On the Weakest Failure Detector for Uniform Reliable Broadcast

Marcos Kawazoe Aguilera Sam Toueg Borislav Deianov

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

April 30, 1999

Abstract

Uniform Reliable Broadcast (URB) is a communication primitive that requires that if a process delivers
a message, then all correct processes also deliver this message. A recent PODC paper [HR99] uses
Knowledge Theory to determine what failure detectors are necessary to implement this primitive in
asynchronous systems with process crashes and lossy links that are fair. In this paper, we revisit this
problem using a different approach, and provide a result that is simpler, more intuitive, and, in a precise
sense, more general.

1 Introduction

Uniform Reliable Broadcast (URB) is a communication primitive that requires that if a process delivers
a message, then all correct processes also deliver this message [HT94]. A recent PODC paper [HR99]
uses Knowledge Theory to determine what failure detectors are necessary to implement this primitive in
asynchronous systems with process crashes and fair links (roughly speaking, a fair link may lose an infinite
number messages, but if a message is sent infinitely often then it is eventually received).

[HR99] considered systems where up to $f$ process may crash and links are fair, and used Knowledge Theory
to show that solving URB in such a system requires a generalized $f$-useful failure detector (denoted $G^f$ in
here). Such a failure detector is parameterized by $f$ and is described in Figure 1. [HR99] shows that when
$f = n$ or $f = n - 1$, $G^f$ is equivalent to a perfect failure detector.

In this paper, we revisit this problem using the approach in [CHT96], and provide a result that is simpler,
more intuitive, and, in a precise sense, more general, as we now explain.

[HR99] actually studies a problem called Uniform Distributed Coordination. This problem, however, is isomorphic to URB:
init and do in Uniform Distributed Coordination correspond to broadcast and deliver in URB, respectively.
A generalized failure detector [HR99] outputs a pair \((S, k)\) where \(S\) is a subset of processes and \(k\) is a positive integer. Intuitively, the failure detector outputs \((S, k)\) to report that \(k\) processes in \(S\) are faulty. In a run \(r\), the failure detector event \(\text{suspect}_p(S, k)\) is said to be \(f\)-useful for \(r\) if (a) \(S\) contains all processes that crash in \(r\), and (b) \(n - |S| > \min(f, n - 1) - k\). A generalized failure detector is \(f\)-useful if, for all runs \(r\) and processes \(p\), the following two properties hold (where \(r_p(t)\) denotes the prefix of run \(r\) at process \(p\) up to time \(t\)):

- If \(\text{suspect}_p(S, k)\) is in \(r_p(t)\) then there is a subset \(S' \subseteq S\) such that \(|S'| = k\) and for all \(q \in S'\), we have that \(\text{crash}_q\) is in \(r_q(t)\).
- If \(p\) is correct, then there is a \(f\)-useful failure-detector event for \(r\) in \(r_p(t)\), for some \(t\).

Figure 1: Definition of a generalized \(f\)-useful failure detectors.

prove that the weakest failure detector for this problem is a simple failure detector denoted \(\Theta\). \(\Theta\) outputs a set of processes that are currently trusted to be up,\(^2\) such that:

**Completeness:** There is a time after which correct processes do not trust any process that crashes.

**Accuracy:** If there is a correct process then, at every time, every process trusts at least one correct process.

This simple characterization of the weakest failure detector for URB is more general than the one given in [HR99], in the sense that it holds for any system with fair links, regardless of \(f\) or any other types of restrictions or dependencies on process crashes.\(^3\) To illustrate this point, consider the following three systems with \(n\) processors \(\{p_1, p_2, \ldots, p_n\}\):

1. In system \(S_1\), every processor may crash, except that we assume that \(p_1\) and \(p_2\) cannot both crash in the same run (this assumption makes sense if, for example, \(p_1\) and \(p_2\) are configured as symmetric primary/backup servers). Note that in \(S_1\), up to \(f = n - 1\) processors may crash in the same run.

2. In system \(S_2\), every processor may crash, except that processor \(p_1\) is a fault-tolerant highly-available computing server that crashes only when it is left alone in the system (this assumption is not unreasonable: in some existing systems, processes kill themselves if they are unable to communicate with a minimum number of processes). Note that in \(S_2\), up to \(f = n\) processors may crash in the same run.

3. In system \(S_3\), the number of processes that crash is bounded, but this bound \(f\) is not known. Moreover, there are some additional restrictions and dependencies on process crashes (e.g., if more than half of the processes crash then a certain process \(p_1\) commits suicide) but these are also not known.

What is the weakest failure detector for solving URB in each of \(S_1\), \(S_2\) and \(S_3\)? By our result, the answer is simply \(\Theta\).

In contrast, the result in [HR99] cannot be applied to \(S_1\), \(S_2\) and \(S_3\), as we now explain. For \(S_3\), this is obvious because \(f\) is not even known. For \(S_1\), the value of \(f\), namely \(n - 1\), is known. So, one may be

\(^2\)Some failure detectors in the literature output a set of processes suspected to be down; this is just the complement of the set of processes that are trusted to be up.

\(^3\)If one assumes that a majority of processes does not crash, then URB can be solved without any failure detector [BCBT96]. As we explain in Section 11, this does not contradict our result.
tempted to naively plug $f = n - 1$ in the result of [HR99], and to conclude that solving URB in $S_1$ requires $G^{n-1}$ (i.e., a perfect failure detector). This conclusion is incorrect, because [HR99] explicitly assumes that any subset of up to $f = n - 1$ processors can crash in a run — an assumption that does not hold for $S_1$. Similarly, for $S_2$, one cannot just plug $f = n$ in [HR99] to obtain the correct answer.

