above, any limit vector f for P: (1) hints that P is the set of all correct processes, and (2) cannot hint that a proper subset P' of P is the set of all correct processes. This concludes the overview of the proof (the reader should understand why the argument above breaks down without the simplifying assumption).

We next give the actual proof. The transformation algorithm $T_{\mathcal{D}\to\diamond\mathcal{P}}$ uses a table which is determined *a priori* from $\mathcal{D}$ (this is the "table of hints" in our intuitive explanation). We first define this table and show some of its properties (Section 5.4). We then describe and prove the correctness of the transformation algorithm $T_{\mathcal{D}\to\diamond\mathcal{P}}$ that uses this table (Section 5.5).

5.4 The Predetermined Table

Let $\mathcal{E}$ be an environment and $\mathcal{D}$ be any failure detector with finite range $\mathcal{R} = \{v_1, v_2, \ldots, v_\ell\}$. Let $\mathcal{I}$ be a quiescent implementation of s-send and s-receive that uses $\mathcal{D}$ in environment $\mathcal{E}$. Assume that if no process s-sends any bit then $\mathcal{I}$ does not send any messages (this simplifying assumption is removed in the Appendix).

Given $v_j \in \mathcal{R}$, a process $p \in \Pi$, and a failure detector history H with range $\mathcal{R}$, we say that v_j is *a limit value for* p *and* H if, for infinitely many t, $H(p,t) = v_j$. Let f be an assignment of failure detector values to every process in Π, i.e., $f : \Pi \longrightarrow \mathcal{R}$. Let P be a non-empty set of processes. We say that f *is a limit vector for* P *and* H if for all $p \in P$, $f(p)$ is a limit value for p and H. The set of all limit vectors for P and H is denoted $L_P(H)$. Let $E_P^{\mathcal{D},\mathcal{E}} = \{f \mid \exists F \in \mathcal{E}, \exists H \in \mathcal{D}(F) : P = \mathit{correct\_proc}(F) \text{ and } f \in L_P(H)\}$. Roughly speaking, $E_P^{\mathcal{D},\mathcal{E}}$ is the set of limit vectors that could occur when P is the set of correct processes.

The table used by the transformation algorithm $T_{\mathcal{D}\to\diamond\mathcal{P}}$ consists of all the sets $E_P^{\mathcal{D},\mathcal{E}}$ where P ranges over all non-empty subset of processes. Note that this table is finite. We omit the superscript $\mathcal{D},\mathcal{E}$ from $E_P^{\mathcal{D},\mathcal{E}}$ whenever it is clear from the context.

Lemma 3 *Let $F \in \mathcal{E}$, $P = \mathit{correct\_proc}(F)$ and $H \in \mathcal{D}(F)$. Assume $P \neq \emptyset$. If $f \in L_P(H)$ then $f \in E_P$ and $f \notin E_{P'}$ for every P' such that $\emptyset \subset P' \subset P$.*

Proof. Let $f \in L_P(H)$. The fact that $f \in E_P$ is immediate from the definition of E_P. Let P' be such that $\emptyset \subset P' \subset P$. Suppose, for contradiction, that $f \in E_{P'}$. Then there exists a failure pattern $F' \in \mathcal{E}$ and $H' \in \mathcal{D}(F')$ such that $P' = \mathit{correct\_proc}(F')$ and $f \in L_{P'}(H')$.

We now obtain a contradiction by using the quiescent implementation $\mathcal{I}$ of s-send and s-receive. Let p be a process in P' and q be a process in $P \setminus P'$. We construct two runs, R_1 and R_2 of $\mathcal{I}$ using $\mathcal{D}$, as follows:

- Run R_1 has failure pattern F' and failure detector history H'. Initially p s-sends some bit b to q. Processes in P' take steps and those in $\Pi \setminus P'$ do not. Processes in P' take steps in round-robin fashion such that every time a process $r \in P'$ takes a step, it obtains $f(r)$ from its failure detector module (since $f \in L_{P'}(H')$, $f(r)$ is a limit value for r and H'). Moreover, every process in P' receives every message sent to it.

  Since $\mathcal{I}$ is quiescent, there is a time t_1 after which no messages are sent or received. Assume without loss of generality that at time t_1 all processes in P' took the same number k of steps (otherwise, choose another time $t_1' > t_1$). Note that all messages in R_1 are sent within the finite period of time $[0, t_1]$. Thus, the fact that all processes in P' receive all the messages sent to them is consistent with the link failure pattern of F' (even if in F' some of the links are not fair).

- Run R_2 has failure pattern F and failure detector history H. Initially, processes in R_2 behave as in R_1: p s-sends some bit b to q; moreover, each process in P' take the same k steps as in R_1, and process in $\Pi \setminus P'$ do not take any steps. More precisely, processes in P' take steps in round-robin fashion such that every time a process $r \in P'$ takes a step, it obtains $f(r)$ from its failure detector module (since $f \in L_P(H)$ and $r \in P' \subset P$, $f(r)$ is a limit value for r and H). Moreover, every process in P' receives every message sent to it, and all messages sent to processes in $\Pi \setminus P'$ are lost. This goes on until each process in P' takes k steps, exactly as in R_1.[8]

  Let t_2 be the time when this happens. After t_2, processes in P take steps in round-robin fashion such that every time a process $r \in P'$ takes a step, it obtains $f(r)$ from its failure detector module (it does not matter what a process $r \in P \setminus P'$ gets from its failure detector module, as long as it is compatible with H). Moreover, after t_2 no process s-sends any bit. This completes the description of run R_2.

  Note that at time t_2, each process in P' is in the same state as in run R_1 at time t_1. Moreover, each process in $P \setminus P'$ is in its initial state. By a simple induction argument we can show that after time t_2 in R_2: (a) processes in P' continue to behave as in R_1, (b) processes in $P \setminus P'$ behave as if they were in a run of $\mathcal{I}$ in which no process ever s-sends any bit, and (c) no process sends any message (this induction uses the simplifying assumption that in a run in which there are no s-sends, no process sends any message). Therefore, in R_2, process q (which is in $P \setminus P'$) never receives any messages. This implies that q does not s-receive b from p.

Note that in R_2: (a) both p and q are correct; (b) p s-sends b to q; and (c) q does not s-receive b from p. Thus, $\mathcal{I}$ is not a correct implementation of s-send and s-receive — a contradiction. □

5.5 The Transformation Algorithm

The algorithm $T_{\mathcal{D}\to\mathcal{D}'}$ that transforms $\mathcal{D}$ to an eventually perfect failure detector $\mathcal{D}' = \diamond\mathcal{P}$ in environment $\mathcal{E}$ is shown in Fig. 1. $T_{\mathcal{D}\to\mathcal{D}'}$ uses the table of sets E_P (for all non-empty subsets of processes P) that has been determined *a priori* from the given $\mathcal{D}$ and $\mathcal{E}$. It also uses an implementation of qr-send and qr-receive between every pair of processes. A simple implementation is by repeated retransmissions and diffusion (it does not have to be quiescent).

All variables are local to each process. Vector f stores the last failure detector value that p qr-received from each process; *Order* is an ordered set that records the order in which the last failure detector value from each process was qr-received; $\mathcal{D}'_p$ denotes the output of the eventually perfect failure detector that p is simulating (a set of processes that p currently suspects).

[8] This behavior of the links is consistent with F because for any finite period of time, any link (whether fair or not in F) may lose or not lose any message.

For every process p:

  Initialization:
    for all $q \in \Pi$ **do** $f[q] \leftarrow \bot$
    $Order \leftarrow \emptyset$
    $\mathcal{D}'_p \leftarrow \emptyset$
    { For each $\emptyset \subset P \subseteq \Pi$, the set $E_P^{\mathcal{D},\mathcal{E}}$ is determined *a priori* from $\mathcal{D}$ and $\mathcal{E}$ }

  cobegin
    || *Task 1*:
      repeat periodically
        $v \leftarrow \mathcal{D}_p$ {query $\mathcal{D}$}
        for all $q \in \Pi$ **do** qr-send v **to** q

    || *Task 2*:
      upon qr-receive w **from** q **do** {upon receipt of a failure detector value from q}
        $f[q] \leftarrow w$
        $Order \leftarrow q \mid\mid (Order \setminus \{q\})$ {process q is moved to the front of *Order*}
        if for some $k \geq 1$, $f \in E^{\mathcal{D},\mathcal{E}}_{Order[1..k]}$ **then**
          let k_0 be the smallest such k
          $\mathcal{D}'_p \leftarrow \Pi \setminus Order[1..k_0]$ {suspect processes not in $Order[1..k_0]$}
  coend

Figure 1: Transformation of $\mathcal{D}$ to an eventually perfect failure detector $\mathcal{D}'$ in environment $\mathcal{E}$

In Task 1, each process p periodically qr-sends the output of its failure detector module $\mathcal{D}_p$ to every process q. Upon the qr-receipt of a failure detector value from process q in Task 2, process p enters it into $f[q]$, and moves q to the front of $Order$. Then, p checks if there is some prefix $Order[1..k]$ of $Order$ such that $f \in E_{Order[1..k]}$. If there is, it sets $\mathcal{D}'$ to the complement of the smallest such prefix.

