Heartbeat: A Timeout-Free Failure Detector for 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

May 30, 1997

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 without failure detectors. We then show how to solve it using a new failure detector, called *heartbeat*. In contrast to previous failure detectors that have been used to circumvent impossibility results, the heartbeat failure detector *is* implementable, and its implementation does *not* use timeouts. These results have wide applicability: they can be used to transform many existing algorithms that tolerate only process crashes 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.

The heartbeat failure detector is novel: besides being implementable without timeouts, it does not output lists of suspects as typical failure detectors do. If we restrict failure detectors to output only lists of suspects, quiescent reliable communication requires $\diamond\mathcal{P}$ [ACT97a], which is not implementable. Combined with the results of this paper, this shows that traditional failure detectors that output only lists of suspects have fundamental limitations.

1 Motivation

This paper introduces *heartbeat*, a failure detector that can be implemented without timeouts, and shows how it can be used to solve the problem of *quiescent* reliable communication in asynchronous message-passing systems with process crashes and lossy links.

To illustrate this problem consider 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, by ARPA/ONR grant N00014-96-1-1014, and by an Olin Fellowship.

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, DFKM96, Gue95, LH94, 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 achieving quiescent applications and the use of a (non-quiescent) failure detection service as a tool to achieve this goal.

How can we use an unreliable failure detector to achieve quiescent reliable communication in the presence of process and link failures? Consider 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.

In [ACT97a], Aguilera *et al.* show that among all failure detectors that output lists of suspects, $\diamond\mathcal{P}$ is the *weakest* one that can be used to solve the above problem.[1] Unfortunately, $\diamond\mathcal{P}$ is not implementable in asynchronous systems with process crashes (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 this paper we show that this is not so.

[1] See [CHT96] for the concept of "weakest" for failure detectors.

2 The Heartbeat Failure Detector

We will show that quiescent reliable communication can be achieved with a failure detector that *can be implemented without timeouts* in systems with process crashes and lossy links. This 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 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.[2]

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

Thus, $\mathcal{HB}$ should not be confused with existing implementations of failure detectors (some of which, such as those in Ensemble and Phoenix, have modules that are also called *heartbeat* [vR97, Cha97]). Even though existing failure detectors are also based on the repeated sending of a heartbeat, *they use timeouts* on heartbeats in order to derive lists of processes considered to be up or down; applications can only see these lists. In contrast, $\mathcal{HB}$ simply counts heartbeats, and shows these counts to applications.

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 (which are actually quite short). 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.

$\mathcal{HB}$ can be used to solve the problem of quiescent reliable communication and it is implementable, but its counters are unbounded. Can we solve this problem using a failure detector that is both implementable and has bounded output? [ACT97a] proves that the answer is *no*: The weakest failure detector *with bounded output* that can be used to solve quiescent reliable communication is $\diamond P$.

Thus, the difference between $\mathcal{HB}$, whose output is unbounded, and existing failure detectors, whose output is bounded, is more than "skin deep". The results in this paper combined with those of [ACT97a], show that failure detectors with bounded output (including those that output lists of processes) are restricted in power and/or applicability.

3 Outline of the Results

We focus on two types of reliable communication mechanisms: *quasi reliable send/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 [HT94] 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.

We first show that there is no quiescent implementation of quasi reliable send/receive or of reliable broadcast in a network with process crashes and message losses. This holds even if we assume that links can lose only a finite number of messages.

We then show how to use failure detectors to circumvent the above impossibility result. We describe failure

[2]As we will see, however, in some types of networks the actual implementation is not entirely trivial.

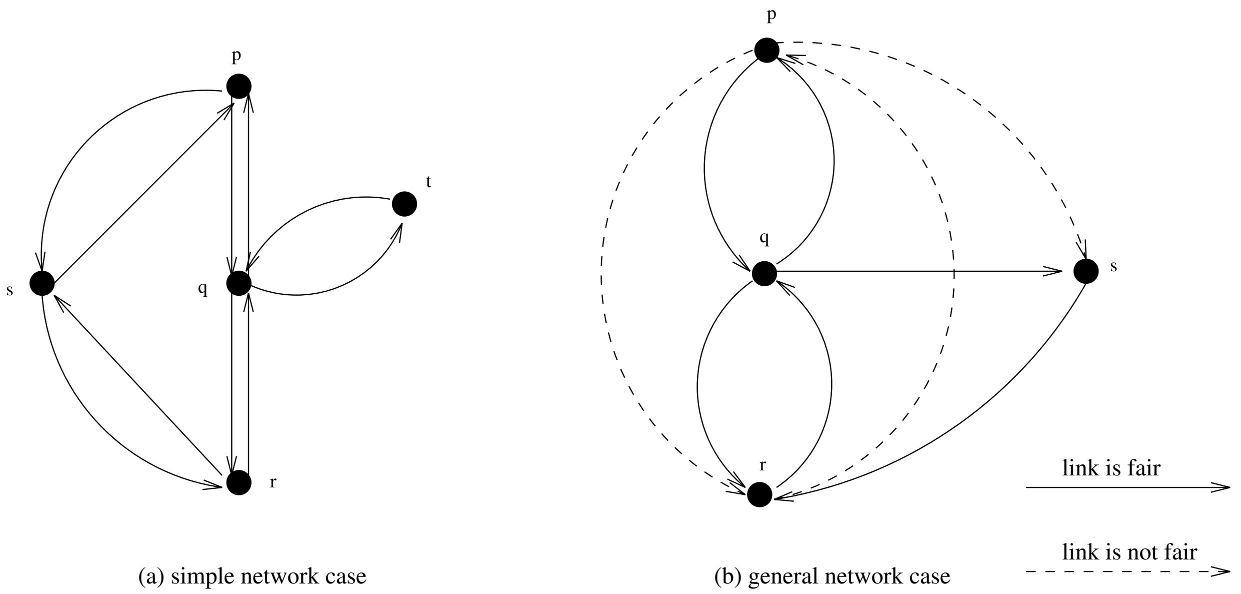

Figure 1: Examples of the simple and general network cases

detector $\mathcal{HB}$, and show that it is strong enough to achieve quiescent reliable communication, but weak enough to be implementable, in each one of the following two types of communication networks. In both types of networks, we assume that each correct process is connected to every other correct process through a *fair* path, i.e., a path containing only fair links and correct processes.[3] In the first type, all links are bidirectional and fair (Fig. 1a). In the second one, some links are unidirectional, and some links have no restrictions on message losses, i.e., they are not fair (Fig. 1b). Examples of such networks are networks that contain several unidirectional rings that intersect.

