I R I S A
INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

PUBLICATION
INTERNE
N° 1356

# QUIESCENT UNIFORM RELIABLE BROADCAST AS AN INTRODUCTORY SURVEY TO FAILURE DETECTOR ORACLES

MICHEL RAYNAL

ISSN 1166-8687

IRISA
CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

# Quiescent Uniform Reliable Broadcast as an Introductory Survey to Failure Detector Oracles

Michel Raynal

Thème 1 — Réseaux et systèmes
Projet Adp

Publication interne n ° 1356 — Octobre 2000 — 13 pages

**Abstract:** This paper is a short and informal introduction to failure detector oracles for asynchronous distributed systems prone to process crashes and fair lossy channels. A distributed coordination problem (namely, the implementation of Uniform Reliable Broadcast with a quiescent protocol) is used as a paradigm to visit two types of such oracles. One of them is a "guessing" oracle in the sense that it provides a process with information that the processes could only approximate if they had to compute it. The other is a "hiding" oracle in the sense that it allows to isolate and encapsulate the part of a protocol that has not the required behavioral properties. A quiescent uniform reliable broadcast protocol is described. The guessing oracle is used to ensure the "uniformity" requirement stated in the problem specification. The "hiding" oracle is used to ensure the "quiescence" property required for the protocol behavior.

**Key-words:** Asynchronous Distributed Systems, Failure Detectors, Fair Lossy Channels, Fault-Tolerance, Oracles, Process Crashes, Quiescent Protocol, Uniform Reliable Broadcast.

*(Résumé : tsvp)*

email: raynal@irisa.fr

Centre National de la Recherche Scientifique
(UPRESSA 6074) Université de Rennes 1 – Insa de Rennes

Institut National de Recherche en Informatique
et en Automatique – unité de recherche de Rennes

# Une introduction aux détecteurs de défaillances à l'aide du problème de la diffusion fiable

**Résumé :** Ce rapport constitue une introduction au concept de détecteurs de défaillances introduit par S. Toueg et son équipe. La construction d'un protocole qui ne requiert qu'un nombre fini de messages pour résoudre le problème de la diffusion fiable uniforme sert de fil conducteur et d'illustration à cette introduction.

**Mots clés :** Canaux équitables, détecteur de défaillances, diffusion fiable, oracles, panne franche, protocole finalement silencieux, système réparti asynchrone, tolérance aux défaillances.

# 1 Introduction

One of the most striking and disturbing fact of the fault-tolerant asynchronous distributed computing field is the number of impossibility results that have been stated and proved in the past years [16, 17]. One of the most outstanding of those results is related to the *Consensus* problem. This problem is defined as follows: each process proposes a value and the processes that do not crash have to agree (termination) on the same value which has to be one of the proposed values (safety). It has been shown by Fischer, Lynch and Paterson that this apparently simple problem actually has no deterministic solution as soon as even only one process can crash [10]. This is the famous FLP impossibility result. On the other side, it is also important to note that a characterization of the problems that can be solved in presence of at most one process crash has also been proposed [6].

When a problem cannot be solved in a given model (representing a particular context) several attitudes are possible. One consists in modifying the problem statement in order to get solutions to a close (but "modified") problem. For the consensus problem, this approach has consisted in weakening the termination property and designing probabilistic protocols that solve this modified problem [5]. Another attitude consists in enriching the underlying fault-prone asynchronous distributed system with appropriate oracles in order that the problem becomes solvable in the augmented system.

The *oracle* notion has first been introduced as a *language* whose words can be recognized in one step from a particular state of a Turing machine [11, 15]. The main characteristic of such oracles is to *hide* a sequence of computation steps in a single step, or to *guess* the result of a non-computable function. They have been used to define equivalence classes of problems and hierarchies of problems when they are considered with respect to the assumptions they require to be solved. In our case, the *oracle* notion is related to the detection of failures. These oracles do not change the pattern of failures that affect the execution in which they are used. Their main characteristic is not related to the number of computation steps they hide, but to the guess they provide about failures. Such oracles have been proposed and investigated in the past years. Following their designers (mainly S. Toueg) they are usually called *failure detectors* [3, 2, 7]. A given failure detector oracle is related to a problem (or a class of related problems). Of course, it has to be strong enough to allow to solve the concerned problem, but, maybe more important, it has to be as weak as possible in order to fix the "failure detector" borderline beyond which the problem cannot be solved.