We now show that the failure detector constructed by this algorithm, namely $\mathcal{D}'$, is an eventually perfect failure detector. Consider a run of this algorithm with failure pattern $F \in \mathcal{E}$ and failure detector history $H \in \mathcal{D}(F)$, such that $correct\_proc(F) \neq \emptyset$. Let t be the number of processes that crash in F, i.e., $t = |\Pi \setminus correct\_proc(F)|$. Henceforth, p denotes a correct process in F and variables f and $Order$ are local to p.

Lemma 4 *There is a time after which (1)* $Order[1..n{-}t] = correct\_proc(F)$*, and (2)* $f \in L_{Order[1..n-t]}(H)$.[9]

Proof. Part (1) is clear from the way $Order$ is updated, the fact that p keeps qr-receiving failure detector values from every correct process, and the fact that p eventually stops qr-receiving messages from processes that crash. Part (2) of the lemma follows from part (1) and the fact that the range $\mathcal{R}$ of $\mathcal{D}$ is finite. □

Corollary 5 *There is a time after which (1)* $f \in E_{Order[1..n-t]}$ *and (2) for all* $1 \leq k < n-t$, $f \notin E_{Order[1..k]}$.

Proof. By Lemma 4, there is a time t_0 after which $f \in L_{Order[1..n-t]}(H)$ and $Order[1..n-t] = correct\_proc(F)$. So after time t_0, by Lemma 3, $f \in E_{Order[1..n-t]}$. This shows (1). Let k be such

[9] This does not mean that eventually the values of variables f and $Order$ at p stop changing. It means that, although they may continue to change forever, eventually the predicates (1) and (2) are true forever at p.

that $1 \leq k < n-t$. After t_0, $\emptyset \subset Order[1..k] \subset correct\_proc(F)$, and $f \in L_{correct\_proc(F)}(H)$. So, by Lemma 3, $f \notin E_{Order[1..k]}$. This shows (2). □

Corollary 6 *There is a time after which $\mathcal{D}'_p = \Pi \setminus correct\_proc(F)$.*
Proof. By Corollary 5 and the algorithm, there is a time after which the k_0 selected in line 20 is always $n-t$. Now apply Lemma 4 part (1). □

By Corollary 6, we have:

Theorem 7 *Consider an asynchronous system subject to process crashes and message losses. Suppose failure detector $\mathcal{D}$ with finite range can be used to solve the Single-Shot Reliable Send and Receive problem in environment $\mathcal{E}$, and that the implementation is quiescent. Assume further that if no process ever* s-send*s any bit then this implementation does not send any messages. Then $\mathcal{D}$ can be transformed (in environment $\mathcal{E}$) to the eventually perfect failure detector $\Diamond\mathcal{P}$.*

Theorems 7 and 32 imply that if we restrict ourselves to failure detectors that output a set of suspects, we cannot achieve quiescent reliable communication with a failure detector that can be implemented. Thus, we next introduce $\mathcal{HB}$, a failure detector that does *not* output a set of suspects. $\mathcal{HB}$ can be used to achieve quiescent reliable communication and it is implementable.

6 Definition of $\mathcal{HB}$

A *heartbeat failure detector* $\mathcal{D}$ has the following features. The output of $\mathcal{D}$ at each process p is a list $(p_1, n_1), (p_2, n_2), \ldots, (p_k, n_k)$, where $p_1, p_2, \ldots, p_k$ are the neighbors of p, and each n_j is a nonnegative integer. Intuitively, n_j increases while p_j has not crashed, and stops increasing if p_j crashes. We say that n_j is *the heartbeat value of p_j at p*. The output of $\mathcal{D}$ at p at time t, namely $H(p,t)$, will be regarded as a vector indexed by the set $\{p_1, p_2, \ldots, p_k\}$. Thus, $H(p,t)[p_j]$ is n_j. The *heartbeat sequence of p_j at p* is the sequence of the heartbeat values of p_j at p as time increases. $\mathcal{D}$ satisfies the following properties:

- *$\mathcal{HB}$-Completeness*: At each correct process, the heartbeat sequence of every neighbor that crashes is bounded:

$$\forall F, \forall H \in \mathcal{D}(F), \forall p \in correct\_proc(F), \forall q \in crashed\_proc(F) \cap neighbor(p),$$
$$\exists K \in \mathbf{N}, \forall t \in \mathcal{T} : H(p,t)[q] \leq K$$

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

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

  - At each correct process, the heartbeat sequence of every correct neighbor is unbounded:

$$\forall F, \forall H \in \mathcal{D}(F), \forall p \in correct\_proc(F), \forall q \in correct\_proc(F) \cap neighbor(p),$$
$$\forall K \in \mathbf{N}, \exists t \in \mathcal{T} : H(p,t)[q] > K$$

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

It is easy to generalize the definition of $\mathcal{HB}$ so that the failure detector module at each process p outputs the heartbeat of every process in the system [ACT], rather than just the heartbeats of the neighbors of p, but we do not need this generality here.

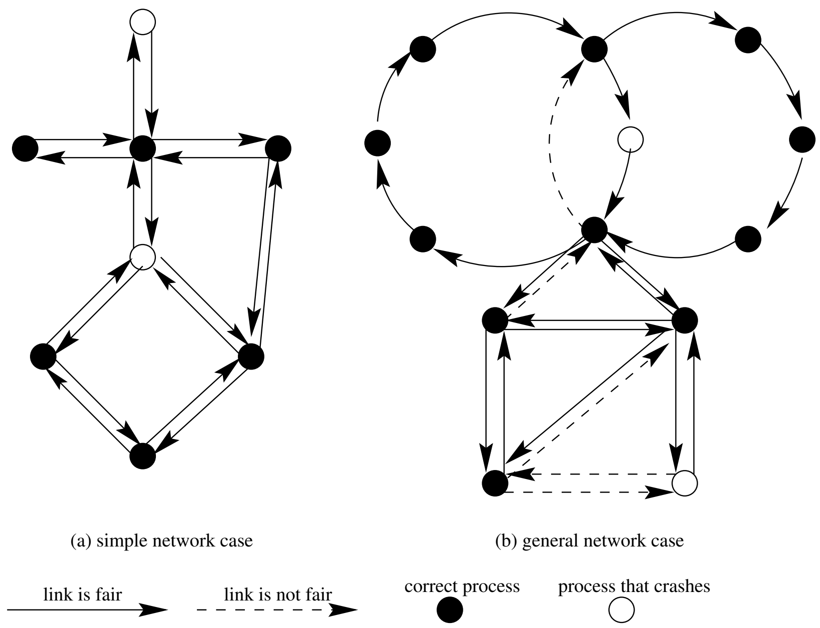

Figure 2: Examples of the simple and general network cases

7 Quiescent Reliable Communication Using $\mathcal{HB}$

The communication networks that we consider are not necessarily completely connected, but we assume that every pair of correct processes is connected through a fair path. We first consider a simple type of such networks, in which every link is assumed to be bidirectional[10] and fair (Fig. 2a). This assumption, a common one in practice, allows us to give efficient and simple algorithms. We then drop this assumption and treat a more general type of networks, in which some links may be unidirectional and/or not fair (Fig. 2b). For both network types, we give quiescent reliable communication algorithms that use $\mathcal{HB}$. Our algorithms have the following 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.

In our algorithms, $\mathcal{D}_p$ denotes the current output of the failure detector $\mathcal{D}$ at process p.

7.1 The Simple Network Case

We assume that all links in the network are bidirectional and fair (Fig. 2a). In this case, the algorithms are very simple. We first give a quiescent implementation of quasi reliable qr-send$_{s,r}$ and qr-receive$_{r,s}$ for the case $r \in neighbor(s)$. For s to qr-send a message m to r, it repeatedly sends m to r every time the heartbeat of r increases, until s receives $ack(m)$ from r. Process r qr-receives m from s the first time it receives m from s, and r sends $ack(m)$ to s every time it receives m from s.

From this implementation, and Remark 2 in Section 3.3, we can obtain a quiescent implementation of reliable broadcast. Then, from Remark 1, we can obtain a quiescent implementation of quasi reliable send and receive for every pair of processes.

[10] In our model, this means that link $p \rightarrow q$ is in the network if and only if link $q \rightarrow p$ is in the network. In other words, $q \in neighbor(p)$ if and only if $p \in neighbor(q)$.

7.2 The General Network Case

In this case (Fig. 2b), some links may be unidirectional, e.g., the network may contain several unidirectional rings that intersect with each other. Moreover, some links may not be fair (and processes do not know which ones are fair).

Achieving quiescent reliable communication in this type of network is significantly more complex than before. For instance, suppose that we seek a quiescent implementation of quasi reliable send and receive. In order for the sender s to qr-send a message m to the receiver r, it has to use a diffusion mechanism, even if r is a neighbor of s (since the link $s \rightarrow r$ may not be fair). Because of intermittent message losses, this diffusion mechanism needs to ensure that m is repeatedly sent over fair links. But when should this repeated send stop? One possibility is to use an acknowledgement mechanism. Unfortunately, the link in the reverse direction may not be fair (or may not even be part of the network), and so the acknowledgement itself has to be diffused. But diffusing the acknowledgements quiescently and reliably introduces a "chicken and egg" problem. We now explain how our algorithms avoid this problem.

We give a quiescent implementation of reliable broadcast in Figure 3. This implementation can be used to obtain quasi reliable send and receive between every pair of processes (see Remark 1 in Section 3.3). For each message m that is 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 $got_p[m]$.[11] 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 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 19 through 27 is executed atomically.[12] Each concurrent execution of the *diffuse* task (lines 9 to 17) has its own copy of all the local variables in this task.