For each network type, we first describe quiescent protocols that use $\mathcal{HB}$ to solve quasi reliable send/receive and reliable broadcast, and then show how to implement $\mathcal{HB}$. For the first type of networks, a common one in practice, the implementation of $\mathcal{HB}$ and the reliable communication protocols are very simple and efficient. The algorithms for the second type are significantly more complex.

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

We then explain how $\mathcal{HB}$ can be used to easily 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 [Ben83, Rab83, BT85, CMS89, FM90, AT96, CT96], for atomic broadcast in [CT96], for k-set agreement in [Cha93], for atomic commitment in [Gue95], for approximate agreement in [DLP$^+$86], etc.

Finally, 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.[4]

To summarize, the main contributions of this paper are:

[3]This assumption precludes permanent network partitioning.

[4]The link failure model in [BCBT96] is slightly different from the one used here (cf. Section 11).

1. This is the first work that explores 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 describe a simple and *implementable* failure detector $\mathcal{HB}$ that can be used to solve this problem.

3. $\mathcal{HB}$ 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].

4. $\mathcal{HB}$ is novel: it is implementable without timeouts, and it does not output lists of suspects as typical failure detectors do [BDM97, CT96, Gue95, GLS95, LH94, SM95]. The results of this paper, combined with those in [ACT97a], show that lists of suspects is not always the best failure detector output.[5]

Reliable communication is a fundamental problem that 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 by focusing on the use of unreliable failure detectors to achieve quiescent reliable communication in the presence of process crashes and link failures. The work by Basu *et al.* in [BCBT96] is the closest to ours, but their protocols do not use failure detectors and are not quiescent. In Section 11, 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 4. Section 5 defines the reliable communication primitives that we focus on. In Section 6, we show that, without failure detectors, quiescent reliable communication is impossible. To overcome this problem, we define heartbeat failure detectors in Section 7, we show how to use them to achieve quiescent reliable communication in Section 8, and show how to implement them in Section 9. In Section 10, we consider two stronger types of communication primitives. In Section 11, we explain how to use heartbeat failure detectors to extend several previous results. In Section 12, we mention a generalization of our results for the case where the network may partition. A brief discussion of protocol quiescence versus protocol termination concludes the paper.

4 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 the network. 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.

To simplify the presentation of our model, we assume the existence of a discrete global clock. This is merely a fictional device: the 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.

4.1 Processes and Process Failures

The system consists of a set of n processes, $\Pi = \{1, \ldots, n\}$. Processes can fail by *crashing*, i.e., by prematurely halting. A *failure pattern* F is a function from $\mathcal{T}$ to 2^{Π}, where $F(t)$ denotes the set of processes that have crashed through time t. Once a process crashes, it does not "recover", i.e., $\forall t : F(t) \subseteq F(t+1)$. We define

[5] The authors of [CHT96] anticipated this possibility: they put no restrictions on the output of unreliable failure detectors when they determine the weakest one necessary to solve consensus.

$crashed(F) = \bigcup_{t \in \mathcal{T}} F(t)$ and $correct(F) = \Pi - crashed(F)$. If $p \in crashed(F)$ we say p *crashes (or is faulty) in* F and if $p \in correct(F)$ we say p is *correct in* F.

4.2 Links and Link Failures

Some pairs of processes in the network are connected through unidirectional 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 $neighbor(p)$.

With every link $p \to q$ we associate two primitives: $\mathsf{send}_{p,q}(m)$ and $\mathsf{receive}_{q,p}(m)$. We say that process p *sends message* m *to process* q if p invokes $\mathsf{send}_{p,q}(m)$. We assume that if p is correct, it eventually returns from this invocation. We allow process p to send the same message m more than once through the same link. We say that process q *receives message* m *from process* p if q returns from the execution of $\mathsf{receive}_{q,p}(m)$. We describe a link $p \to q$ by the properties that its $\mathsf{send}_{p,q}$ and $\mathsf{receive}_{q,p}$ primitives satisfy. We assume that links do not create messages, i.e., every link $p \to q$ in the network satisfies:

- *Integrity*: For all $k \geq 1$, if q receives m from p k times, then p previously sent m to q at least k times.

A lossy link can fail by dropping messages. A link $p \to q$ is *fair* if $\mathsf{send}_{p,q}$ and $\mathsf{receive}_{q,p}$ satisfy Integrity and:

- *Fairness*: If q is correct and p sends m to q an infinite number of times, then q receives m from p an infinite number of times.

4.3 Network Connectivity

A path $(p_1, \ldots, p_k)$ is *fair* if processes $p_1, \ldots, p_k$ are correct and links $p_1 \to p_2, \ldots, p_{k-1} \to p_k$ are fair. We assume that every pair of distinct correct processes is connected through a fair path. Without loss of generality, we can assume that this path is *simple* (i.e., no process appears twice in that path).

4.4 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 process can query its local failure detector module at any time. 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 failure detector outputs of $\mathcal{D}$). $\mathcal{D}(F)$ denotes the set of possible failure detector histories permitted by $\mathcal{D}$ for the failure pattern F. We stress that the output of a failure detector depends *only* on the failure pattern F; it cannot depend on the behavior of applications. This means that failure detectors can neither obtain feedback from applications nor be used by applications to transmit information in any manner.

As an example, consider a *Strong* failure detector $\mathcal{D}$ [CT96]. Each failure detector module of $\mathcal{D}$ outputs a *set of processes* that are suspected to have crashed, i.e., $\mathcal{R}_{\mathcal{D}} = 2^{\Pi}$. $\mathcal{D}$ satisfies the following two properties:

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

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

- *Weak Accuracy*: Some correct process is never suspected. More precisely:

$$\forall F, \forall H \in \mathcal{D}(F), \exists p \in correct(F),$$
$$\forall t \in \mathcal{T}, \forall q \in \Pi - F(t) : p \notin H(q, t)$$

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

Let $\mathcal{C}$ be a class of failure detectors. An algorithm solves a problem using $\mathcal{C}$ if it can solve this problem using any $\mathcal{D} \in \mathcal{C}$. An algorithm implements $\mathcal{C}$ if it implements some $\mathcal{D} \in \mathcal{C}$.

5 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 11.1). We also briefly consider stronger types of communication primitives — reliable send and receive, and uniform reliable broadcast — in Section 10.

5.1 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, $\mathsf{send}_{s,r}$ and $\mathsf{receive}_{r,s}$, that must satisfy Integrity and the following property:

- *Quasi No Loss*[6]: For all $k \geq 1$, if both s and r are correct and s sends m to r exactly k times, then r receives m from s at least k times.

