Heartbeat: A Timeout-Free Failure Detector for Quiescent Reliable Communication*

Marcos Kawazoe Aguilera Wei Chen Sam Toueg

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

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 $\diamond\mathcal{P}$ [2], 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 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, 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 [12], 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 [5, 15, 20, 24, 27].

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 $\diamond\mathcal{P}$ [12]. 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 [2], 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. Unfortunately, $\diamond\mathcal{P}$ is not implementable in asynchronous systems with process crashes (this would violate a known impossibility result [18, 12]). 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.

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* [28, 10]). 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? [2] 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 [2], 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 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 [22] ensures that if a correct process broadcasts a

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

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 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 unidirectional rings that intersect.

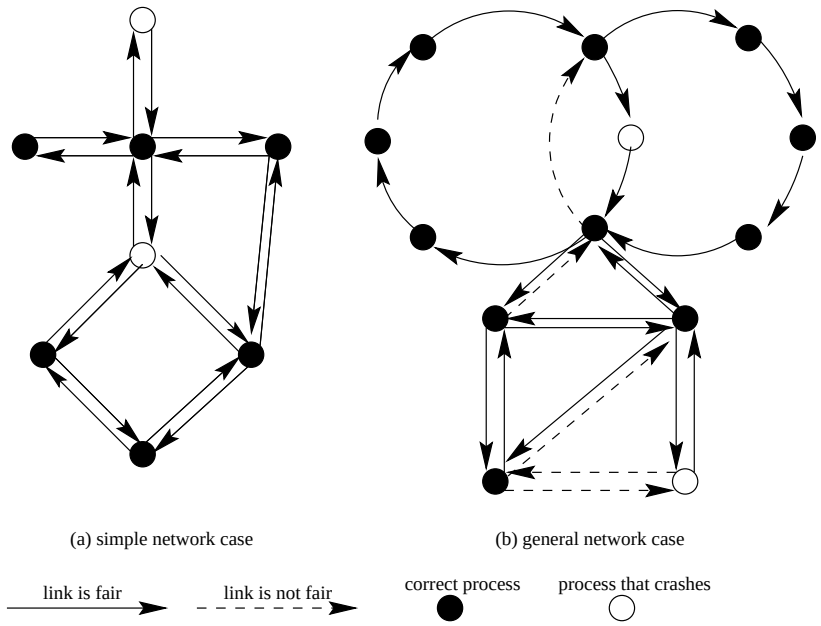

Fig. 1. Examples of the simple and general network cases

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.

[3] This assumption precludes permanent network partitioning.

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 [4, 8, 9, 12, 14, 17, 26], for atomic broadcast in [12], for k-set agreement in [13], for atomic commitment in [20], for approximate agreement in [16], etc.

Finally, we show that $\mathcal{HB}$ can be used to extend the work in [6] 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:

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 [6].
4. $\mathcal{HB}$ is novel: it is implementable without timeouts, and it does not output lists of suspects as typical failure detectors do [5, 12, 20, 21, 24, 27]. The results of this paper, combined with those in [2], 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 [25] 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 [6] is the closest to ours, but their protocols do not use failure detectors and are not quiescent. In Section 10, we use $\mathcal{HB}$ to extend the results of [6] 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 explain how to use heartbeat failure detectors to extend several previous results. In Section 11, 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.

All proofs are omitted here due to space limitations. They are provided in [1].

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

[4] The link failure model in [6] is slightly different from the one used here (cf. Section 10).

[5] This was anticipated in [11].

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 $crashed(F) = \bigcup_{t \in \mathcal{T}} F(t)$ 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 $p \rightarrow q$, and if, in addition, $q \neq p$ we say that q is a *neighbor* of p. The set of neighbors of p is denoted by $neighbor(p)$.