When we consider the consensus problem, several failure detector classes have been defined to solve it [7]. It has also been shown that one of these classes is the weakest that can be used to solve consensus [8]. A failure detector belongs to this class if it satisfies the following two properties. *Completeness*: Eventually, every process that crashes is suspected by every correct process. *Eventual Weak Accuracy*: Eventually, there is a correct process that is not suspected by the correct processes. As we can see, the completeness is on the actual detection of crashes, while the accuracy limits the mistakes a failure detector can make. Several consensus protocols based on this weakest failure detector oracle have been designed [7, 18]. It is important to note that a failure detector satisfying the previous properties cannot be implemented in an asynchronous distributed system prone to process crashes (if it was, it would contradict the FLP impossibility result!). However, a failure detector that does its best to approximate these properties can be built. When the behavior of the underlying system allows it to satisfy the completeness and the eventual accuracy properties during long enough time, the current execution of the consensus protocol can terminate, and consequently the current instance of the consensus problem can be solved.

This paper is an introductory visit to failure detector oracles for asynchronous distributed systems where processes can fail by crashing and links can fail by dropping messages. To do this visit, we consider a distributed computing problem related to distributed coordination, namely the *Uniform Reliable Broadcast* (URB) problem [12]. This is an important problem as it constitutes a basic distributed computing building block. Informally, URB is defined by two primitives (Broadcast and Deliver), such that (1) if a process delivers a message $m$ then all processes that do not crash eventually deliver $m$, and (2) each process that does not crash eventually delivers at least the messages it broadcasts. By interpreting the pair Broadcast/Deliver as This_is_an_order/Execute_it, it is easy to see that URB abstracts a family of distributed coordination problems [3, 13].

Furthermore, in order to fully benefit from the visit, we are interested in solving the URB problem with a *quiescent* protocol. This means that, for each application message $m$ that is broadcast by a process, the protocol eventually stops sending protocol messages. This is a very important property: it guarantees that the network load generated by the calls to the Broadcast primitive remains finite despite process and links failures.

The paper is made up of seven sections. Section2 introduces the underlying system layer and Section 3 reminds a few results related to the net effect of process and links failures. Then, Section 4 defines the URB problem. Section 5 presents a "guessing" and a "hiding" failure detector oracles (that have been introduced for the first time in [3] and [2], respectively). These oracles are then used in Section 6 as underlying building blocks to define a quiescent URB protocol. Section 7 concludes the paper.

## 2 Asynchronous Distributed System Model

The system model consists of a finite set $\Pi$ of processes (namely, $\Pi = \{p_1, \ldots, p_n\}$). They communicate and synchronize by sending and receiving messages through channels. Every pair of processes $p_i$ and $p_j$ is connected by a channel which is denoted $(p_i, p_j)$.

**Processes with crash failures.** A process can fail by *crashing*, i.e., by prematurely halting. A crashed process does not recover. A process behaves correctly (i.e., according to its specification) until it (possibly) crashes. By definition, a *correct* process is a process that never crash. A *faulty* process is a process that is not correct. In the following, $f$ denotes the maximum number of processes that may be faulty ($f \leq n - 1$). There is no assumption on the relative speed of processes.

**Fair lossy channels.** In addition to process crashes, we consider that channels can fail by dropping messages. Nevertheless, they are assumed to be fair lossy. This means that for each channel $(p_i, p_j)$ we have the following properties:

- FLC-Fairness (Termination): If $p_i$ sends a message $m$ to $p_j$ an infinite number of times and $p_j$ is correct, then eventually $p_j$ receives $m$.
- FLC-Validity: If $p_j$ receives a message $m$ from $p_i$, then $p_i$ previously sent $m$ to $p_j$.
- FLC-Integrity: If $p_j$ receives a message $m$ infinitely often from $p_i$, then $p_i$ sends $m$ infinitely often to $p_j$.

It is important to note that (1) there is no a priori assumption on the message transfer delays, and (2) a message can be duplicated a finite number of times. The basic communication primitives used by a process $p_i$ are: *send* () *to* $p_j$, and *receive* () *from* $p_j$.