Note that Quasi No Loss together with Integrity implies that for all $k \geq 0$, if both s and r are correct and s sends m to r exactly k times, then r receives m from s exactly k times.

We want to implement quasi reliable send/receive primitives using the (lossy) send/receive primitives that are provided by the network. In order to differentiate between these two, the first set of primitives is henceforth denoted by SEND/RECEIVE, and the second one, by send/receive. Informally, an implementation of $\mathsf{SEND}_{s,r}$ and $\mathsf{RECEIVE}_{r,s}$ is *quiescent* if a finite number of invocations of $\mathsf{SEND}_{s,r}$ cause only a finite number of invocations of sends throughout the network.

5.2 Reliable Broadcast

Reliable broadcast [BT85] is defined in terms of two primitives: $\mathsf{broadcast}(m)$ and $\mathsf{deliver}(m)$. We say that process *p broadcasts message m* if p invokes $\mathsf{broadcast}(m)$. 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 $\mathsf{deliver}(m)$. Primitives broadcast and deliver satisfy the following properties[HT94]:

- *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 $sender(m)$.

[6] A stronger property, called *No Loss*, is used in Section 10.1 to define *reliable* send and receive.

```
1  For every process p:
2
3      To execute broadcast(m):
4          deliver(m)
5          for all q ∈ neighbor(p) do SEND_{p,q}(m)
6          return
7
8      upon RECEIVE_{p,q}(m) do
9          if p has not previously executed deliver(m) then
10             deliver(m)
11             for all q ∈ neighbor(p) do SEND_{p,q}(m)
```

Figure 2: Quiescent implementation of reliable broadcast using a quiescent implementation of SEND and RECEIVE primitives between neighbors

We want to implement reliable broadcast using the (lossy) send and receive primitives that are provided by the network. Informally, an implementation of reliable broadcast is *quiescent* if a finite number of invocations of broadcast cause only a finite number of invocations of sends throughout the network.

5.3 Relation between 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* $\mathsf{SEND}_{p,q}$ *and* $\mathsf{RECEIVE}_{q,p}$ *for every pair of processes* p *and* q.

The implementation is trivial: to SEND a message m to q, p simply broadcasts the message $M = (m, p, q, k)$ using the given quiescent implementation of reliable broadcast, where $sender(M) = p$ and $seq(M) = k$, a sequence number that p has not used before. Upon the delivery of (m, p, q, k), a process r RECEIVEs m from p if $r = q$, and discards m otherwise. This implementation of $\mathsf{SEND}_{p,q}$ and $\mathsf{RECEIVE}_{q,p}$ is clearly correct and quiescent.

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* $\mathsf{SEND}_{p,q}$ *and* $\mathsf{RECEIVE}_{q,p}$ *for all processes* p *and* $q \in \text{neighbor}(p)$*, then we can obtain a quiescent implementation of reliable broadcast.*

The implementation of reliable broadcast, a simple flooding algorithm taken from [HT94], is given in Figure 2 (the code consisting of lines 9 and 10 is executed atomically[7]). It is clear that this implementation is quiescent. Indeed, for every message m, an invocation of $\mathsf{broadcast}(m)$ can cause at most $n - 1$ invocations of SEND per process. Moreover, since the implementation of SEND is quiescent, each invocation of SEND causes only a finite number of invocations of sends. Thus, a finite number of invocations of broadcast causes a finite number of invocations of sends.

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

6 Impossibility of Quiescent Reliable Communication

Quiescent reliable communication cannot be achieved in a network with process crashes and message losses. This holds even if the network is completely connected, only a finite number of messages can be lost, and processes have access to a Strong failure detector.

Theorem 1 *Consider a network 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, and the implementation can use $\mathcal{S}$.*

Proof (Sketch). Assume, by contradiction, that there exists a quiescent implementation I of quasi reliable $\mathsf{SEND}_{s,r}$ and $\mathsf{RECEIVE}_{r,s}$ using $\mathcal{S}$. We now construct three runs of I, namely, R_0, R_1 and R_2, in which only s may SEND a message M to r and no other process invokes any SEND.

In run R_0, s SENDs no messages, all processes are correct, all messages are received one time unit after they are sent, and the failure detector behaves perfectly (i.e., no process suspects any other process). Since I is quiescent, there is a time t_0 after which no messages are sent or received. By the Integrity property of SEND and RECEIVE, process r never RECEIVEs any message.

Run R_1 is identical to run R_0 up to time t_0; at time $t_0 + 1$, s SENDs M to r, and r crashes; after time $t_0 + 1$, no processes crash, and all messages are received one time unit after they are sent; at all times, the failure detector behaves perfectly (i.e., r is suspected by all processes from time $t_0 + 1$ on, and there are no other suspicions). 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 and its failure detector module behave 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 and their failure detector modules 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 the send and receive primitives satisfy the Integrity property. Assume that for some $k \geq 1$, some process q receives m from some process p k times. There are several cases. (1) If $q = r$ then r receives m from p k times in R_0 (since r behaves in the same way in R_0 and R_2). In R_0, by the Integrity property of send and receive, p sent m to r at least k times. 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 by time t_0 in R_2. (2) If $q \neq r$ and $p = r$, then q receives m from r k times in R_1 (since q behaves in the same way in R_1 and R_2). In R_1, by the Integrity property of send and receive, r sent m to q at least k times. 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 by time t_0 in R_2. (3) If $q \neq r$ and $p \neq r$, then q receives m from p k times in R_1 (since q behaves in the same way in R_1 and R_2). By the Integrity property of send and receive in R_1, p sent m to q at least k times. 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. Therefore, the send and receive primitives in R_2 satisfy the Integrity property.

We next show that in R_2 the send and receive primitives 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 the send and receive primitives satisfy the Fairness property.

Finally, we show that in R_2 the failure detector satisfies the properties of a Strong failure detector. Indeed, there are no crashes and therefore Strong Completeness holds vacuously; also there exists a process, namely s, which is never suspected by any process, and so Weak Accuracy holds.

We conclude that R_2 is a possible run of I using $\mathcal{S}$ 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 SENDs M to r; and (c) r does not $\mathsf{RECEIVE}$ M. This violates the Quasi No Loss property of $\mathsf{SEND}_{s,r}$ and $\mathsf{RECEIVE}_{r,s}$, and so I is not an implementation of $\mathsf{SEND}_{s,r}$ and $\mathsf{RECEIVE}_{r,s}$ — a contradiction. □

Theorem 1 and Remark 1 immediately imply:

Corollary 2 *There is no quiescent implementation of reliable broadcast, even if the implementation can use $\mathcal{S}$.*

To overcome these impossibility results, we now introduce the heartbeat failure detector.