We now outline the proof that, for the general network case, Fig. 3 is a quiescent implementation of reliable broadcast that uses $\mathcal{HB}$. The first few lemmata are obvious.

Lemma 8 (Uniform Integrity) *For every message m, every process delivers message m at most once, and only if m was previously broadcast by $sender(m)$.*

Lemma 9 (Validity) *If a correct process broadcasts a message m, then it eventually delivers m.*

Lemma 10 *For any processes p and q, (1) if at some time t, $q \in got_p[m]$ then $q \in got_p[m]$ at every time $t' \geq t$; (2) When $got_p[m]$ is initialized, $p \in got_p[m]$; (3) if $q \in got_p[m]$ then q delivered m.*

Lemma 11 *For every m and path, there is a finite number of distinct messages of the form $(m, *, path)$.*

[11] 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; in the proof of Lemma 15 we will indicate exactly where the sending to *every* neighbor whose heartbeat increased is necessary.

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

```
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 the heartbeat failure detector }
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
15                     send (m, got[m], p) to q
16                 prev_hb ← hb
17         until neighbor(p) ⊆ got[m]
18
19     upon receive (m, got_msg, path) from q do
20         if p has not previously executed deliver(m) then
21             deliver(m)
22             got[m] ← {p}
23             fork task diffuse(m)
24         got[m] ← got[m] ∪ got_msg
25         path ← path · p
26         for all q such that q ∈ neighbor(p) and q appears at most once in path do
27             send (m, got[m], path) to q
```

Figure 3: General network case — quiescent implementation of broadcast and deliver using $\mathcal{HB}$

Lemma 12 *If some process sends a message of the form $(m, *, path)$, then no process appears more than twice in path.*

Lemma 13 *Suppose link $p \to q$ is fair, and p and q are correct processes. If p delivers a message m, then q eventually delivers m.*

Proof. Suppose, by contradiction, that p delivers m and q never delivers m. Since p delivers m and it is correct, it forks task $diffuse(m)$. Since q does not deliver m, by Lemma 10 part (3) q never belongs to $got_p[m]$. Since p is correct, this implies that p executes the loop in lines 11–17 an infinite number of times. Since q is a correct neighbor 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 an infinite number of times, and so p sends a message of the form $(m, *, p)$ to q an infinite number of times. By Lemma 11, there exists a subset $g_0 \subseteq \Pi$ such that p sends message (m, g_0, p) infinitely often to q. So, by the Fairness property of link $p \to q$, q eventually receives (m, g_0, p). Therefore, q delivers m. This contradicts the assumption that q does not deliver m. $\square$

Lemma 14 (Agreement) *If a correct process p delivers a message m, then every correct process q eventually delivers m.*

Proof (Sketch). By successive applications of Lemma 13 over any fair path from p to q. $\square$

We now show that the algorithm in Fig. 3 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. This implies that the implementation sends only a finite number of messages when broadcast is invoked a finite number of times.

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 15 *Let p be a correct process and q be a correct neighbor of p. If p forks task $diffuse(m)$, then eventually condition $q \in got_p[m]$ holds forever.*

Proof. By Lemma 10 part (1), we only need to show that eventually q belongs to $got_p[m]$. Suppose, by contradiction, that q never belongs to $got_p[m]$. Let $(p_1, p_2, \ldots, p_{k'})$ be a simple fair path[13] from p to q with $p_1 = p$ and $p_{k'} = q$. Let $(p_{k'}, p_{k'+1}, \ldots, p_k)$ be a simple fair path 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.

We claim that for each $j \in \{1, \ldots, k-1\}$, 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 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, since q never belongs to $got_p[m]$ and q is a neighbor of $p_1 = p$, then p_1 executes the loop in lines 11–17 an infinite number of times. Since q is a correct neighbor of p_1, the $\mathcal{HB}$-Accuracy property guarantees that the heartbeat sequence of q at p_1 is nondecreasing and unbounded. Thus, 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 a correct neighbor of p_1, its heartbeat sequence is nondecreasing and unbounded, and so p_1 sends messages of the form $(m, *, p_1)$ to p_2 an infinite number of times.[14] By Lemma 11, there is some g_1 such that p_1 sends (m, g_1, p_1) to p_2 an infinite number of times. Note that Lemma 10 parts (1) and (2) implies 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 g_j containing $\{p_1, p_2, \ldots, p_j\}$. By the Fairness property of 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 11, 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. $\square$

Corollary 16 *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 correct then eventually condition $q \in got_p[m]$ holds forever by Lemma 15. If q crashes, 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 $diffuse(m)$. $\square$

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

[14] This is where the proof uses the fact that p sends 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).

Lemma 17 (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 12, 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 16 shows that there is a time after which p_0 stops sending messages in its task $\mathit{diffuse}(m)$. Since p_0 only sends a message in task $\mathit{diffuse}(m)$ or in line 27, then p_0 sends messages of the form $(m, *, path_0)$ in line 27 an infinite number of times. For each $(m, *, path_0)$ that p_0 sends in line 27, 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 8, 9, 14, and 17 we have:

Theorem 18 *For the general network case, the algorithm in Fig. 3 is a quiescent implementation of reliable broadcast that uses $\mathcal{HB}$.*

From this theorem and Remark 1 in Section 3.3 we have:

Corollary 19 *In the general network case, quasi reliable send and receive between every pair of processes can be implemented with a quiescent algorithm that uses $\mathcal{HB}$.*

8 Implementations of $\mathcal{HB}$

We now give implementations of $\mathcal{HB}$ for the two types of communication networks that we considered in the previous sections. These implementations do not use timeouts.

8.1 The Simple Network Case

We assume all links in the network are bidirectional and fair (Fig. 2a). In this case, the implementation is obvious. Each process periodically sends a HEARTBEAT message to all its neighbors; upon the receipt of such a message from process q, p increases the heartbeat value of q.

8.2 The General Network Case

In this case some links are unidirectional and/or not fair (Fig. 2b). The implementation is more complex than before because each HEARTBEAT has to be diffused, and this introduces the following problem: when a process p receives a HEARTBEAT message it has to relay it even if this is not the first time p receives such a message. This is because this message could be a new "heartbeat" from the originating process. But this could also be an "old" heartbeat that cycled around the network and came back, and p must avoid relaying such heartbeats.

The implementation is given in Fig. 4. Every process p executes two concurrent tasks. In the first task, p periodically sends 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 message from process q, p increases

```
For every process p:

    Initialization:
        for all q ∈ neighbor(p) do D_p[q] ← 0

    cobegin
        || Task 1:
            repeat periodically
                for all q ∈ neighbor(p) do send (HEARTBEAT, p) to q

        || Task 2:
            upon receive (HEARTBEAT, path) from q do
                for all q such that q ∈ neighbor(p) and q appears in path do
                    D_p[q] ← D_p[q] + 1
                path ← path · p
                for all q such that q ∈ neighbor(p) and q does not appear in path do
                    send (HEARTBEAT, path) to q
    coend
```

Figure 4: General network case — implementation of $\mathcal{HB}$

the heartbeat values of all its neighbors that appear in $path$. Then p appends itself to $path$ and forwards message $(\mathsf{HEARTBEAT}, path)$ to all its neighbors that do not appear in $path$.

We now show that, for the general network case, the algorithm in Fig. 4 implements $\mathcal{HB}$.

Lemma 20 *At every process p, the heartbeat sequence of every neighbor q is nondecreasing.*

Proof. Obvious. □

Lemma 21 *At each correct process p, the heartbeat sequence of every correct neighbor q is unbounded.*

Proof (Sketch). It is possible that link $q \to p$ is not fair or not even in the network. However, there is a simple fair path $P = (p_1, \ldots, p_k)$ from q to p with $p_1 = q$ and $p_k = p$. Process $p_1 = q$ sends its heartbeat to all its neighbors infinitely often. Since the links $p_1 \to p_2$, …, $p_{k-1} \to p_k$ are fair and each p_j is correct, the heartbeats of q are relayed infinitely often through that path, and $p_k = p$ receives them infinitely often. □

Corollary 22 ($\mathcal{HB}$-Accuracy) *At each process the heartbeat sequence of every neighbor is nondecreasing, and at each correct process the heartbeat sequence of every correct neighbor is unbounded.*

Proof. From Lemmata 20 and 21. □

The proofs of the next two Lemmata are obvious.

Lemma 23 *If some process p sends $(\mathsf{HEARTBEAT}, path)$ then (1) p is the last process in $path$ and (2) no process appears twice in $path$.*

Lemma 24 *Let p, q be processes, and $path$ be a non-empty sequence of processes. If p receives message $(\mathsf{HEARTBEAT}, path \cdot q)$ an infinite number of times, then q receives message $(\mathsf{HEARTBEAT}, path)$ an infinite number of times.*

Lemma 25 (**$\mathcal{HB}$-Completeness**) *At each correct process, the heartbeat sequence of every neighbor that crashes is bounded.*

Proof (Sketch). Let p be a correct process and let q be a neighbor of p that crashes. Suppose that the heartbeat sequence of q at p is not bounded. Then p increments $\mathcal{D}_p[q]$ an infinite number of times. So, for an infinite number of times, p receives messages of the form $(\mathsf{HEARTBEAT}, *)$ with a second component that contains q. By Lemma 23 part (2), 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 such that p receives $(\mathsf{HEARTBEAT}, path)$ an infinite number of times.