# 3 A Few Results

**The case of a single channel.** When we consider a system as simple as one made up of two processes connected by a channel, there are some impossibility results related to the effect of process crashes, channel unreliability, or the constraint to use only bounded sequence numbers (see [17] -chapter 22- for an in-depth presentation of these results). Let a reliable channel $c_{rel}$ be a channel such that there is no loss, no duplication, no creation, and no reordering. Let us consider two processors connected by a channel $c$. The aim is to design on top of $c$ a protocol offering a reliable channel $c_{rel}$.
- Let us assume that $c$ is reliable, each processor can crash and recover but has not access to a non-volatile memory. There is no protocol that builds a reliable channel $c_{rel}$ and that tolerates the crash/recovery of the processors [9]. To tolerate it, a non-volatile memory is necessary in order that the processor state can survive crashes.
- Let us assume that the processors cannot crash, and the underlying channel $c$ can duplicate or reorder messages (but it does not create or lose messages). Moreover, only bounded sequence numbers are allowed. It is impossible to design a protocol that implements a reliable channel $c_{rel}$ on top of $c$ [21].
- Let us assume that the underlying channel $c$ can lose and reorder messages but cannot duplicate them. Moreover, the processors do not crash, and only bounded sequence numbers are allowed. There is a protocol that builds $c_{rel}$ on top of $c$, but this protocol is highly inefficient [1].

**Simulation of reliable channels in presence of process crashes.** The effect of lossy channels on the solvability of problems in general is discussed in [4]. Two main results are stated.
- The first concerns a specific class of problems, namely those whose specification does not refer to faulty processes. This is the class of *correct-restricted* problems. An algorithm is provided that transforms any protocol solving a correct-restricted problem and working with process crashes and reliable channels into a protocol working with process crashes and fair lossy links.
- The second result is more general in the sense that it does not consider a particular class of problems. It presents a protocol that, given a system with fair lossy channels and a majority of correct processes, simulates a system with reliable channels. Informally, this shows that a majority of correct processes is powerful enough to cope with message losses when channels are fair.
The two proposed transformations do not provide quiescent protocols.

# 4 Uniform Reliable Broadcast

**Definition.** The *Uniform Reliable Broadcast* problem (URB) is defined in terms of two communication primitives: Broadcast() and Deliver(). When a process issues Broadcast($m$), we say that it "*broadcasts*" $m$. Similarly, when a process issues Deliver($m$), we say that it "*delivers*" $m$. Every broadcast message is unique[1]. This means that if an application process invokes *Broadcast*($m_1$) and *Broadcast*($m_2$) with $m_1$ and $m_2$ having the same content, $m_1$ and $m_2$ are considered as two different messages by the underlying layer.

Uniform Reliable Broadcast is formally defined by the following set of properties [12]:

- URB-Termination: If a correct process *broadcasts* $m$, then any correct process *delivers* $m$ (no messages from correct processes are lost).

[1]This can easily be realized, at the underlying level, by associating with each application message $m$ a pair made up of its sender identity, plus a sequence number.

- URB-Validity: If a process *delivers* $m$, then $m$ has been *broadcast* by some process (no spurious message).
- URB-Integrity: A process *delivers* a message $m$ at most once (no duplication).
- URB-Agreement: If a (correct or not) process *delivers* $m$, then any correct process *delivers* $m$ (no message UR_delivered by a process is missed by a correct process).

The last property is sometimes called "Uniform Agreement". Its non-uniform counterpart would be: "If a correct process *delivers* $m$, then any correct process *delivers* $m$". The Uniformity requirement obliges to also consider the messages delivered by faulty processes. The *Reliable Broadcast* problem is similar to URB except for the Agreement property that is non-uniform.

Let us remark that, differently from the other properties, the URB-Termination property does not apply to faulty processes. This means that the correct processes deliver the same set of messages $S$, and that the set of messages delivered by a faulty process is always a subset of $S$.

**URB with reliable channels.** Figure 1 describes a simple quiescent protocol (defined in [12]) that solves the URB problem in asynchronous distributed systems made up of processes that (1) can crash, and (2) are fully connected by reliable channels (no loss, no duplication, and no creation of messages). To broadcast a message $m$, a process $p_i$ sends it to itself. Then, when a process receives a message $m$ for the first time, it forwards it before delivering it. Consequently, due to channel reliability, it follows that the four URB properties are satisfied.

```
(1) Procedure Broadcast(m):
(2)         send msg(m) to p_i

(3) when msg(m) received from p_k:
(4)         if (first reception of m) then
(5)                 ∀j ≠ i,k do send msg(m) to p_j enddo;
(6)                 Deliver(m)
(7)         endif
```

Figure 1: A Quiescent URB Protocol for Reliable Channels

## 5 Enriching the System Model with Appropriate Oracles

A main difficulty in solving the URB problem in presence of fair lossy links lies in ensuring the URB-Agreement property which states: "If a process delivers a message $m$, then any correct process delivers $m$". This means that a process can deliver a message only when it is sure that this message will eventually be received by each correct process. It has been shown that failure detector oracles are required to overcome this problem [3, 13]. The failure detector (called $\Theta$) described in the next paragraph is an answer to this problem. It has been introduced in [3].