7 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 faulty neighbor is bounded. Formally:

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

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

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

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

$$\begin{array}{l} \forall F, \forall H \in \mathcal{D}(F), \forall p \in \mathit{correct}(F), \\ \forall q \in \mathit{correct}(F) \cap \mathit{neighbor}(p), \\ \forall K \in \mathbb{N}, \exists t \in \mathcal{T} : H(p,t)[q] > K \end{array}$$

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* [ACT97b], rather than just the heartbeats of the neighbors of p, but we do not need this generality here.

8 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[8] and fair (Fig. 1a). 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. 1b). 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.

8.1 The Simple Network Case

We assume that all links in the network are bidirectional and fair (Fig. 1a). We first give a quiescent implementation of quasi reliable $\mathsf{SEND}_{s,r}$ and $\mathsf{RECEIVE}_{r,s}$ for the case $r \in neighbor(s)$ (see Fig. 3). To SEND a message m to r, process s first forks the task *repeat_send*(r, m, seq) where *seq* is a fresh sequence number, and then it returns from this SEND. Task *repeat_send*(r, m, seq), which runs in the background, repeatedly send (MSG, m, seq) to r, where MSG is a tag. This send occurs every time s queries its failure detector module and notices that the heartbeat value of r has increased. The task *repeat_send* terminates if s $\mathsf{receive}$s an acknowledgement (ACK, seq) from r. This acknowledgement is sent by r every time it $\mathsf{receive}$s (MSG, m, seq). Process r $\mathsf{RECEIVE}$s m at the first time it $\mathsf{receive}$s (MSG, m, seq).

The code consisting of lines 7 and 8 is executed atomically, as well as the code consisting of lines 23 and 24. If there are several concurrent executions of the *repeat_send* task (lines 11–18), then each execution must have its own private copy of all the local variables in this task, namely, r, m, *seq*, *hb* and *prev_hb_r*.

We now show that the algorithm of Fig. 3 is correct and quiescent. Note that all variables are local to each process. When ambiguities may arise, a variable local to process p is subscripted by p, e.g., hb_p is the local variable hb of process p.

Lemma 3 (Integrity) *For all $k \geq 1$, if r $\mathsf{RECEIVE}$s m from s k times, then s SENT m to r at least k times.*

Proof. Note that, for any sn, r only $\mathsf{RECEIVE}$s m from s on the first time it $\mathsf{receive}$s (MSG, m, sn) from s. Since r $\mathsf{RECEIVE}$s m k times then there are k different values $sn_1, \ldots, sn_k$ such that r $\mathsf{receive}$s (MSG, m, sn_j) from s for each $j \in \{1, \ldots, k\}$. By the Integrity property of send and $\mathsf{receive}$, s sends (MSG, m, sn_j) to r before r $\mathsf{receive}$s (MSG, m, sn_j) from s. This send can only occur in line 16 of Task *repeat_send*(r, m, sn_j). For each $j \in \{1, \ldots, k\}$, Task *repeat_send*(r, m, sn_j) can only be forked during an invocation of $\mathsf{SEND}_{s,r}(m)$, and each such invocation forks only one task *repeat_send*. Therefore, s SENT m to r at least k times. $\square$

Let m be a message and $k \geq 1$, and consider a run in which s invokes $\mathsf{SEND}_{s,r}(m)$ exactly k times. For $j = 1, \ldots, k$, we can associate a sequence number sn_j with the j-th invocation of $\mathsf{SEND}_{s,r}(m)$ as follows: we let sn_j be the value of the global variable *seq* after line 7 is executed during the j-th invocation.[9] Note that if $i \neq j$ then $sn_i \neq sn_j$, since during each invocation of $\mathsf{SEND}_{s,r}$ the global variable *seq* is increased by one, and it can never be decreased.

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

[9] If s crashes during the j-th invocation before executing line 7, we let sn_j be equal to one plus the value of the global variable *seq* at the time of the invocation.

```
1  For process s:
2
3      Initialization:
4          seq ← 0                                  { seq is the current sequence number }
5
6      To execute SEND_{s,r}(m):
7          seq ← seq + 1
8          fork task repeat_send(r, m, seq)
9          return
10
11     task repeat_send(r, m, seq):
12         prev_hb_r ← −1
13         repeat periodically
14             hb ← D_p                             { query the heartbeat failure detector }
15             if prev_hb_r < hb[r] then
16                 send_{s,r}(MSG, m, seq)
17                 prev_hb_r ← hb[r]
18         until receive_{s,r}(ACK, seq) from r
19
20 For process r:
21
22     upon receive_{r,s}(MSG, m, seq) do
23         if this is the first time r receives (MSG, m, seq) from s then RECEIVE_{r,s}(m)
24         send_{r,s}(ACK, seq)
```

Figure 3: Simple network case — quiescent implementation of $\mathsf{SEND}_{s,r}$ and $\mathsf{RECEIVE}_{r,s}$ using $\mathcal{HB}$ for $r \in neighbor(s)$

Lemma 4 (Quasi No Loss) *For all $k \geq 1$, if both s and r are correct and s SENDs m to r exactly k times, then r $\mathsf{RECEIVE}$s m from s at least k times.*

Proof. Suppose, by contradiction, that r $\mathsf{RECEIVE}$s m from s less than k times. Notice that r $\mathsf{RECEIVE}$s m from s every time it $\mathsf{receive}$s from s a message of the form $(\mathsf{MSG}, m, *)$ that it did not $\mathsf{receive}$ before. Let sn_j be the sequence number associated with the j-th invocation of $\mathsf{SEND}_{s,r}(m)$. Since all the sn_j's are distinct, there is some $j \in \{1, \ldots, k\}$ such that (MSG, m, sn_j) is never $\mathsf{received}$ by r from s. Since each sequence number is associated with a unique message, r sends (ACK, sn_j) only if it $\mathsf{receive}$s (MSG, m, sn_j). We conclude that r never sends (ACK, sn_j) to s. By the Integrity property of send and $\mathsf{receive}$, s never $\mathsf{receive}$s (ACK, sn_j) from r. When s invokes $\mathsf{SEND}_{s,r}(m)$ for the j-th time, it forks a task *repeat_send* to repeatedly send (MSG, m, sn_j) to r. This task will be referred to as task T. Task T never terminates, because it can only do so if s crashes or if s $\mathsf{receive}$s (ACK, sn_j) from r. Therefore, the loop in lines 13–18 of task T is repeated infinitely often. Moreover, since r is correct, the $\mathcal{HB}$-Accuracy property guarantees that the heartbeat sequence of r at s is nondecreasing and unbounded, and thus the condition in line 15 evaluates (in task T) to true an infinite number of times. Therefore s sends (MSG, m, sn_j) to r an infinite number of times. By the Fairness property of send and $\mathsf{receive}$, r $\mathsf{receive}$s (MSG, m, sn_j) from s at least once — a contradiction. $\square$