Let $path = (p_1, \ldots, p_k)$. Then, for some $j \leq k$, $p_j = q$. If $j = k$ then, by the Uniform Integrity property of the links and by Lemma 23 part (1), q sends $(\mathsf{HEARTBEAT}, path)$ to p an infinite number of times. This contradicts the fact that q crashes. If $j < k$ then, by repeated applications of Lemma 24, we conclude that p_{j+1} receives message $(\mathsf{HEARTBEAT}, (p_1, \ldots, p_j))$ an infinite number of times. Therefore, by the Uniform Integrity property of the links and Lemma 23 part (1), p_j sends $(\mathsf{HEARTBEAT}, (p_1, \ldots, p_j))$ to p_{j+1} an infinite number of times. Since $p_j = q$, this contradicts the fact that q crashes. □

By Corollary 22 and the above lemma, we have:

Theorem 26 *For the general network case, the algorithm in Fig. 4 implements $\mathcal{HB}$.*

9 Stronger Communication Primitives

Quasi reliable send and receive and reliable broadcast are sufficient to solve many problems (see Section 10.1). However, stronger types of communication primitives, namely, *reliable send and receive*, and *uniform reliable broadcast*, are sometimes needed. We now give quiescent implementations of these primitives for systems with process crashes and message losses.

Let t be the number of processes that may crash. [BCBT96] shows that if $t \geq n/2$ (i.e., half of the processes may crash) these primitives cannot be implemented, even if we assume that links may lose only a finite number of messages and we do not require that the implementation be quiescent.

We now show that if $t < n/2$ then there are quiescent implementations of these primitives for the two types of network considered in this paper. The implementations that we give here are simple and modular but are inefficient (in terms of number of messages sent). More efficient ones can be obtained by modifying the algorithms in Sections 7.1 and 7.2. Hereafter, we assume that $t < n/2$.

9.1 Reliable Send and Receive

Consider any two distinct processes s and r. We define *reliable send and receive from s to r* in term of two primitives: $\mathsf{r\text{-}send}_{s,r}$ and $\mathsf{r\text{-}receive}_{r,s}$. We require that if a correct process invokes r-send it eventually returns from this invocation. If a process s returns from the invocation of $\mathsf{r\text{-}send}_{s,r}(m)$ we say that s *completes the* r-send *of message m to r*. With quasi reliable send and receive, it is possible that s completes the qr-send of m to r, then s crashes, and r never qr-receives m (even though it does not crash). In contrast, with reliable send and receive primitives, if s completes the r-send of message m to a correct process r then r eventually r-receives m (even if s crashes). More precisely, reliable send and receive satisfy the following properties:

- *Uniform Integrity*: For all $k \geq 1$, if r r-receives m from s exactly k times by time t, then s r-sent m to r at least k times before time t.

```
1  For process s:
2
3      Initialization:
4          seq ← 0                                    { seq is the current sequence number }
5
6      To execute r-send_{s,r}(m):
7          seq ← seq + 1
8          lseq ← seq
9          broadcast(m, lseq, s, r)
10         wait until qr-received (ACK, lseq) from t + 1 processes
11         return
12
13 For every process p:
14
15     upon deliver(m, lseq, s, r) do
16         qr-send_{p,s}(ACK, lseq)
17         if p = r then r-receive_{r,s}(m)
```

Figure 5: Quiescent implementation of $\text{r-send}_{s,r}$ and $\text{r-receive}_{r,s}$ for $n > 2t$

- *No Loss*: For all $k \geq 1$, if r is correct and s completes the r-send of m to r exactly k times by time t, then r eventually r-receives m from s at least k times.[15]

A quiescent implementation of r-send and r-receive can be obtained using a quiescent implementation of reliable broadcast and of qr-send/qr-receive between every pair of processes. Roughly speaking, when s wishes to r-send m to r, it broadcasts a message that contains m, s, r and a fresh sequence number, and then waits to qr-receive $t+1$ acknowledgements for that message before returning from this invocation of r-send. When a process p delivers this broadcast message, it qr-sends an acknowledgement back to s, and if $p = r$ then it also r-receives m from s. This algorithm is shown in Figure 5 (the code consisting of lines 7 and 8 is executed atomically).

9.2 Uniform Reliable Broadcast

The Agreement property of reliable broadcast states that if a *correct* process delivers a message m, then all correct processes eventually deliver m. This requirement allows a *faulty* process (i.e., one that subsequently crashes) to deliver a message that is never delivered by the correct processes. This behavior is undesirable in some applications, such as *atomic commitment* in distributed databases [BT93, Gra78, Had86]. For such applications, a stronger version of reliable broadcast is more suitable, namely, *uniform reliable broadcast* which satisfies Uniform Integrity, Validity (Section 3.2) and:

- *Uniform Agreement* [NT90]: If *any* process delivers a message m, then all correct processes eventually deliver m.

A quiescent implementation of uniform reliable broadcast can be obtained using quiescent implementations of reliable broadcast, and of quasi reliable send and receive between every pair of processes. Roughly

[15]The No Loss and Quasi No Loss properties are very similar to the Strong Validity and Validity properties in Section 6 of [HT93].

speaking, when p wishes to uniform-broadcast m, it broadcasts m. Upon the delivery of m, each process r qr-sends an acknowledgement to every process, waits for the qr-receipt of such acknowledgements from $t+1$ processes, and then uniform-delivers m.

10 Using $\mathcal{HB}$ to Extend Previous Work

$\mathcal{HB}$ can be used to extend previous work in order to solve problems with algorithms that are both quiescent and tolerant of process crashes and messages losses.

10.1 Extending Existing Algorithms to Tolerate Link Failures

$\mathcal{HB}$ can be used to transform many existing algorithms that tolerate process crashes into quiescent algorithms that tolerate both process crashes and message losses. For example, consider the randomized consensus algorithms of [Ben83, CMS89, FM90, Rab83], the failure-detector based ones of [AT96, CT96], the probabilistic one of [BT85], and the algorithms for atomic broadcast in [CT96], k-set agreement in [Cha93], atomic commitment in [Gue95], and approximate agreement in [DLP$^+$86]. All these algorithms tolerate process crashes. Moreover, it is easy to verify that the only communication primitives that they actually need are quasi reliable send and receive, and/or reliable broadcast. Thus, in systems where $\mathcal{HB}$ is available, all these algorithms can be made to tolerate both process crashes and message losses (with fair links) by simply plugging in the quiescent communication primitives given in Section 7.[16] The resulting algorithms tolerate message losses and are quiescent.

10.2 Extending Results of [BCBT96]

Another way to solve problems with quiescent algorithms that tolerate both process crashes and message losses is obtained by extending the results of [BCBT96]. That work addresses the following question: given a problem that can be solved in a system where the only possible failures are process crashes, is the problem still solvable if links can also fail by losing messages? One of the models of lossy links considered in [BCBT96] is called *fair lossy*. Roughly speaking, a fair lossy link $p \to q$ satisfies the following property: If p sends an infinite number of messages to q and q is correct, then q receives an infinite number of messages from p (see Section 11.3 for a brief comparison between fair lossy and fair links).

[BCBT96] establishes the following result: any problem P that can be solved in systems with process crashes can also be solved in systems with process crashes and fair lossy links, provided P is *correct-restricted*[17] or a majority of processes are correct. For each of these two cases, [BCBT96] shows how to transform any algorithm that solves P in a system with process crashes, into one that solves P in a system with process crashes and fair lossy links. The algorithms that result from these transformations, however, are not quiescent: each transformation requires processes to repeatedly send messages forever.

Given $\mathcal{HB}$, we can modify the transformations in [BCBT96] to ensure that if the original algorithm is quiescent then so is the transformed one. Roughly speaking, the modification consists of (1) adding message acknowledgements; (2) suppressing the sending of a message from p to q if either (a) p has received an acknowledgement for that message from q, or (b) the heartbeat of q has not increased since the last time p sent a message to q; and (3) modifying the meaning of the operation "append $Queue_1$ to $Queue_2$" so that only the elements in $Queue_1$ that are not in $Queue_2$ are actually appended to $Queue_2$. The results in [BCBT96], combined with the above modification, show that if a problem P can be solved with a quiescent

[16] This can also be done to algorithms that require reliable send/receive or uniform reliable broadcast by plugging in the implementations given in Section 9, provided a majority of processes are correct.

[17] Intuitively, a problem P is correct-restricted if its specification does not refer to the behavior of faulty processes [BN92, Gop92].