Although $\Theta$ is the weakest failure detector that can be used to ensure the URB-Agreement property [3], its only use is not sufficient to get a quiescent protocol: the broadcast of an application message can still generate an infinite number of network messages. Actually, ensuring the quiescence property requires that a process $p_i$ be able to know if another process $p_j$ is still alive: if $p_j$ is not, $p_i$ can stop sending messages to $p_j$ even if the last message it sent to it has not yet been acknowledged.

Several failure detectors can be designed to allow a process $p_i$ to get this information. Some (as $\Theta$) provide outputs with bounded size. Others provide outputs whose size is not bounded. It has been shown that the failure detector oracles of the first category cannot be implemented [7], while some of the second category can be. Hence, in the following we present an oracle of the second category called *Heartbeat* that can be implemented. This oracle has been introduced in [2].

### 5.1 A Guessing Failure Detector Oracle: $\Theta$

This failure detector is defined by the following properties [3]. Each process $p_i$ is endowed with a local variable $\text{TRUSTED}_i$ whose aim is to contain identities of processes that are currently perceived as non-crashed by $p_i$ (this variable is updated by $\Theta$ and read by $p_i$). The failure detector $\Theta$ ensures that these variables satisfy the following properties:

- $\Theta$-Completeness: There is a time after which, for any process $p_i$, $\text{TRUSTED}_i$ does not include faulty processes.
- $\Theta$-Accuracy: At every time, for any process $p_i$, $\text{TRUSTED}_i$ includes at least one correct process. (Note that the correct process trusted by $p_i$ is allowed to change over time.)

In the general case ($f < n$), the $\Theta$ oracle cannot be implemented in an asynchronous distributed system. That is why we place it in the family of "guessing" failure detector oracles. Differently, when the system satisfies the additional assumption $f < n/2$, it can be implemented (such an implementation is described in [3]).

### 5.2 A Hiding Failure Detector Oracle: *Heartbeat*

The *Heartbeat* failure detector oracle provides each process $p_i$ with an array of counters $\text{HB}_i[1..n]$ (initialized to $[0, \ldots, 0]$) such that:

- HB-Completeness: For each process $p_i$, $\text{HB}_i[j]$ stops increasing if $p_j$ is faulty.
- HB-Accuracy: $\text{HB}_i[j]$ never decreases, and $\text{HB}_i[j]$ never stops increasing if $p_i$ and $p_j$ are correct.

A Hearbeat failure detector can be easily implemented, e.g., by requiring each process to periodically send "I am alive" messages. This implementation entails the sending of an infinite number of messages by each correct process: it is not quiescent. That is the reason why we place it in the family of "hiding" failure detector oracles. A set of modules (one per process) realizing a Hearbeat oracle can be used to encapsulate and isolate the non-quiescent part of a protocol and thereby hides its undesirable behaviors.

## 6 A Protocol

### 6.1 Description of the Protocol

A quiescent URB protocol is described in Figure 2 for a process $p_i$. It is based on the previously described failure detectors and the classical acknowledgement mechanism. An important local data managed by a process $p_i$ is $rec\_by_i[m]$ which records the processes that, to $p_i$'s knowledge, have received a copy of the application message $m$. The protocol uses two types of messages, tagged "msg" and "ack", respectively. They are called *protocol messages*, to distinguish them from the

messages broadcast by the application. Each protocol message tagged "msg" carries an application message, while one tagged "ack" carries only the identity of an application message.

To broadcast an application message $m$, a process $p_i$ sends a protocol message tagged "msg" and including $m$ to itself (line 2). When, it receives a protocol message carrying an application message $m$ for the first time (line 12), a process $p_i$ activates the task *Diffuse*($m$) which repeatedly (lines 5-10) sends $m$ to the processes that, from $p_i$'s point of view, have no copy of $m$ and are alive. The Heartbeat failure detector is used by $p_i$ to know which processes are locally perceived as being alive. It is important to note that, as soon as the test at line 7 remains permanently false for all $j$, then $p_i$ stops sending messages (but does not necessarily terminate as it can keep on executing the loop if the condition of line 10 remains false[2]). Each time $p_i$ receives a protocol message tagged "msg", it sends back an "ack" message to inform the sender that it has got a copy of $m$ (line 16). When a process receives an "ack" message, it updates accordingly the local data $rec\_by_i[m]$ (line 18).