We now show that the implementation in Fig. 3 is quiescent. In order to do so, we focus on a single invocation of SEND and show that it causes only a finite number of invocations of sends. This immediately implies that a finite number of invocations of SENDs cause only a finite number of invocations of sends. So consider one particular invocation I of $\mathsf{SEND}_{s,r}(m)$, and let *sn* be the sequence number associated with I. It is clear that, if s does not crash, I causes invocations of $\mathsf{send}_{s,r}(\mathsf{MSG}, m, sn)$ in task *repeat_send*(r, m, sn). When r $\mathsf{receive}$s (MSG, m, sn) from s, it invokes $\mathsf{send}_{r,s}(\mathsf{ACK}, sn)$. So I may also cause invocations of $\mathsf{send}_{r,s}(\mathsf{ACK}, sn)$, and it is clear that I does not cause any other invocations of sends. Therefore, a send caused by I is either $\mathsf{send}_{s,r}(\mathsf{MSG}, m, sn)$ or $\mathsf{send}_{r,s}(\mathsf{ACK}, sn)$.

We next show that s sends (MSG, m, sn) to r only a finite number of times, and that r sends (ACK, sn) to s only a finite number of times. This implies that I causes only a finite number of sends.

Lemma 5 *s sends (MSG, m, sn) to r an infinite number of times if and only if r sends (ACK, sn) to s an infinite number of times.*

Proof. If s sends (MSG, m, sn) to r an infinite number of times, then s is correct, and the condition in line 15 evaluates to true an infinite number of times. Therefore, the heartbeat sequence of r at s is unbounded. So, by $\mathcal{HB}$-Completeness, r is correct. Then by the Fairness property of send and $\mathsf{receive}$, r $\mathsf{receive}$s (MSG, m, sn) an infinite number of times. Since r sends (ACK, sn) to s each time it $\mathsf{receive}$s (MSG, m, sn), r sends (ACK, sn) to s an infinite number of times.

Conversely, if r sends (ACK, sn) to s an infinite number of times, then r $\mathsf{receive}$s (MSG, m, sn) an infinite number of times, so, by the Integrity property of send and $\mathsf{receive}$, s sends (MSG, m, sn) to r an infinite number of times . □

Corollary 6 *s sends (MSG, m, sn) to r only a finite number of times.*

Proof. For a contradiction, suppose that s sends (MSG, m, sn) to r an infinite number of times. Then r sends (ACK, sn) to s an infinite number of times by Lemma 5. By the Fairness property of send and $\mathsf{receive}$, s eventually $\mathsf{receive}$s (ACK, sn) from r. Thus, the condition in line 18 (of task *repeat_send*(r, m, sn)) becomes true. Therefore, the task *repeat_send*(r, m, sn) eventually terminates and so it sends (MSG, m, sn) to r only a finite number of times — a contradiction. □

Corollary 7 *r sends (ACK, sn) to s only a finite number of times.*

Proof. From Lemma 5 and Corollary 6. □

Lemma 8 *The algorithm of Fig. 3 is quiescent.*

Proof. From Corollaries 6 and 7, and the remarks before Lemma 5. □

From Lemmata 3, 4, and 8 we have:

Theorem 9 *For the simple network case and any $r \in$ neighbor(s), Fig. 3 is a quiescent implementation of quasi reliable $\mathsf{SEND}_{s,r}$ and $\mathsf{RECEIVE}_{r,s}$ that uses $\mathcal{HB}$.*

From this theorem, and Remarks 1 and 2, we have:

Corollary 10 *In the simple network case, quasi reliable send and receive between every pair of processes and reliable broadcast can both be implemented with quiescent algorithms that use $\mathcal{HB}$.*

8.2 The General Network Case

In this case (Fig. 1b), 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 SEND a message m to the receiver r, it has to use a diffusion mechanism, even if r is a neighbor of s (since link $s \to r$ may be unfair). 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 "reliably" diffused — a chicken and egg problem.

Figure 4 shows a quiescent implementation of reliable broadcast (by Remark 1 it can be used to obtain quasi reliable send and receive between every pair of processes). 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. In order to 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 $diffuse(m)$ at p 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 a message containing m to *all* neighbors whose heartbeat increased — even to those who are already in $got_p[m]$.[10] The task terminates when all neighbors of p are contained in $got_p[m]$.

All messages sent by the algorithm are of the form $(m, got\_msg, path)$ where got_msg is a set of processes and $path$ is a sequence of processes. Upon the receipt of such a message, 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 $(m, got\_msg, path)$ to all its neighbors that appear at most once in $path$.

The code consisting of lines 18 through 26 is executed atomically. Each concurrent execution of the *diffuse* task (lines 9 to 16) has its own private copy of all the local variables in this task, namely m, *hb*, and *prev_hb*.

We now show that this implementation is correct and quiescent.

Lemma 11 (Uniform Integrity) *For every message m, every process delivers message m at most once, and only if m was previously broadcast by $sender(m)$.*

Proof (Sketch). Let m be a message and p be a process. Line 19 guarantees that p delivers m at most once. Now suppose process p delivers m. It can do that either in line 4 or line 20. In the first case, p previously broadcast m and clearly $p = sender(m)$. In the second case, p must have received a message of the form $(m, *, *)$. By the Integrity property of send and receive, a message of the form $(m, *, *)$ was previously sent. An easy induction shows that this can only happen if m was previously broadcast by $sender(m)$. □

Lemma 12 (Validity) *If a correct process broadcasts a message m, then it eventually delivers m.*

Proof. If a correct process broadcasts m then it eventually executes line 4 and delivers m. □

We next show that every process in $got_p[m]$ delivered m. But first, we should be concerned about when the initialization of $got_p[m]$ takes place. We do not assume $got_p[m]$ is initialized to the empty set at start-up. Doing so would be impractical since the set of all possible messages m that can ever be broadcast can be infinite. Instead,

[10] It may appear that p does not need to send this message to processes in $got_p[m]$, since they already got it! The reader should verify that this "optimization" would make the algorithm fail.