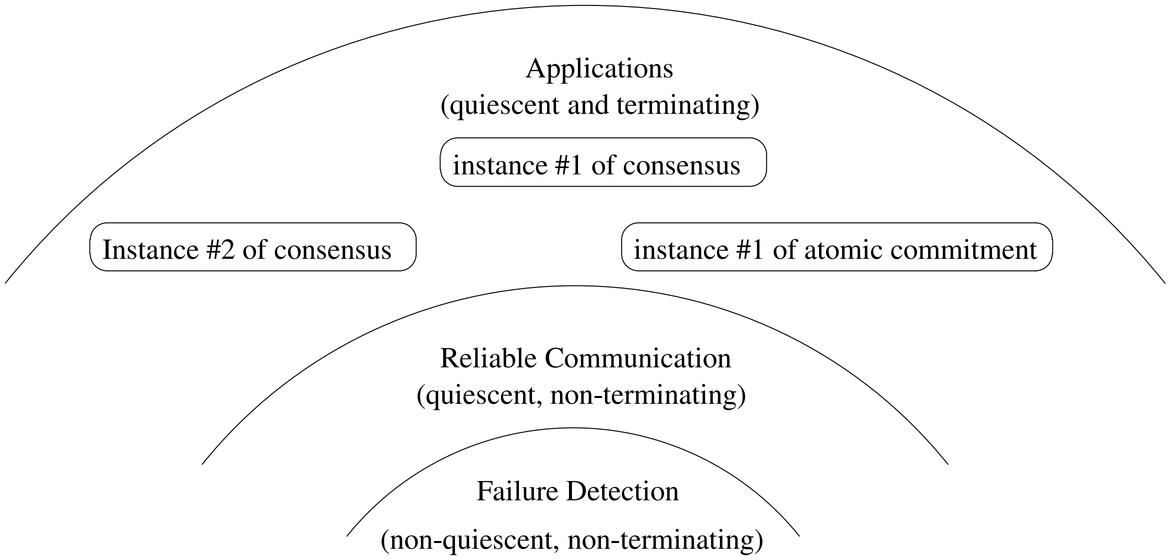

Figure 6: Layering that separates applications, reliable communication, and failure detection.

algorithm in a system with crash failures only, and either P is correct-restricted or a majority of processes are correct, then P is solvable with a quiescent algorithm that uses $\mathcal{HB}$ in a system with crash failures and fair lossy links.

11 Concluding Remarks

11.1 About Message Buffering

We now address the issue of message buffering in the implementation of quasi reliable send and receive, and of reliable broadcast (Section 7). 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. 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,[18] but the heartbeat mechanism ensures that processes stop sending messages to p soon after p actually crashes, and much before its removal.

11.2 Quiescence versus Termination

In this paper, we considered reliable communication protocols that tolerate process crashes and message losses, and focused on achieving quiescence. What about achieving termination? A *terminating* protocol guarantees that every process eventually reaches a halting state from which it cannot take further actions. A terminating protocol is obviously quiescent, but the converse is not necessarily true. For example, consider the protocol described at the beginning of Section 1. In this protocol, (a) s sends a copy of m repeatedly

[18]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.

until it receives $ack(m)$ from r, and then it halts; and (b) upon each receipt of m, r sends $ack(m)$ back to s. In the absence of process crashes this protocol is quiescent. However, the protocol is not terminating because r never halts: r remains (forever) ready to reply to the receipt of a possible message from s.

Can we use $\mathcal{HB}$ to obtain reliable communication protocols that are *terminating*? The answer is no, *even for systems with no process crashes*, as we now explain. Consider a system with message losses (fair links) and no process crashes. [KT88] proves that for any terminating protocol P, and any initial configuration of P, there are runs of P in which all processes halt without receiving any message. This implies that a terminating protocol cannot solve the reliable communication problem (in systems with fair links).

To deal with this problem, we propose a layering that allows *applications* to terminate. This layering, shown in Fig. 6, separates applications, reliable communication, and failure detection. At the lowest level, there are failure detectors, such as $\mathcal{HB}$. Of course, these are neither quiescent nor terminating. At the middle level, there are reliable communication protocols, such as those that we described in Sections 3 and 9. These communication protocols are quiescent (thanks to the failure detectors at the lower level) but not terminating. Finally, at the top level, there are applications, such as concurrent instances of consensus, atomic broadcast, atomic commitment protocols, etc. Applications are both quiescent *and* terminating: they achieve termination thanks to the reliable communication layer. For example, consider an instance of consensus. Once a process decides, it delegates the task of broadcasting the decision value to the reliable communication layer, and then it terminates (without waiting for the broadcast to terminate). Since every correct process eventually decides and terminates, this instance of consensus terminates.

If necessary, termination in the reliable communication layer can also be achieved in practice, as we now explain. A reliable communication protocol is unable to terminate when processes cannot determine whether a non-responsive process has crashed or it is only very slow. However, as we mentioned in our discussion of message buffering, a process that actually crashes is eventually removed by the operating system or a group membership protocol (and the remaining processes are notified accordingly). When this happens, the communication protocol can terminate. Note that with the heartbeat mechanism quiescence can be achieved long before termination (this is because when a process crashes, it may take a relatively long time to decide that it actually crashed, but its heartbeat count at other processes stops increasing almost immediately).

As a final remark, we note that some communication protocols, such as standard *data link* protocols, are inherently non-terminating: they are shared communication services that are always "ready" for message transmission. The reliable communication protocols (in our middle level) could also be viewed in the same way, namely, as non-terminating shared services that are always ready for message transmission.

11.3 Fair Links versus Fair Lossy Links

Fair links and fair lossy links are two typical models of lossy links considered in the literature.[19] Roughly speaking, a fair link guarantees that for every m, if p sends m to q an infinite number of times, and q is correct, then q receives m an infinite number of times. On the other hand, a fair lossy link guarantees that if p sends an infinite number of messages to q, and q is correct, then q receives an infinite number of messages from p. Fair lossy links and fair links differ in a subtle way. For instance, if process p sends the infinite sequence of distinct messages $m_1, m_2, m_3, \ldots$ to q and $p \rightarrow q$ is fair lossy, then q is guaranteed to receive an infinite subsequence, whereas if $p \rightarrow q$ is fair, q may receive nothing (because each distinct message is sent only once). On the other hand, if p sends the infinite sequence $m_1, m_2, m_1, m_2, \ldots$ and $p \rightarrow q$ is fair lossy, q may never receive a copy of m_2 (while it receives m_1 infinitely often), whereas if $p \rightarrow q$ is fair, q is guaranteed to receive an infinite number of copies of both m_1 and m_2.

In this paper, we chose the fair links model. A natural question is whether our results still hold with fair lossy links instead. It turns out that the answer is yes, as we now explain. First note that Theorems 1,

[19] In [Lyn96], these links correspond to the Strong and Weak Loss Limitation properties, respectively.

7 and 32 still hold because their proofs rely only on the fact that lossy links may lose (or not lose) messages arbitrarily during any finite period of time — a behavior allowed by fair lossy links. Moreover, the algorithms in Sections 7 and 8 can be easily modified to work with fair lossy links through the use of piggybacking; namely, every time a process wishes to send a message, it piggybacks all the messages that it previously sent.[20] Finally, the algorithms in Section 9 are still correct because they do not directly use the communication links; rather, they rely only on the communication algorithms of Section 7.

11.4 Quiescent versus Non-Quiescent Transformations

We proved that if $\mathcal{D}$ is a failure detector with finite range that can be used to solve quiescent reliable communication, then $\mathcal{D}$ can be transformed to $\diamond\mathcal{P}$. Our transformation is not quiescent: to "extract" $\diamond\mathcal{P}$ out of $\mathcal{D}$, processes keep on sending messages forever. This, however, does not invalidate the two facts that we wanted to show, namely:

1. $\mathcal{D}$ encodes at least as much information as $\diamond\mathcal{P}$.
2. $\mathcal{D}$ *cannot* be implemented (this follows from the transformation from $\mathcal{D}$ to $\diamond\mathcal{P}$, and the fact that $\diamond\mathcal{P}$ cannot be implemented).

This shows that finite-range failure detectors have some inherent limitation (because there is a failure detector with infinite range, namely $\mathcal{HB}$, that can be used to solve quiescent reliable communication such that (1) $\mathcal{HB}$ does not encode $\diamond\mathcal{P}$ and (2) $\mathcal{HB}$ can be implemented).

Even though a non-quiescent transformation was sufficient to establish our results, quiescent transformations are necessary when comparing the power of failure detectors to solve tasks with quiescent algorithms, as we now explain. If $\mathcal{D}$ can be transformed to $\mathcal{D}'$, we can conclude that $\mathcal{D}$ is (at least) as powerful as $\mathcal{D}'$ in terms of task solving (intuively, a *task* is a relation between inputs and outputs [BMZ88, HS93]). If the transformation from $\mathcal{D}$ to $\mathcal{D}'$ is not quiescent, however, $\mathcal{D}$ may not be as powerful as $\mathcal{D}'$ in terms of solving tasks *quiescently*: there may be a task that can be solved quiescently with $\mathcal{D}'$ but not with $\mathcal{D}$. On the other hand, if the transformation from $\mathcal{D}$ to $\mathcal{D}'$ *is* quiescent, we can conclude that $\mathcal{D}$ is (at least) as powerful as $\mathcal{D}'$ in terms of solving tasks with quiescent algorithms. The study of quiescent transformations is a new and interesting subject of research.

11.5 Extension to Partitionable Networks

In this paper, we considered networks that do not partition: we assumed that every pair of correct processes are reachable from each other through fair paths. In a subsequent paper [ACT], we drop this assumption and consider *partitionable networks*. We first generalize the definition of $\mathcal{HB}$ and show how to implement it in such networks. We then consider generalized versions of reliable communication and of consensus for partitionable networks, and use $\mathcal{HB}$ to solve these problems with quiescent protocols (to solve consensus we also use a generalization of the *Eventually Strong* failure detector [CT96]).

Acknowledgments

We are grateful to Anindya Basu and Bernadette Charron-Bost for having provided extensive comments that improved the presentation of this paper. We would also like to thank Tushar Deepak Chandra for suggesting the name Heartbeat, and for providing insightful comments regarding the simplifying assumption

[20] With the fair links used in this paper, this expensive piggybacking is avoided. We believe that in practice, links that intermittently lose messages are *both* fair and fair lossy.

of Theorem 7 which helped us write the Appendix. We are also grateful to the anonymous referees for their helpful comments.