Finally, if $p_i$ has not yet delivered an application message $m$, it does it as soon as it knows that at least one correct process got it (condition $\text{TRUSTED}_i \subseteq rec\_by_i[m]$ at line 19).

```
(1) Procedure Broadcast(m):
(2)         send msg(m) to p_i

(3) Task Diffuse(m):
(4)         prev_hb_i[m] ← [−1, ... , −1];
(5)         repeat periodically
(6)                 cur_hb_i ← HB_i;
(7)                 ∀j ≠ i: if ((prev_hb_i[m][j] < cur_hb_i[j]) ∧ (j ∉ rec_by_i[m]))
(8)                         then send msg(m) to p_j endif;
(9)                 prev_hb_i[m] ← cur_hb_i;
(10)        until   (∀j ∈ [1..n] : (j ∈ rec_by_i[m]) ) endrepeat

(11) when msg(m) is received from p_k:
(12)        if (first reception of m)
(13)                then rec_by_i[m] ← {i, k};
(14)                     activate task Diffuse(m)
(15)                else  rec_by_i[m] ← rec_by_i[m] ∪ {k} endif;
(16)        send ack(m) to p_k

(17) when ack(m) is received from p_k:
(18)        rec_by_i[m] ← rec_by_i[m] ∪ {k}

(19) when ((p_i has not yet delivered m) ∧ (TRUSTED_i ⊆ rec_by_i[m]))
(20)        do Deliver(m) enddo
```

Figure 2: A Quiescent Uniform Reliable Broadcast Protocol

[2]It is important not to confuse a *quiescent* protocol and a *terminating* protocol. Ensuring termination requires stronger failure detector oracles, namely, oracles that allow to know exactly which processes have crashed and which have not [14].

### 6.2 Proof

The proof that the protocol described in Figure 2 satisfies URB-Integrity (no duplication of a broadcast message) and URB-Validity (no creation of application messages) are left to the reader. The proof has the same structure as the proof given in [3].

**Lemma 1** *If a correct process starts Diffuse($m$), eventually all correct processes start Diffuse($m$).*

**Proof** Let us first observe that if the identity $k$ belongs to $rec\_by_i[m]$, this is because $p_i$ received msg($m$) or ack($m$) from $p_k$ and updated consequently $rec\_by_i[m]$ at line 13, 15 or 18, from which we conclude that $p_k$ has a copy of $m$.

Let us consider a correct process $p_i$ that starts Diffuse($m$). It launches this task at line 14 when it receives $m$ for the first time. Let $p_j$ be a correct process. As $p_j$ is correct, $\text{HB}_i[j]$ keeps on increasing and the subcondition ($prev\_hb_i[m][j] < cur\_hb_i[j]$) is infinitely often true. We consider two cases:
- Case ($j \in rec\_by_i[m]$). In that case, due to the previous observation, $p_j$ has a copy of $m$. We conclude from the protocol text, that $p_j$ started *Diffuse*($m$) when it received $m$ for the first time.
- Case ($j \notin rec\_by_i[m]$). In that case $p_i$ keeps on sending copies of $m$ to $p_j$ (at line 8). Due to the FLC-Fairness property of the channel $(p_i, p_j)$, $p_j$ eventually receives $m$ from $p_i$ and, if not yet done, starts *Diffuse*($m$). $\square_{Lemma\ 1}$

**Lemma 2** *If all correct processes start Diffuse($m$), they eventually execute* Deliver($m$).

**Proof** Let us assume that all the correct processes execute *Diffuse*($m$) and let $p_i$ and $p_j$ be two correct processes. So, $p_i$ sends $m$ to $p_j$ until it knows that $m$ has been received by $p_j$ (i.e., until $j \in rec\_by_i[m]$). Due to the acknowledgment mechanism and the FLC-Fairness property of the underlying channels, this eventually occurs. It follows that, for each correct process $p_i$, $rec\_by_i[m]$ eventually includes all correct processes.