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

Figure 4: General network case — quiescent implementation of broadcast and deliver using $\mathcal{HB}$

each process p initializes $got_p[m]$ either when it broadcasts m (see line 5), or when it first "hears" about m (see line 21). This guarantees that $got_p[m]$ is never used before it is initialized. We next establish invariants for $got_p[m]$, but one should always keep in mind that these invariants only hold after initialization has occurred.

Lemma 13 *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.*

Proof (Sketch). (1) and (2) are clear from the algorithm and (3) follows from the Integrity property of send and receive. □

Lemma 14 *For every m and path, there is a finite number of distinct messages of the form $(m, *, path)$.*

Proof. Any message of the form $(m, *, path)$ is equal to $(m, g, path)$ for some $g \subseteq \Pi$, where Π is a finite set. □

Lemma 15 *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 13 part (3) q never belongs to $got_p[m]$. Since

p is correct, this implies that p executes the loop in lines 11–16 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 14, 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 send and receive, q eventually receives (m, g_0, p). Therefore, q delivers m. This contradicts the assumption that q does not deliver m. □

Lemma 16 (Agreement) *If a correct process delivers a message m, then all correct processes eventually deliver m.*

Proof. Suppose that some correct process p delivers m. For every correct process q, there is a simple fair path $(p_1, \ldots, p_k)$ from p to q with $p_1 = p$ and $p_k = q$. By successive applications of Lemma 15, we conclude that $p_2, p_3, \ldots, p_k$ eventually deliver m. Therefore $q = p_k$ eventually delivers m. □

We now show that the implementation of Fig. 4 is quiescent. In order to do so, we focus on a single invocation of broadcast and show that it causes only a finite number of invocations of sends. This implies that a finite number of invocations of broadcast cause only a finite number of invocations of sends.

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 17 *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 13 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 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 $1 \leq i < k$, let $P_i = (p_1, p_2, \ldots, p_i)$. Note that for $1 \leq i < k$, process p_{i+1} appears at most once in P_i.

We claim that for each $j = 1, \ldots, k-1$, 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 send and receive 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 11–16 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 a message of the form $(m, *, p_1)$ to p_2 an infinite number of times. By Lemma 14, 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 13 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 send and receive, 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 14, there exists $g_{j+1} \subseteq \Pi$ such that g_{j+1} contains $\{p_1, p_2, \ldots, p_{j+1}\}$ and p_{j+1} sends (m, g_{j+1}, P_{j+1}) to p_{j+2} an infinite number of times. □

Corollary 18 *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 17. If q is faulty, 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 $diffuse(m)$. $\square$

Lemma 19 *If some process* sends *a message of the form $(m, *, path)$, then no process appears more than twice in path.*

Proof (Sketch). By line 25 of the algorithm, a process sends a message $(m, g, path)$ to a process q only if q appears at most once in *path*. The result follows by an easy induction that uses this fact and the Integrity property of send and $\mathsf{receive}$. $\square$

Lemma 20 (Quiescence) *Eventually every process stops* sending *messages of the form $(m, *, *)$.*

Proof. Suppose for a contradiction that some process p never stops sending messages of the form $(m, *, *)$. Note that p must be correct. By Lemma 19, the third component of a message of the form $(m, *, *)$ ranges over a finite set of values. Therefore, for some $path$, p sends an infinite number of messages of the form $(m, *, path)$. By Lemma 14, for some $g \subseteq \Pi$, p sends an infinite number of messages $(m, g, path)$. So, for some process q, process p invokes $\mathsf{send}_{p,q}(m, g, path)$ an infinite number of times.

There are two cases. First, if $path$ is empty we immediately reach a contradiction since a send with empty $path$ can occur neither in line 14 nor in line 26. For the second case, suppose that $path$ consists of at least one process and let $path = (p_1, \ldots, p_k)$, where $k \geq 1$. Corollary 18 shows that there is a time after which p stops sending messages in its task $diffuse(m)$. Since p only invokes send in task $diffuse(m)$ or in line 26, then an infinite number of invocations of $\mathsf{send}_{p,q}(m, g, path)$ occurs at line 26. Each such invocation can occur only when p $\mathsf{receive}$s a message of the form $(m, *, path')$ where $path' = (p_1, \ldots, p_{k-1})$. So p $\mathsf{receive}$s a message of the form $(m, *, path')$ an infinite number of times. By the Integrity property of send and $\mathsf{receive}$, there is an infinite number of sends of a message of this form to p. By Lemma 14, for some $g' \subseteq \Pi$, there is an infinite number of sends $(m, g', path')$ to p. Therefore, there exists a correct process p' such that $\mathsf{send}_{p',p}(m, g', path')$ is invoked an infinite number of times. By repeating this argument $k-1$ more times we conclude that there exist $g^{(k)} \subseteq \Pi$, and correct processes $p^{(k)}$ and $p^{(k-1)}$ such that $\mathsf{send}_{p^{(k)},p^{(k-1)}}(m, g^{(k)}, path^{(k)})$ is invoked an infinite number of times, where $path^{(k)}$ is empty. $\square$

From Lemmata 11, 12, 16, and 20 we have:

Theorem 21 *For the general network case, Fig. 4 is a quiescent implementation of reliable broadcast that uses $\mathcal{HB}$.*

From this theorem and Remark 1 we have:

Corollary 22 *In the general network case, quasi reliable send and receive between every pair of processes can be implemented with a quiescent algorithm that uses $\mathcal{HB}$.*

9 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.

```
1  For every process p:
2
3      Initialization:
4          for all q ∈ neighbor(p) do D_p[q] ← 0
5
6      cobegin
7          || Task 1:
8              repeat periodically
9                  for all q ∈ neighbor(p) do send_{p,q}(HEARTBEAT)
10
11         || Task 2:
12             upon receive_{p,q}(HEARTBEAT) do
13                 D_p[q] ← D_p[q] + 1
14     coend
```

Figure 5: Simple network case — implementation of $\mathcal{HB}$

9.1 The Simple Network Case

We assume all links in the network are bidirectional and fair (Fig. 1a). In this case, the implementation is obvious. Every process p executes two concurrent tasks (Fig. 5). In the first one, p periodically sends message HEARTBEAT to all its neighbors. The second task handles the receipt of HEARTBEAT messages. Upon the receipt of such a message from process q, p increases the heartbeat value of q.

We now prove that the implementation is correct.