With every link $p \rightarrow q$ 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 $p \rightarrow q$ by the properties that its $\mathsf{send}_{p,q}$ and $\mathsf{receive}_{q,p}$ primitives satisfy. We assume that links do not create or duplicate messages, i.e., every link $p \rightarrow q$ in the network satisfies:

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

A lossy link can fail by dropping messages. A link $p \rightarrow q$ is *fair* if $\mathsf{send}_{p,q}$ and $\mathsf{receive}_{q,p}$ satisfy Uniform 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 $p_1 \rightarrow p_2$, $\ldots$, $p_{k-1} \rightarrow p_k$ are fair. We assume that every pair of distinct correct processes is connected through a fair 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. Note that the output of a failure detector depends *only* on the failure pattern F. Thus, it does not depend on the behavior of applications.

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 consider quasi reliable send and receive and reliable broadcast, because these communication primitives are sufficient to solve many problems (see Section 10.1). The full version of this paper [1] also considers stronger types of communication primitives, namely, reliable send and receive, and uniform reliable broadcast.

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 Uniform 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 by time t, then r eventually receives m from s at least k times.

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

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 it sends only a finite number of messages when $\mathsf{SEND}_{s,r}$ is invoked a finite number of times.[7]

5.2 Reliable Broadcast

Reliable broadcast [9] 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[22]:

[6] A stronger property, called *No Loss*, is used to define reliable send and receive [1].

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

– *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 $sender(m)$.

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 it sends only a finite number of messages when broadcast is invoked a finite number of times.

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

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 and only a finite number of messages can be lost.

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.*

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.*

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:

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

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

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

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

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

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* [3], 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). In this case, the algorithms are very simple. We first give a quiescent implementation of quasi reliable $\mathsf{SEND}_{s,r}$ and $\mathsf{RECEIVE}_{r,s}$ for the case $r \in \mathit{neighbor}(s)$. For s to SEND a message m to r, it repeatedly sends m to r every time the heartbeat of r increases, until s receives $\mathit{ack}(m)$ from r. Process r RECEIVEs m from s the first time it receives m from s, and r sends $\mathit{ack}(m)$ to s every time it receives m from s.

From this implementation, and Remark 2, 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.

[8] 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, $q \in \mathit{neighbor}(p)$ if and only if $p \in \mathit{neighbor}(q)$.

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 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 "reliably" diffused — a chicken and egg problem.

Figure 2 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 $got_p[m]$.[9] 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 19 through 27 is executed atomically.[10] Each concurrent execution of the *diffuse* task (lines 9 to 17) has its own copy of all the local variables in this task.

Theorem 3. *For the general network case, the algorithm in Fig. 2 is a quiescent implementation of reliable broadcast that uses $\mathcal{HB}$.*

Corollary 4. *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.

[9] It may appear that p does not need to send this message to processes in $got_p[m]$, since they already got it! With this "optimization" the algorithm is no longer quiescent.

[10] 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_{p,q}(m, got[m], p)
16                  prev_hb ← hb
17          until neighbor(p) ⊆ got[m]
18
19      upon receive_{p,q}(m, got_msg, path) 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_{p,q}(m, got[m], path)
```

Fig. 2. General network case — quiescent implementation of broadcast and deliver using $\mathcal{HB}$

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. 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.

9.2 The General Network Case

In this case some links are 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. 3. Every process p executes two concurrent tasks. In the first task, p periodically sends message $(\text{HEARTBEAT}, p)$ to all its neigh-

bors. 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*.