Let us now consider the set $\text{TRUSTED}_i$. Due to the HB-Completeness property of the Heartbeat failure detector, $\text{TRUSTED}_i$ eventually does not include faulty processes. It follows that the condition ($\text{TRUSTED}_i \subseteq rec\_by_i[m]$) eventually becomes true, and then $p_i$ executes Deliver($m$). $\square_{Lemma\ 2}$

**Theorem 1** URB-Termination. *If a correct process executes* Broadcast($m$), *then all correct processes execute* Deliver($m$).

**Proof** If a correct process $p_i$ executes Broadcast($m$), it sends msg($m$) to itself (line 2) and consequently starts the task *Diffuse*($m$) (lines 12-14). Then, due to Lemma 1, all correct processes start *Diffuse*($m$), and due to Lemma 2, they all execute Deliver($m$). $\square_{Theorem\ 1}$

**Theorem 2** URB-Agreement. *If a process executes* Deliver($m$), *then all correct processes execute* Deliver($m$).

**Proof** If a (correct or not) process $p_i$ executes Deliver($m$), then the condition ($\text{TRUSTED}_i \subseteq rec\_by_i[m]$) was satisfied just before it executes it. Due to the HB-Accuracy property of the Heartbeat failure detector, $\text{TRUSTED}_i$ includes at least one correct process $p_j$. Hence, $p_j \in rec\_by_i[m]$, from which we conclude that there is at least one correct process that received $m$ (at line 11). As $p_j$ is correct, it started the task *Diffuse*($m$) when it received $m$ for the first time. It then follows from Lemmas 1 and 2 that each correct process executes Deliver($m$). $\square_{Theorem\ 2}$

**Theorem 3** Quiescence. *Each invocation of* Broadcast($m$) *gives rise to a finite number of protocol messages.*

**Proof** Let us observe that the reception of an "ack" protocol message never entails the sending of a protocol message. It follows that we only have to show that, for any application message $m$, eventually no process sends protocol messages of the form msg($m$).

A msg($m$) protocol message is sent at line 8 by the task *Diffuse*($m$). So, we have to show that any process $p_i$ eventually stops executing line 8. This is trivial if $p_i$ crashes. So, let us consider that $p_i$ is correct. There are two cases according to the destination process $p_j$:
- Case 1: $p_j$ is faulty. Then due to the HB-Completeness, $\text{HB}_i[j]$ eventually stops increasing, and from then on we permanently have $prev\_hb_i[m][j] = cur\_hb_i[j] = \text{HB}_i[j]$, from which we conclude that $p_i$ stops sending messages to $p_j$.
- Case 2: $p_j$ is correct. In that case the subcondition ($prev\_hb_i[m][j] < cur\_hb_i[j]$) is infinitely often true. So, let us consider the second subcondition, namely, ($j \notin rec\_by_i[m]$). Let us assume that the subcondition ($j \in rec\_by_i[m]$) is never satisfied. We show a contradiction.
If ($j \in rec\_by_i[m]$) is never satisfied, it follows that $p_i$ sends an infinite number of protocol messages msg($m$) to $p_j$. Due to the FLC-Fairness property of the channel $(p_i, p_j)$, $p_j$ eventually receives an infinite number of copies of $m$. Each time it receives msg($m$), $p_j$ sent back ack($m$) to $p_i$ (line 16). It then follows that, due to FLC-Fairness property of the channel $(p_j, p_i)$, $p_i$ receives an ack($m$) protocol message from $p_j$. At the first reception of such a protocol message, $p_i$ includes $j$ in $rec\_by_i[m]$ (line 18). Finally, let us note that a process identity is never removed from $rec\_by_i[m]$. So from now on, the condition ($j \in rec\_by_i[m]$) remains permanently true. A contradiction. $\square_{Theorem\ 3}$

### 6.3 Favoring Early Quiescence

This guideline in the design of the protocol described in Figure 2 was simplicity. It is possible to improve the protocol by allowing early quiescence (in some cases, this can also reduce the number of protocol messages that are exchanged). To favor early quiescence, the variable $rec\_by_i[m]$ of each process has to be updated as soon as possible. This can be done in the following way:
- (1) add the current content of $rec\_by_i[m]$ to each protocol message sent by a process $p_i$;
- (2) add to each "ack" message the corresponding application message (instead of only its identity);
- (3) send "ack" messages to all the processes (instead of only the sender of the corresponding "msg" message).

The resulting protocol is described in Figure 3. Its improved behavior is obtained at the price of bigger protocol messages. Its proof is similar to the proof of Section 6.2. The main difference lies in the way it is proved that $j \in rec\_by_i[m]$ means that $p_j$ has got a copy of $m$.