Lemma 23 ($\mathcal{HB}$-Completeness) *At each correct process, the heartbeat sequence of every faulty neighbor is bounded.*

Proof. Obvious. □

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

Proof. This is clear since $\mathcal{D}_p[q]$ can only be changed in line 13. □

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

Proof. Since $q \in neighbor(p)$ and all links are bidirectional, we have $p \in neighbor(q)$. Moreover, since q is correct, its Task 1 executes forever. Therefore, q sends an infinite number of HEARTBEAT messages to p. By the Fairness property of send and receive, p receives an infinite number of HEARTBEAT messages from q. Every time p receives HEARTBEAT from q, it increments $\mathcal{D}_p[q]$ in line 13. So, p increments $\mathcal{D}_p[q]$ an infinite number of times. Moreover, by Lemma 24, $\mathcal{D}_p[q]$ can never be decremented. So, the heartbeat sequence of q at p is unbounded. □

Corollary 26 ($\mathcal{HB}$-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 24 and 25. □

From Lemma 23 and the above corollary, we have:

Theorem 27 *For the simple network case, Fig. 5 implements $\mathcal{HB}$.*

1 For every process p:

2

3 Initialization:

4 **for all** $q \in neighbor(p)$ **do** $\mathcal{D}_p[q] \leftarrow 0$

5

6 **cobegin**

7 || *Task 1*:

8 **repeat periodically**

9 **for all** $q \in neighbor(p)$ **do** $\mathsf{send}_{p,q}(\mathsf{HEARTBEAT}, p)$

10

11 || *Task 2*:

12 **upon** $\mathsf{receive}_{p,q}(\mathsf{HEARTBEAT}, path)$ **do**

13 **for all** q such that $q \in neighbor(p)$ and q appears in $path$ **do**

14 $\mathcal{D}_p[q] \leftarrow \mathcal{D}_p[q] + 1$

15 $path \leftarrow path \cdot p$

16 **for all** q such that $q \in neighbor(p)$ and q does not appear in $path$ **do**

17 $\mathsf{send}_{p,q}(\mathsf{HEARTBEAT}, path)$

18 **coend**

Figure 6: General network case — implementation of $\mathcal{HB}$

9.2 The General Network Case

In this case some links may be unidirectional and/or not fair (Fig. 1b). 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. 6. 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 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 proceed to prove the correctness of the implementation.

Lemma 28 *At every process p, the heartbeat sequence of every neighbor q is nondecreasing.*

Proof. This is clear since $\mathcal{D}_p[q]$ can only be changed in line 14. □

Lemma 29 *At each correct process, the heartbeat sequence of every correct neighbor is unbounded.*

Proof. Let p be a correct process, and q be a correct neighbor of p. Let $P = (p_1, \ldots, p_k)$ be a simple fair path from q to p with $p_1 = q$ and $p_k = p$. For $j = 1, \ldots, k$, let $P_j = (p_1, \ldots, p_j)$. For each $j = 1, \ldots, k-1$, we claim that p_j sends $(\mathsf{HEARTBEAT}, P_j)$ to p_{j+1} an infinite number of times. We show this by induction on j. For the base case ($j = 1$), note that $p_1 = q$ is correct, so its Task 1 executes forever and therefore p_1 sends $(\mathsf{HEARTBEAT}, p_1)$ to all its neighbors, and thus to p_2, an infinite number of times. For the induction step, let

$j < k-1$ and assume that p_j sends $(\mathsf{HEARTBEAT}, P_j)$ to p_{j+1} an infinite number of times. Since p_{j+1} is correct and the link $p_j \to p_{j+1}$ is fair, p_{j+1} receives $(\mathsf{HEARTBEAT}, P_j)$ an infinite number of times. Moreover, p_{j+2} does not appear in P_{j+1} and p_{j+2} is a neighbor of p_{j+1}, so each time p_{j+1} receives $(\mathsf{HEARTBEAT}, P_j)$, it sends $(\mathsf{HEARTBEAT}, P_{j+1})$ to p_{j+2} in line 17. Therefore, p_{j+1} sends $(\mathsf{HEARTBEAT}, P_{j+1})$ to p_{j+2} an infinite number of times. This shows the claim.

For $j = k-1$ this claim shows that p_{k-1} sends $(\mathsf{HEARTBEAT}, P_{k-1})$ to p_k an infinite number of times. Process p_k is correct and link $p_{k-1} \to p_k$ is fair, so p_k receives $(\mathsf{HEARTBEAT}, P_{k-1})$ an infinite number of times. Note that $q \in \mathit{neighbor}(p_k)$ (since $p_k = p$) and $q \in P_{k-1}$ (since $p_1 = q$). So every time p_k receives $(\mathsf{HEARTBEAT}, P_{k-1})$, it increments $\mathcal{D}_{p_k}[q]$ in line 14. So $\mathcal{D}_{p_k}[q]$ is incremented an infinite number of times. Note that, by Lemma 28, $\mathcal{D}_{p_k}[q]$ can never be decremented. So, the heartbeat sequence of q at $p_k = p$ is unbounded. □

Corollary 30 ($\mathcal{HB}$-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 28 and 29. □

Lemma 31 *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.*

Proof (Sketch). This follows from lines 9, 15 and 16, and a simple induction that uses the Integrity property of send and receive. □

Lemma 32 *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.*

Proof. Let M be the message $(\mathsf{HEARTBEAT}, \mathit{path} \cdot q)$ and let M_0 be the message $(\mathsf{HEARTBEAT}, \mathit{path})$. Suppose p receives M an infinite number of times. By the Integrity property of send and receive, some process p' sends M to p an infinite number of times. By Lemma 31 part (1), we have $q = p'$. Since the length of $\mathit{path} \cdot q$ is at least two, q can only send M in line 17. So q only sends M if it receives M_0. Therefore q receives M_0 an infinite number of times. □

Lemma 33 ($\mathcal{HB}$-Completeness) *At each correct process, the heartbeat sequence of every faulty neighbor is bounded.*

Proof (Sketch). Let p be a correct process and let q be a faulty neighbor of p. 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 31 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 $(\mathsf{HEARTBEAT}, \mathit{path})$ an infinite number of times.