```
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, p)
10
11          || Task 2:
12              upon receive_{p,q}(HEARTBEAT, path) do
13                  for all q such that q ∈ neighbor(p) and q appears in path do
14                      D_p[q] ← D_p[q] + 1
15                  path ← path · p
16                  for all q such that q ∈ neighbor(p) and q does not appear in path do
17                      send_{p,q}(HEARTBEAT, path)
18      coend
```

Fig. 3. General network case — implementation of $\mathcal{HB}$

Theorem 5. *For the general network case, the algorithm in Fig. 3 implements $\mathcal{HB}$.*

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 [8, 14, 17, 26], the failure-detector based ones of [4, 12], the probabilistic one of [9], and the algorithms for atomic broadcast in [12], k-set agreement in [13], atomic commitment in [20], and approximate agreement in [16]. 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 8. 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 [6]. 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 [6] is called *fair lossy*. Roughly speaking, a fair lossy link $p \rightarrow q$ 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 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.[11]

[6] 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*[12] or a majority of processes are correct. For each of these two cases, [6] 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 [6] 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 [6], 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.

11 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 [3], 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 [3] we also

[11] In [6], 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.

[12] Intuitively, a problem P is correct-restricted if its specification does not refer to the behavior of faulty processes [7, 19].

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 [12]).

12 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 [23] 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, Bernadette Charron-Bost, 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

1. M. K. Aguilera, W. Chen, and S. Toueg. Heartbeat: a timeout-free failure detector for quiescent reliable communication. Technical Report 97-1631, Department of Computer Science, Cornell University, May 1997.
2. M. K. Aguilera, W. Chen, and S. Toueg. On the weakest failure detector for quiescent reliable communication. Technical report, Department of Computer Science, Cornell University, July 1997.
3. M. K. Aguilera, W. Chen, and S. Toueg. Quiescent reliable communication and quiescent consensus in partitionable networks. Technical Report 97-1632, Department of Computer Science, Cornell University, June 1997.
4. M. K. Aguilera and S. 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, Oct. 1996.
5. Ö. Babaoğlu, R. Davoli, and A. Montresor. Partitionable group membership: specification and algorithms. Technical Report UBLCS-97-1, Dept. of Computer Science, University of Bologna, Bologna, Italy, January 1997.
6. A. Basu, B. Charron-Bost, and S. 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, Oct. 1996.
7. 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.

8. M. 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, Aug. 1983.
9. G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *J. ACM*, 32(4):824–840, Oct. 1985.
10. T. D. Chandra, April 1997. Private Communication.
11. T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.
12. T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.
13. S. Chaudhuri. More *choices* allow more *faults*: Set consensus problems in totally asynchronous systems. *Information and Computation*, 105(1):132–158, July 1993.
14. B. Chor, M. Merritt, and D. B. Shmoys. Simple constant-time consensus protocols in realistic failure models. *Journal of the ACM*, 36(3):591–614, 1989.
15. D. Dolev, R. Friedman, I. Keidar, and D. Malkhi. Failure detectors in omission failure environments. Technical Report 96-1608, Department of Computer Science, Cornell University, Ithaca, New York, 1996.
16. D. Dolev, N. A. Lynch, S. S. Pinter, E. W. Stark, and W. E. Weihl. Reaching approximate agreement in the presence of faults. *J. ACM*, 33(3):499–516, July 1986.
17. P. Feldman and S. Micali. An optimal algorithm for synchronous Byzantine agreement. Technical Report MIT/LCS/TM-425, Laboratory for Computer Science, Massachusetts Institute of Technology, June 1990.
18. M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, Apr. 1985.
19. A. Gopal. *Fault-Tolerant Broadcasts and Multicasts: The Problem of Inconsistency and Contamination*. PhD thesis, Cornell University, Jan. 1992.
20. R. 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.
21. R. Guerraoui, M. Larrea, and A. 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.
22. V. Hadzilacos and S. 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.
23. R. Koo and S. Toueg. Effects of message loss on the termination of distributed protocols. *Inf. Process. Lett.*, 27(4):181–188, Apr. 1988.
24. W.-K. Lo and V. 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.
25. N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc., 1996.
26. M. Rabin. Randomized Byzantine generals. In *Proceedings of the 24th Symposium on Foundations of Computer Science*, pages 403–409. IEEE Computer Society Press, Nov. 1983.
27. L. S. Sabel and K. Marzullo. Election vs. consensus in asynchronous systems. Technical Report 95-1488, Department of Computer Science, Cornell University, Ithaca, New York, Febrary 1995.
28. R. van Renesse, April 1997. Private Communication.

This article was processed using the LaTeX macro package with LLNCS style