References

[ACT] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Using the heartbeat failure detector for quiescent reliable communication and consensus in partitionable networks. Submitted to the journal *Theoretical Computer Science*.

[AT96] Marcos Kawazoe Aguilera and Sam Toueg. Randomization and failure detection: a hybrid approach to solve consensus. In *Proceedings of the 10th International Workshop on Distributed Algorithms*, Lecture Notes on Computer Science, pages 29–39. Springer-Verlag, October 1996.

[BCBT96] Anindya Basu, Bernadette Charron-Bost, and Sam Toueg. Simulating reliable links with unreliable links in the presence of process crashes. In *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.

[Ben83] Michael Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. In *Proceedings of the 2nd ACM Symposium on Principles of Distributed Computing*, pages 27–30, August 1983.

[BMZ88] Ofer Biran, Shlomo Moran, and Shmuel Zaks. A combinatorial characterization of the distributed tasks that are solvable in the presence of one faulty processor. In *Proceedings of the 7th ACM Symposium on Principles of Distributed Computing*, pages 263–275, August 1988.

[BN92] R. Bazzi and G. Neiger. Simulating crash failures with many faulty processors. In A. Segal and S. Zaks, editors, *Proceedings of the 6th International Workshop on Distributed Algorithms*, volume 647 of *Lecture Notes on Computer Science*, pages 166–184. Springer-Verlag, 1992.

[BT85] Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM*, 32(4):824–840, October 1985.

[BT93] Özalp Babaoğlu and Sam Toueg. Non-blocking atomic commitment. In Sape J. Mullender, editor, *Distributed Systems*, chapter 6. Addison-Wesley, 1993.

[Cha93] Soma Chaudhuri. More *choices* allow more *faults*: Set consensus problems in totally asynchronous systems. *Information and Computation*, 105(1):132–158, July 1993.

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

[CHT96] 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.

[CMS89] Benny Chor, Michael Merritt, and David B. Shmoys. Simple constant-time consensus protocols in realistic failure models. *Journal of the ACM*, 36(3):591–614, 1989.

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

[DLP+86] Danny Dolev, Nancy A. Lynch, Shlomit S. Pinter, Eugene W. Stark, and William E. Weihl. Reaching approximate agreement in the presence of faults. *Journal of the ACM*, 33(3):499–516, July 1986.

[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.

[FM90] Paul Feldman and Silvio Micali. An optimal algorithm for synchronous Byzantine agreement. Technical Report MIT/LCS/TM-425, Laboratory for Computer Science, Massachusetts Institute of Technology, June 1990.

[GLS95] Rachid Guerraoui, Michael Larrea, and André Schiper. Non blocking atomic commitment with an unreliable failure detector. In *Proceedings of the 14th IEEE Symposium on Reliable Distributed Systems*, pages 41–50, 1995.

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

[Gra78] James N. Gray. Notes on database operating systems. In R. Bayer, R. M. Graham, and G. Seegmuller, editors, *Operating Systems: An Advanced Course*, volume 66 of *Lecture Notes on Computer Science*. Springer-Verlag, 1978. Also appears as IBM Research Laboratory Technical report RJ2188.

[Gue95] Rachid Guerraoui. Revisiting the relationship between non-blocking atomic commitment and consensus. In *Proceedings of the 9th International Workshop on Distributed Algorithms*, volume 972 of *Lecture Notes on Computer Science*, pages 87–100, Le Mont-St-Michel, France, 1995. Springer-Verlag.

[Had86] Vassos Hadzilacos. On the relationship between the atomic commitment and consensus problems. In *Proceedings of the Workshop on Fault-Tolerant Distributed Computing*, volume 448 of *Lecture Notes on Computer Science*, pages 201–208. Springer-Verlag, March 1986.

[HMR97] Michel Hurfin, Achour Mostefaoui, and Michel Raynal. Consensus in asynchronous systems where processes can crash and recover. Technical Report 1144, Institut de Recherche en Informatique et Systèmes Aléatoires, Université de Rennes, November 1997.

[HS93] Maurice Herlihy and Nir Shavit. The asynchronous computability theorem for t-resilient tasks. In *Proceedings of the 25th ACM Symposium on Theory of Computing*, pages 111–120. ACM Press, May 1993.

[HT93] Vassos Hadzilacos and Sam Toueg. Fault-tolerant broadcasts and related problems. In Sape J. Mullender, editor, *Distributed Systems*, chapter 5, pages 97–145. Addison-Wesley, 1993. A revised and extended version appears in [HT94].

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

[KT88] Richard Koo and Sam Toueg. Effects of message loss on the termination of distributed protocols. *Information Processing Letters*, 27(4):181–188, April 1988.

[LH94] Wai-Kau Lo and Vassos Hadzilacos. Using failure detectors to solve consensus in asynchronous shared-memory systems. In *Proceedings of the 8th International Workshop on Distributed Algorithms*, pages 280–295, Terschelling, The Netherlands, 1994.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc., 1996.

[MR89] Yoram Moses and Gil Roth. On reliable message diffusion. In *Proceedings of the 8th ACM Symposium on Principles of Distributed Computing*, pages 119–128, August 1989.

[NT90] Gil Neiger and Sam Toueg. Automatically increasing the fault-tolerance of distributed algorithms. *Journal of Algorithms*, 11(3):374–419, 1990.

[OGS97] Rui Oliveira, Rachid Guerraoui, and André Schiper. Consensus in the crash-recover model. Technical Report 97-239, Département d'Informatique, Ecole Polytechnique Fédérale, Lausanne, Switzerland, August 1997.

[Rab83] Michael Rabin. Randomized Byzantine generals. In *Proceedings of the 24th Symposium on Foundations of Computer Science*, pages 403–409. IEEE Computer Society Press, November 1983.

[SM95] Laura S. Sabel and Keith Marzullo. Election vs. consensus in asynchronous systems. Technical Report 95-1488, Department of Computer Science, Cornell University, Ithaca, New York, Febrary 1995.

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

Appendix

A Removing the Simplifying Assumption from Theorem 7

We now give an extended, more complex proof of Theorem 7 without the simplifying assumption.

Let $\mathcal{E}$ be an environment and $\mathcal{D}$ be any failure detector with finite range $\mathcal{R} = \{v_1, v_2, \ldots, v_\ell\}$. Let $\mathcal{I}$ be a quiescent implementation of s-send and s-receive that uses $\mathcal{D}$ in environment $\mathcal{E}$.

As in the simpler proof in Section 5, the transformation algorithm $T_{\mathcal{D}\rightarrow\diamond\mathcal{P}}$ uses a finite table that is predetermined from $\mathcal{D}$. We first define this table and show some of its properties (Section A.1). We then describe and prove the correctness of the transformation algorithm $T_{\mathcal{D}\rightarrow\diamond\mathcal{P}}$ that uses this table (Section A.2).

A.1 The Predetermined Table

For the definitions in this proof, let:

- v_j be a failure detector value, i.e., $v_j \in \mathcal{R}$
- p be a process, i.e., $p \in \Pi$
- F be a failure pattern
- H be a failure detector history with range $\mathcal{R}$
- f be an assignment of failure detector values to every process in Π, i.e., $f : \Pi \longrightarrow \mathcal{R}$
- P and P_0 be non-empty set of processes
- $p_0, p_1, \ldots, p_{m-1}$ be the processes in P (where $m = |P|$ and $p_0 < p_1 < \ldots < p_{m-1}$)

Definition 1 We say that v_j is *a limit value for* p *and* H if, for infinitely many t, $H(p, t) = v_j$. □

Definition 2 We say that f *is a limit vector for* P *and* H if for all $p \in P$, $f(p)$ is a limit value for p and H. The set of all limit vectors for P and H is denoted $L_P(H)$. □

Definition 3 $\mathit{RRIRounds}(P, f)$ is defined as follows.

Consider the round-robin execution of implementation $\mathcal{I}$ in which: (a) processes in P take steps forever in a round-robin fashion[21] and processes in $\Pi \setminus P$ do not take any steps, (b) no process ever s-sends any bit, (c) every time a process $p \in P$ queries its failure detector module, p gets $f(p)$, (d) every time a process $p \in P$ takes a step, p receives the earliest message sent to it that it did not yet receive (thus, every $p \in P$ eventually receives each message sent to it), and (e) all messages sent to processes in $\Pi \setminus P$ are lost.[22]

There are two possible cases in the above round-robin execution of $\mathcal{I}$:

- *Every process eventually stops sending messages*. In this case, after some number k of round-robin rounds, no process ever receives any messages. We say that "round-robin initialization (r.r.i.) occurs in k rounds", and define $\mathit{RRIRounds}(P, f) = k$.

- *Some process never stops sending messages*. In this case, we define $\mathit{RRIRounds}(P, f) = \infty$. □

Intuitively, we say that F and H allow round-robin initialization for P and f if the following holds: (a) in the above execution with P and f, r.r.i. occurs in k rounds for some k, and (b) there is a schedule compatible with F and H that allows this k-round r.r.i. More precisely:

[21] That is, p_0 takes the first step, then p_1 takes a step, and so on, so that the j-th step is taken by process $p_{(j-1) \bmod m}$.

[22] It is possible that this is *not* a valid execution of $\mathcal{I}$ using $\mathcal{D}$ in environment $\mathcal{E}$.