Let $\mathit{path} = (p_1, \ldots, p_k)$. Then, for some $j \leq k$, $p_j = q$. If $j = k$ then, by the Integrity property of send and receive and by Lemma 31 part (1), q sends $(\mathsf{HEARTBEAT}, \mathit{path})$ to p an infinite number of times. This contradicts the fact that q is faulty. If $j < k$ then, by repeated applications of Lemma 32, we conclude that p_{j+1} receives message $(\mathsf{HEARTBEAT}, (p_1, \ldots, p_j))$ an infinite number of times. Therefore, by the Integrity property of send and receive and Lemma 31 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 is faulty. □

By Corollary 30 and the above lemma, we have:

Theorem 34 *For the general network case, Fig. 6 implements $\mathcal{HB}$.*

For process s:

  Initialization:
    $seq \leftarrow 0$ { seq is the current sequence number }

  To execute R-SEND$_{s,r}(m)$:
    $seq \leftarrow seq + 1$
    $lseq \leftarrow seq$
    broadcast$(m, lseq, s, r)$
    wait until RECEIVEd (ACK, $lseq$) from $t + 1$ processes
    return

For every process p:

  upon deliver$(m, lseq, s, r)$ **do**
    SEND$_{p,s}$(ACK, $lseq$)
    if $p = r$ **then** R-RECEIVE$_{r,s}(m)$

Figure 7: Quiescent implementation of R-SEND$_{s,r}$ and R-RECEIVE$_{r,s}$ for $n > 2t$

10 Stronger Communication Primitives

Quasi reliable send and receive and reliable broadcast are sufficient to solve many problems (see Section 11.1). 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 highly inefficient. More efficient ones can be obtained by modifying the algorithms in Figures 3 and 4. Hereafter, we assume that $t < n/2$.

10.1 Reliable Send and Receive

If a process s returns from the invocation of send$_{s,r}(m)$ we say that s *completes the sending of message m to r*. With quasi reliable send and receive, it is possible that s completes the sending of m to r, then s crashes, and r never receives m (even though it does not crash). In contrast, with reliable send and receive primitives, if s completes the sending of message m to a correct process r then r eventually receives m (even if s crashes). More precisely, reliable send and receive satisfy Integrity (Section 4.2) and:

- *No Loss*: For all $k \geq 1$, if r is correct and s completes the sending of m to r exactly k times, then r receives m from s at least k times.[11]

[11] The No Loss and Quasi No Loss properties are very similar to the Strong Validity and Validity properties in Section 6 of [HT94].

```
For every process p:

    To execute uniform-broadcast(m):
        broadcast(m)
        return

    upon deliver(m) do
        for all q ∈ Π do SEND_{p,q}(ACK, m)
        wait until RECEIVEd (ACK, m) from t + 1 processes
        uniform-deliver(m)
```

Figure 8: Quiescent implementation of uniform reliable broadcast for $n > 2t$

Reliable send and receive primitives are denoted R-SEND/R-RECEIVE. As before, SEND/RECEIVE denote the quasi reliable primitives.

Figure 7 shows a quiescent implementation of R-SEND and R-RECEIVE (the code consisting of lines 7 and 8 is executed atomically). It uses reliable broadcast and SEND/RECEIVE between every pair of processes. We have already shown that these primitives have quiescent implementations using $\mathcal{HB}$ for the two types of network in consideration.

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 RECEIVE $t+1$ acknowledgements for that message before returning from this invocation of R-SEND. When a process p delivers this broadcast message, it SENDs an acknowledgement back to s, and if $p = r$ then it also R-RECEIVEs m from s. The proof of correctness is straightforward and thus omitted.

10.2 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 [Gra78, Had86, BT93]. For such applications, a stronger version of reliable broadcast is more suitable, namely, *uniform reliable broadcast* which satisfies Uniform Integrity, Validity (Section 5.2) and:

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

Figure 8 shows a quiescent implementation of uniform reliable broadcast which uses reliable broadcast and SEND/RECEIVE between every pair of processes. The proof of correctness is straightforward and thus omitted.

11 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.

11.1 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, Rab83, CMS89, FM90], the failure-detector based ones of [CT96, AT96], 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]. These algorithms tolerate process crashes, and they use quasi reliable send and receive, and/or reliable broadcast, as their sole communication primitives. All of these algorithms can be made to tolerate both process crashes and message losses (with fair links) in two simple steps: (1) implement $\mathcal{HB}$ as described in Section 9, and (2) plug in the quiescent communication primitives given in Section 8.[12] The resulting algorithms tolerate message losses and are quiescent.

11.2 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. Fair lossy and fair links differ in a subtle way. For instance, if process p sends the sequence of distinct messages $m_1, m_2, m_3, \ldots$ to q and $p \to q$ is fair lossy, then q is guaranteed to receive an infinite subsequence, whereas if $p \to q$ is fair, q may receive nothing (because each distinct message is sent only once). On the other hand, if p sends the sequence $m_1, m_2, m_1, m_2, \ldots$ and $p \to q$ is fair lossy, q may never receive a copy of m_2 (while it receives m_1 infinitely often), whereas if $p \to q$ is fair, q is guaranteed to receive an infinite number of copies of both m_1 and m_2.[13]

[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*[14] 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 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.

[12]Similar steps can be applied to algorithms that use reliable send/receive or uniform reliable broadcast, provided a majority of processes are correct, by plugging in the implementations given in Section 10.

[13]In [BCBT96], message piggybacking is used to overcome message losses. To avoid this piggybacking, in this paper we adopted the model of fair links: message losses can now be overcome by separately sending each message repeatedly.

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

12 Generalization to Networks that Partition

In this paper, we assumed that every pair of correct processes are reachable from each other through fair paths. In [ACT97b], we drop this assumption and consider the more general problem of quiescent reliable communication in networks that may partition. In particular, we (a) generalize the definitions of quasi reliable send and receive and of reliable broadcast, (b) generalize the definition of the heartbeat failure detector and implement it in networks that may partition, and (c) show that this failure detector can be used to achieve quiescent reliable communication in such networks. In [ACT97b] we also consider the problem of consensus for networks that may partition, and we use $\mathcal{HB}$ to solve this problem with a quiescent protocol (we also use a generalization of the *Eventually Strong* failure detector [CT96]).

13 Quiescence versus Termination

In this paper we considered 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 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*. This follows from the result in [KT88] which shows that in a system with message losses (fair links) and no process crashes there is no terminating protocol that guarantees knowledge gain.

Acknowledgments

We are grateful to Anindya Basu and Vassos Hadzilacos 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.

References

[ACT97a] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. On the weakest failure detector to achieve quiescence. Manuscript, April 1997.

[ACT97b] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Quiescent reliable communication and quiescent consensus in partitionable networks. Technical Report 97-1632, Department of Computer Science, Cornell University, June 1997.

[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.

[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.

[DFKM96] Danny Dolev, Roy Friedman, Idit Keidar, and Dahlia Malkhi. Failure detectors in omission failure environments. Technical Report 96-1608, Department of Computer Science, Cornell University, Ithaca, New York, 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 13–15, 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*, pages 87–100, Le Mont-St-Michel, France, 1995. Springer Verlag, LNCS 972.

[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.

[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.

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

[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.