### 6.4 Strong Uniform Reliable Broadcast

The URB-Termination property of the URB problem is only on the correct processes. Said another way, a message broadcast by a process that crashes (either during the broadcast or even later) is not required to be delivered. Its actual delivery depends on the system behavior. This can be a drawback for some applications. So, let us define *Strong Uniform Reliable Broadcast* (S_URB). We define this communication service as being similar to URB except for the termination property which is:

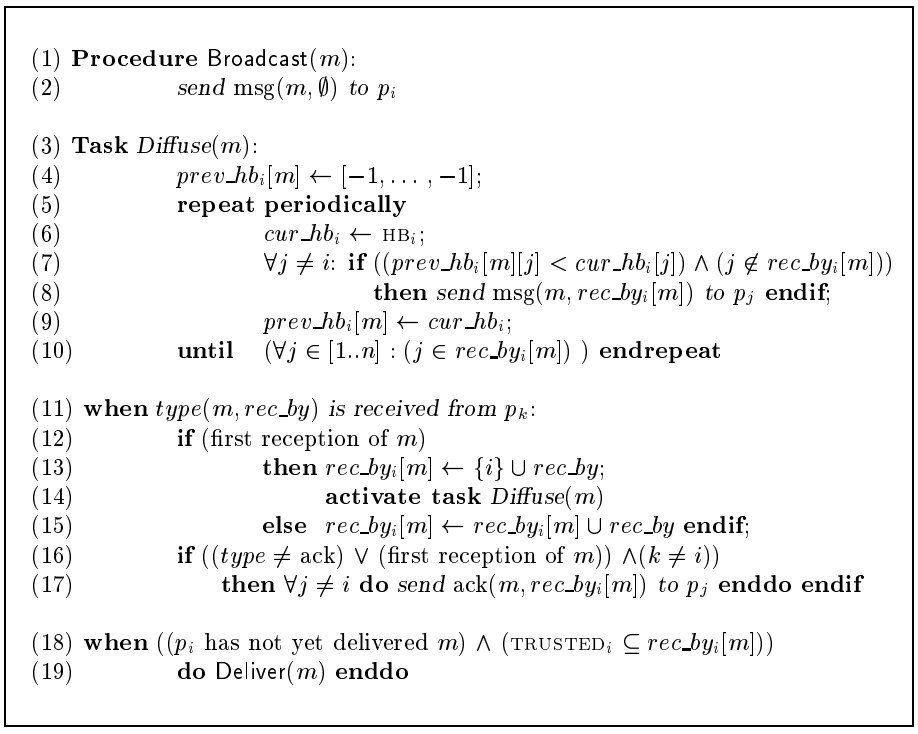


Figure 3: An Improved Quiescent URB Protocol

- S_URB-Termination: If a process completes the execution of Broadcast($m$), then a correct process delivers $m$. (This means that, whatever the future behavior of its sender, no message that has been broadcast is lost).

We conclude from the combination of the URB-Agreement and S_URB-Termination properties that each correct process delivers all the messages whose broadcasts have been completed.

The S_URB-Termination property can easily be implemented. When we consider Figure 2, only a very simple modification of the procedure Broadcast($m$) is required. Namely, the only statement **wait** (TRUSTED$_i \subseteq rec_by_i[m]$) has to be added at the end of this procedure. It ensures that when a broadcast completes, the corresponding application message is known by at least one correct process (that will disseminate it in its *Diffuse* task).

# 7 Conclusion

Failure detector oracles are becoming a fundamental issue in the design of fault-tolerant distributed applications designed to run on fault-prone distributed systems. The aim of this paper was to provide a simple introduction to their philosophy and to illustrate it with some of them, namely, a "guessing" and a "hiding" failure detector oracles.

The design of a quiescent protocol solving the Uniform Reliable Broadcast problem has been used as a paradigm to show why failure detector oracles are required and how they can be used. The guideline for the design of this protocol was simplicity (as we have seen, more efficient protocols can be designed).

The reader interested in more details on the concept of failure detector oracles, the problems they can help to solve, and their uses, can consult [2, 3, 7, 8, 13, 14, 18, 19, 20, 22].

# References


[1] Afek Y., Attiya H., Fekete A.D., Fischer M., Lynch N., Mansour Y., Wang D. and Zuck L., Reliable Communication over Unreliable Channels. *Journal of the ACM*, 41(6):1267-1297, 1994.