Definition 4 We say that F *and* H *allow round-robin initialization (r.r.i.) for* P *and* f if (a) $RRIRounds(P, f) = k$ for some k, and (b) if there are times $t_0 < t_1 < \ldots < t_{mk-1}$ such that for every $0 \le j \le mk - 1$, (1) $p_{j \bmod m}$ is not crashed at time t_j, i.e., $p_{j \bmod m} \notin F(t_j)$ and (2) the failure detector module of $p_{j \bmod m}$ at time t_j outputs $f(p_{j \bmod m})$, i.e., $H(p_{j \bmod m}, t_j) = f(p_{j \bmod m})$. □

Definition 5 $L_{P,P_0}(F, H) = \{f \mid f \in L_P(H)$ and F and H allow r.r.i. for P_0 and $f\}$. □

Definition 6 $E^{\mathcal{D},\mathcal{E}}_{P,P_0} = \{f \mid \exists F \in \mathcal{E}, \exists H \in \mathcal{D}(F) : P = correct\_proc(F) \text{ and } f \in L_{P,P_0}(F, H)\}$. □

Roughly speaking, $E^{\mathcal{D},\mathcal{E}}_{P,P_0}$ is the set of limit vectors f that could occur when P is the set of correct processes and it is possible to have r.r.i. for P_0 and f.

The table used by the transformation algorithm $T_{\mathcal{D}\to\diamond\mathcal{P}}$ consists of all the sets $E^{\mathcal{D},\mathcal{E}}_{P,P_0}$ where P and P_0 range over all non-empty subset of processes. Note that this table is finite. We omit the superscript $\mathcal{D}, \mathcal{E}$ from $E^{\mathcal{D},\mathcal{E}}_{P,P_0}$ whenever it is clear from the context.

Lemma 27 *Let* $F \in \mathcal{E}$*,* $P = correct\_proc(F)$*,* $H \in \mathcal{D}(F)$ *and* $f \in L_P(H)$*. Assume* $P \neq \emptyset$*. Then* $RRIRounds(P, f) < \infty$.

Proof. We can construct a run R of implementation $\mathcal{I}$ using $\mathcal{D}$ with $F \in \mathcal{E}$, such that all processes behave exactly as in the round-robin execution of $\mathcal{I}$ that was used to define $RRIRounds(P, f)$. To see this, note that since $F \in \mathcal{E}$, $P = correct\_proc(F)$, $H \in \mathcal{D}(F)$ and $f \in L_P(H)$, we can find times for the round-robin steps of correct processes such that, for each time u at which a process p takes a step, the output $H(p, u)$ of its failure detector module is $f(p)$. Since $\mathcal{I}$ is quiescent, there is a time after which no process sends any message in run R. Thus, $RRIRounds(P, f) < \infty$. □

Lemma 28 *Let* $F \in \mathcal{E}$*,* $P = correct\_proc(F)$*,* $H \in \mathcal{D}(F)$*. Assume* $P \neq \emptyset$ *and let* P_0 *be such that* $P \subseteq P_0 \subseteq \Pi$*. If* $f \in L_{P,P_0}(F, H)$ *then* $f \in E_{P,P_0}$ *and* $f \notin E_{P',P_0}$ *for all* P' *such that* $\emptyset \subset P' \subset P$.

Proof. Let $f \in L_{P,P_0}(F, H)$. The fact that $f \in E_{P,P_0}$ is immediate from the definition of E_{P,P_0}. Let P' be such that $\emptyset \subset P' \subset P$. Suppose, for contradiction, that $f \in E_{P',P_0}$. Then there exists a failure pattern $F' \in \mathcal{E}$ and $H' \in \mathcal{D}(F')$ such that $P' = correct\_proc(F')$ and $f \in L_{P',P_0}(F', H')$.

We now obtain a contradiction by using the quiescent implementation $\mathcal{I}$. Let p be a process in P' and q be a process in $P \setminus P'$. We construct three runs of $\mathcal{I}$, namely, R_0, R_1 and R_2. Roughly speaking, each one of these runs starts with an r.r.i. for P_0 and f. After this initialization, in R_0 nothing else happens, in R_1 process p s-sends some bit to q but q crashes, and in R_2 process p s-sends the same bit to q and q is correct. We will reach a contradiction by arguing that in R_2 process q behaves as in R_0, and thus it never s-receives any bit from p — this violates the defining property of s-send and s-receive.

Runs R_0, R_1 and R_2 are defined as follows:[23]

- Run R_0 has failure pattern F and failure detector history H. Since $f \in L_{P,P_0}(F, H)$, $f \in L_P(H)$, and F and H allow r.r.i. for P_0 and f. R_0 consists initially of a r.r.i. for P_0 and f. More precisely, initially: (a) processes in P_0 take steps in a round-robin fashion and processes in $\Pi \setminus P_0$ do not take any steps, (b) no process s-sends any bit, (c) every time a process $r \in P_0$ queries its failure detector module, r gets $f(r)$, (d) every time a process $r \in P_0$ takes a step, r receives the earliest message sent to it that it did not yet receive, and (e) all messages sent to processes in $\Pi \setminus P_0$ are lost. This goes on

[23] In each one of these runs, we will require that for a certain *finite* period of time, some messages are lost while others are not. As we explained in our model (Section 2.5), this behavior is consistent with any link failure pattern.

until each process in P_0 has taken $RRIRounds(P_0, f)$ steps. Let t_0 be the time when this happens. After t_0, processes in P take steps in a round-robin fashion such that every time a process $r \in P$ takes a step, it obtains $f(r)$ from its failure detector module (this is possible because $f \in L_P(H)$); moreover, no process s-sends any bit.

Note that since both p and q are in $P = correct\_proc(F)$, and p does not s-send any bit to q, it must be that q does not s-receive any bit from p. Furthermore, after time t_0, no processes send or receive any messages.

- Run R_1 has failure pattern F' and failure detector history H'. Since $f \in L_{P',P_0}(F', H')$, $f \in L_{P'}(H')$, and F' and H' allow r.r.i. for P_0 and f. Initially, processes in R_1 behave as in R_0, i.e., R_1 starts with a r.r.i. for P_0 and f. Then, execution proceeds as follows: (a) p s-sends some bit b to q, (b) processes in P' take steps in round-robin fashion and processes in $\Pi \setminus P'$ take no steps, (c) every time a process $r \in P'$ takes a step, it obtains $f(r)$ from its failure detector module, (d) every time a process $r \in P'$ takes a step, r receives the earliest message sent to it that it did not yet receive, (e) all messages sent to processes in $\Pi \setminus P'$ are lost.

  Note that, since implementation $\mathcal{I}$ is quiescent, there is a time t_1 after which no messages are sent or received. Assume without loss of generality that at time t_1 every process in P' took the same number k of steps (otherwise, choose another time $t'_1 > t_1$).

- Run R_2 has failure pattern F and failure detector history H. Initially, processes in R_2 behave as in R_1: R_2 starts with a r.r.i. for P_0 and f, and then p s-sends b to q and execution continues as in R_1, until each process in P' has taken k steps (this is possible because $f \in L_P(H)$ and $P' \subseteq P$).

  Let t_2 be the time when this happens. After t_2, execution proceeds as follows: (a) no process s-sends any bit, (b) processes in P take steps in round-robin fashion and processes in $\Pi \setminus P$ take no steps, (c) every time a process $r \in P$ takes a step, it obtains $f(r)$ from its failure detector module (this is possible because $f \in L_P(H)$).

  Note that at time t_2, each process in P' is in the same state as in run R_1 at time t_1, and each process in $P \setminus P'$ is in the same state as in run R_0 at time t_0. A simple induction on the steps taken shows that, in R_2, (1) processes in P' have the same behavior as in run R_1; (2) processes in $P \setminus P'$ have the same behavior as in run R_0; (3) no messages are sent or received after time t_2. Since $q \in P \setminus P'$ and q does not s-receive any bit from p in R_0, it does not s-receive any bit from p in R_2.

In summary, in R_2: (a) both p and q are correct; (b) p s-sends b to q; and (c) q does not s-receive b from p. Thus, $\mathcal{I}$ is not a correct implementation of s-send and s-receive — a contradiction. □

A.2 The Transformation Algorithm

The algorithm $T_{\mathcal{D}\to\mathcal{D}'}$ that transforms $\mathcal{D}$ to an eventually perfect failure detector $\mathcal{D}' = \diamond\mathcal{P}$ in environment $\mathcal{E}$ is shown in Fig. 7. $T_{\mathcal{D}\to\mathcal{D}'}$ uses the table of sets E_{P,P_0} (for all non-empty subsets P and P_0 of processes) that has been determined *a priori* from the given $\mathcal{D}$ and $\mathcal{E}$. It also uses an implementation of qr-send and qr-receive between every pair of processes. A simple implementation is by repeated retransmissions and diffusion (it does not have to be quiescent).

All variables are local to each process. *Sequences* is a finite set of finite sequences of pairs (p, v) where $p \in \Pi$ is a process and $v \in \mathcal{R}$ is a failure detector value. It stores possible schedules that could have resulted from F and H. Vector f stores the last failure detector value that p qr-received from each process. *Order* is an ordered set that records the order in which the last failure detector value from each process was qr-received. $\mathcal{D}'_p$ denotes the output of the eventually perfect failure detector that p is simulating (a set of processes that p currently suspects). $AllowsRRI$ is a boolean function that takes three parameters: a set $Sequences$, a set $P = \{p_0, p_1, \ldots, p_{m-1}\} \subseteq \Pi$ (where $p_0 < p_1 < \ldots < p_{m-1}$), and a vector f. It returns true if and only if for some sequence $s \in Sequences$, there exists a subsequence of s that consists of $RRIRounds(P, f)$ repetitions of $(p_0, f(p_0)), (p_1, f(p_1)), \ldots, (p_{m-1}, f(p_{m-1}))$.