[2] Aguilera M.K., Chen W. and Toueg S., On Quiescent Reliable Communication. *SIAM Journal of Computing*, 29(6):2040-2073, 2000.

[3] Aguilera M.K., Toueg S. and Deianov B., Revisiting the Weakest Failure Detector for Uniform Reliable Broadcast. *Proc. 13th Int. Symposium on DIStributed Computing (DISC'99)*, Springer -Verlag LNCS #1693, pp. 21-34, Bratislava (Slovaquia), 1999.

[4] Basu A., Charron-Bost B. and Toueg S., Simulating Reliable Links with Unreliable Links in the Presence of Process Crashes. *Proc. 10th Int. Workshop on Distributed Algorithms (now, DISC)*, Springer -Verlag LNCS #1051, pp. 105-121, , Bologna (Italy), 1996.

[5] Ben-Or M., Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols. *Proc. 2nd ACM Symposium on Principles of Distributed Computing (PODC'83)*, ACM Press, pp. 27-30, Montréal (Canada), 1983.

[6] Biran O., Moran S. and Zaks S., A Combinatorial Characterization of the Distributed 1-Solvable Tasks. *Journal of Algorithms*, 11:420-440, 1990.

[7] Chandra T. and Toueg S., Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225-267, 1996.

[8] Chandra T., Hadzilacos V. and Toueg S., The Weakest Failure Detector for Solving Consensus. *Journal of the ACM*, 43(4):685–722, July 1996.

[9] Fekete A.D., Lynch N., Mansour Y. and Spinelli J., The Impossibility of Implementing Reliable Communication in Face of Crashes. *Journal of the ACM*, 40(5):1087-1107, 1993.

[10] Fischer M.J., Lynch N. and Paterson M.S., Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374–382, 1985.

[11] Garey M.R. and Johnson D.S., *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman W.H. & Co, New York, 340 pages, 1979.

[12] Hadzilacos V. and Toueg S., Reliable Broadcast and Related Problems. In *Distributed Systems*, ACM Press (S. Mullender Ed.), New-York, pp. 97-145, 1993.

[13] Halpern J.Y. and Ricciardi A., A Knowledge-Theoretic Analysis of Uniform Distributed Coordination and Failure Detectors. *Proc. 18th ACM Symposium on Principles of Distributed Computing (PODC'99)*, pp. 73-82, Atlanta (GA), 1999.

[14] Hélary J.-M., Hurfin M., Mostefaoui A., Raynal M. and Tronel F., Computing Global Functions in Asynchronous Distributed Systems with Perfect Failure Detectors. *IEEE Transactions on Parallel and Distributed Systems*, 11(9), 2000.

[15] Hopcroft J.E. and Ullman J.D. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, Reading (MA), 418 pages, 1979.

[16] Lynch N., A Hundred Impossibility Proofs for Distributed Computing. *Invited Talk, Proc. 8th ACM Symposium on Principles of Distributed Computing (PODC'89)*, ACM Press, pp. 1-27, Edmonton (Canada), 1989.

[17] Lynch N., *Distributed Algorithms*. Morgan Kaufmann Pub., San Francisco (CA), 872 pages, 1996.

[18] Mostefaoui A. and Raynal M., Solving Consensus Using Chandra-Toueg's Unreliable Failure Detectors: a General Quorum-Based Approach. *Proc. 13th Symposium on DIstributed Computing (DISC'99)*, Springer Verlag LNCS #1693, Bratislava (Slovakia), pp. 49-63, 1999.

[19] Mostefaoui A. and Raynal M., $k$-Set Agreement with Limited Accuracy Failure Detectors. *Proc. 19th ACM Symposium on Principles of Distributed Computing (PODC'00)*, Portland (OR), pp. 143-152, 2000.

[20] Raynal M. and Tronel F., Restricted Failure Detectors: Definition and Reduction Protocols. *Information Processing Letters*, 72:9197, 1999.

[21] Wang D.-W. and Zuck L.D., Tight Bounds for the Sequence Transmission Problem. *Proc. 8th ACM Symposium on Principles of Distributed Computing (PODC'89)*, ACM Press, pp. 73-83, Edmonton (Canada), 1989.

[22] Yang J., Neiger G. and Gafni E., Structured Derivations of Consensus Algorithms for Failure Detectors. *Proc. 17th ACM Symposium on Principles of Distributed Computing (PODC'98)*, Puerto Vallarta (Mexico), pp.297-308, 1998.