In Task 1, each process p periodically queries its failure detector module, appends a new pair to each sequence in $Sequences$ and then qr-sends *Sequences* and the output of its failure detector module $\mathcal{D}_p$ to every process. Upon the qr-receipt of $(Sequences', v')$ from process q in Task 2, process p enters v' into $f[q]$, moves q to the front of $Order$, and updates $Sequences$. Then, p uses the function $AllowsRRI$ to check whether there is some k such that r.r.i. could have occurred for $Order[1..k]$ and f. If there is, it sets k_0 to the *largest* such k, and then checks if for some k', $f \in E_{Order[1..k'], Order[1..k_0]}$. If so, it sets k_1 to the *smallest* such k', and sets $\mathcal{D}'$ to the complement of $Order[1..k_1]$.

For every process p:

    Initialization:
        for all $q \in \Pi$ **do** $f[q] \leftarrow \perp$
        $Order \leftarrow \emptyset$
        $Sequences \leftarrow \{\lambda\}$
        $\mathcal{D}'_p \leftarrow \emptyset$
        { For each $\emptyset \subset P, P_0 \subseteq \Pi$, the set $E^{\mathcal{D},\mathcal{E}}_{P,P_0}$ is determined *a priori* from $\mathcal{D}$ and $\mathcal{E}$ }

    cobegin
        || *Task 1*:
            repeat periodically
                $v \leftarrow \mathcal{D}_p$ {query $\mathcal{D}$}
                append (p, v) to each sequence in $Sequences$
                for all $q \in \Pi$ **do** qr-send $(Sequences, v)$ **to** q

        || *Task 2*:
            upon qr-receive $(Sequences', v')$ **from** q **do**
                $f[q] \leftarrow v'$
                $Order \leftarrow q \mid\mid (Order \setminus \{q\})$ {process q is moved to the front of *Order*}
                $Sequences \leftarrow Sequences \cup Sequences'$
                if for some $k \geq 1$, $AllowsRRI(Sequences, Order[1..k], f)$ **then**
                    let k_0 be the largest such k
                    if for some $k' \geq 1$, $f \in E^{\mathcal{D},\mathcal{E}}_{Order[1..k'], Order[1..k_0]}$ **then**
                        let k_1 be the smallest such k'
                        $\mathcal{D}'_p \leftarrow \Pi \setminus Order[1..k_1]$ {suspect processes not in $Order[1..k_1]$}
    coend

Figure 7: Transformation of $\mathcal{D}$ to an eventually perfect failure detector $\mathcal{D}'$

We now show that the failure detector constructed by this algorithm, namely $\mathcal{D}'$, is an eventually perfect failure detector. Consider a run of this algorithm with failure pattern $F \in \mathcal{E}$ and failure detector history $H \in \mathcal{D}(F)$, such that $correct\_proc(F) \neq \emptyset$. Let t be the number of processes that crash in F, i.e., $t = |\Pi \setminus correct\_proc(F)|$. Henceforth, p denotes a correct process in F, and f, $Order$, and $Sequences$ are variables local to p.

Lemma 29 *There is a time t_0 after which (1) $\mathit{Order}[1..n-t] = \mathit{correct\_proc}(F)$, (2) $f \in L_{\mathit{Order}[1..n-t]}(H)$ and (3) $\mathit{AllowsRRI}(\mathit{Sequences}, \mathit{Order}[1..n-t], f)$.*[24]

Proof. Note that p eventually stops qr-receiving messages from processes that crash, and p never stops qr-receiving messages from correct processes. From the way Order is updated, there is a time t_1 after which (1) holds.

Let $P = \mathit{correct\_proc}(F)$. Variable f ranges over a finite number of values, so there are functions $f_1, f_2, \ldots, f_N : \Pi \to \mathcal{R}$ such that (a) for every $1 \leq j \leq N$, variable f is equal to f_j an infinite number of times, and (b) there is a time t_2 after which the predicate $f \in \{f_1, f_2, \ldots, f_N\}$ holds. We now show that for every $1 \leq j \leq N$, $f_j \in L_P(H)$, and there is a time τ_j after which $\mathit{AllowsRRI}(\mathit{Sequences}, P, f_j)$ holds. Together with (1) and (b), this implies that after time $t_0 = \max\{t_1, t_2, \tau_1, \tau_2, \ldots, \tau_N\}$, both (2) and (3) hold.

Let $1 \leq j \leq N$. We first claim that each process $q \in P$ obtains $f_j(q)$ from $\mathcal{D}$ in line 13 an infinite number of times — this immediately implies $f_j \in L_P(H)$. To show the claim, note that process p qr-receives a message from q and updates $f[q]$ an infinite number of times. Together with (a), this implies that p qr-receives a message containing $f_j(q)$ from q an infinite number of times, and this implies the claim.

We now show that there is a time τ_j after which $\mathit{AllowsRRI}(\mathit{Sequences}, P, f_j)$ holds. Since $f_j \in L_P(H)$, by Lemma 27, $\mathit{RRIRounds}(P, f_j) = k$ for some $k < \infty$. Let $p_0 < p_1 < \ldots < p_{m-1}$ be the processes in P. By the claim, at some time u_0, p_0 obtains $f_j(p_0)$ from $\mathcal{D}$ in line 13. After doing so, p_0 appends $(p_0, f_j(p_0))$ to all sequences in $\mathit{Sequences}$ and qr-sends a message containing $\mathit{Sequences}$ to all processes. At some time $u'_1 > u_0$, p_1 qr-receives this message and updates $\mathit{Sequences}$. By the claim, at some time $u_1 > u'_1$, p_1 obtains $f_j(p_1)$ from $\mathcal{D}$ in line 13. After doing so, p_1 appends $(p_1, f_j(p_1))$ to all sequences in $\mathit{Sequences}$ and so p_1 obtains a sequence containing $(p_0, f_j(p_0))$ before $(p_1, f_j(p_1))$. We can repeat this argument for all the processes in P in a round-robin order, for $k+1$ rounds, and conclude that eventually $\mathit{AllowsRRI}(\mathit{Sequences}, P, f_j)$ holds. □

Lemma 30 *There is a time t_1 after which for every $m_0 \geq n-t$ such that $\mathit{AllowsRRI}(\mathit{Sequences}, \mathit{Order}[1..m_0], f)$ holds: (1) $f \in E_{\mathit{Order}[1..n-t], \mathit{Order}[1..m_0]}$ and (2) for all $1 \leq m_1 < n-t$, $f \notin E_{\mathit{Order}[1..m_1], \mathit{Order}[1..m_0]}$.*

Proof. By Lemma 29, there is a time t_0 after which (a) $\mathit{Order}[1..n-t] = \mathit{correct\_proc}(F)$, and (b) $f \in L_{\mathit{Order}[1..n-t]}(H)$. Let $t_1 = t_0$. Suppose that at some time $t'_1 > t_1$, $\mathit{AllowsRRI}(\mathit{Sequences}, \mathit{Order}[1..m_0], f)$ holds for some $m_0 \geq n-t$. This implies that F and H allow r.r.i. for $\mathit{Order}[1..m_0]$ and f. From (b), $f \in L_{\mathit{Order}[1..n-t], \mathit{Order}[1..m_0]}(F, H)$ holds at time t'_1. By Lemma 28, $f \in E_{\mathit{Order}[1..n-t], \mathit{Order}[1..m_0]}$.

Let $1 \leq m_1 < n-t$. By (a), $\emptyset \subset \mathit{Order}[1..m_1] \subset \mathit{correct\_proc}(F) \subseteq \mathit{Order}[1..m_0]$ holds at time t'_1. Note that $f \in L_{\mathit{Order}[1..n-t], \mathit{Order}[1..m_0]}(F, H)$ holds at time t'_1. By Lemma 28, $f \notin E_{\mathit{Order}[1..m_1], \mathit{Order}[1..m_0]}$. □

Corollary 31 *There is a time after which $\mathcal{D}'_p = \Pi \setminus \mathit{correct\_proc}(F)$.*

Proof. By Lemma 29 part (3), there is a time t_0 after which every time p qr-receives some message, the **if** in line 22 evaluates to true and the k_0 selected in line 23 is at least $n-t$. After time t_0, by Lemma 30, there is a time after which: every time p qr-receives some message, the **if** in line 24 evaluates to true and the k_1 selected in line 25 is $n-t$. Now apply Lemma 29 part (1). □

By Corollary 31, we have:

[24] This does not mean that eventually the values of variables f, $\mathit{Sequences}$, and Order at p stop changing. It means that, although they may continue to change forever, eventually the predicates (1), (2) and (3) are true forever at p.

Theorem 32 *Consider an asynchronous system subject to process crashes and message losses. Suppose failure detector $\mathcal{D}$ with finite range can be used to solve the Single-Shot Reliable Send and Receive problem in environment $\mathcal{E}$, and that the implementation is quiescent. Then $\mathcal{D}$ can be transformed (in environment $\mathcal{E}$) to the eventually perfect failure detector $\Diamond\mathcal{P}$.*