Since, in some sense, both $G^f$ and $\Theta$ are “minimal” for URB, an important question is now in order: What is the relation between $G^f$ and $\Theta$? To answer this question, we introduce the notions of failure patterns and environments [CHT96]. Roughly speaking, a failure pattern indicates, for each process $p$, whether $p$ crashes and, if so, when. An environment $E$ is a set of failure patterns; and a system with environment $E$ is one where the process crashes must match one of the failure patterns in $E$. Intuitively, environments allow us to express restrictions on process crashes, such as “either $p_1$ or $p_2$, but not both, may crash” (so environments can be used to formally define the systems $S_1$ and $S_2$ described earlier). A commonly-used environment in the literature is $E^f$, the set of all failure patterns in which at most $f$ processes crash: A system with environment $E^f$ allows up to $f$ process crashes, but there are no other constrains or dependencies, i.e., any subset of $f$ processes may crash, and these crashes can occur at any time.

We can now compare $G^f$ and $\Theta$. Roughly speaking, $\Theta$ is the weakest failure detector regardless of the environment $E$, while $G^f$ is necessary and sufficient for environment $E^f$. When $E = E^f$, there is an algorithm that transforms $G^f$ into $\Theta$, and so $\Theta$ is at least as weak as $G^f$ in environment $E^f$.

An important difference between [HR99] and this paper is that [HR99] uses Knowledge Theory [FHMV95] to establish and state its results, while we use algorithmic reductions [CT96]. An advantage of the algorithmic reduction method over the knowledge approach, is that the former allows the derivation of a stronger result: in a nutshell, the knowledge approach determines only what information about failures processes know, while the algorithmic reduction method determines what information about failures processes know and can effectively compute. Specifically, the result in [HR99] is that, in order to solve URB, processes must know the information provided by $G^f$. This does not automatically imply that processes can actually compute $G^f$.\footnote{This is modulo a technicality due to a difference in the two models: in [HR99] all the failure detector events are “seen” by processes, while here processes can “miss” some failure detector values.}

In contrast, the algorithmic reduction given in this paper shows that if processes can solve URB with some failure detector $D$, then they can use $D$ to compute failure detector $\Theta$. This reduction implies that $D$ is at least as strong as $\Theta$ in terms of problem solving: if processes can solve a problem with $\Theta$, they can also solve it with $D$ (by first using $D$ to compute $\Theta$). Note we would not be able to say that $D$ is at least as strong as $\Theta$ (in terms of problem solving) if $D$ only allowed processes to know (but not compute) $\Theta$.

Finally, there is another difference between our approach and the one in [HR99], namely, the universe of failure detectors that is being considered. To understand the meaning of a statement such as “$D$ is the weakest failure detector...”, or “$D$ is necessary...”, one needs to know the universe of failure detectors under consideration (because it is among these failure detectors that $D$ is the “weakest” or “necessary”). In our paper, the universe of failure detectors is explicit and clear: a failure detector is a function of the failure pattern — a natural definition that is widely used [CHT96, HMR97, OGS97, YNG98, LH94]. The universe of failure detectors in [HR99], however, is implicitly defined, and the exact nature and power of the failure detectors considered are not entirely clear. This issue is further discussed in Section 8.

In summary, in this paper we consider the problem of determining the weakest failure detector for solving URB in systems with process crashes and lossy links — a problem that was first investigated in [HR99].\footnote{In Knowledge Theory, processes may know facts that they cannot actually compute. For example, if the system is sufficiently expressive, they know the answer to every unsolved problem in Number Theory, and they also know whether any given Turing Machine halts on blank tape.}
In [HR99], this problem was studied using the framework of Knowledge Theory. In this paper, we tackle this problem using a different approach based on the standard failure detector models and techniques of [CHT96]. The results that we obtain are simple, intuitive and general. More precisely:

1. We provide a single failure detector $\Theta$, and show that it is the weakest failure detector for URB, in any environment. In particular, our result holds even if $f$ is not known.

   In environment $E_f$, $\Theta$ is at least as weak as $G_f$.

2. $\Theta$ is simple and a natural candidate for solving URB. As a result, the algorithm that uses $\Theta$ to solve URB in any environment $E$, is immediate.

3. Our results are derived and can be understood from first principles (they do not require Knowledge Theory).

4. Our “minimality” result is in term of effective computation, not knowledge: roughly speaking, if processes can solve URB, we show how they can compute $\Theta$ (this implies knowledge of $\Theta$; but the converse does not necessarily hold).

5. The universe of failure detectors (with respect to which our minimality result hold) is given explicitly through a simple definition.

The paper is organized as follows. Our model is described in Section 2. In Section 3, we explain what it means for a failure detector to be weaker than another one. Section 4 defines the uniform reliable broadcast problem. Failure detector $\Theta$ is defined in Section 5, and in Section 6, we show how to use it to implement uniform reliable broadcast in systems with process crashes and fair links. In Section 7 we show that $\Theta$ is actually the weakest failure detector for this problem. In Section 8, we briefly discuss the nature and power of failure detectors, and in Section 9 we consider the relation between $G_f$ and $\Theta$. Related work is discussed in Section 10 and we conclude the paper in Section 11.

2 Model

Throughout this paper, in all our results, 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. The system consists of a set of $n$ processes $\Pi = \{1, 2, \ldots, n\}$ that are completely connected by point-to-point (bidirectional) links. The system can experience both process failures and link failures. Processes can fail by crashing, and links can fail by dropping messages. The model, based on the one in [CHT96], is described next.

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

2.1 Failure Patterns and Environments

Processes can fail by crashing, i.e., by halting prematurely. A failure pattern $F$ is a function from $\mathcal{T}$ to $\Pi^n$. Intuitively, $F(t)$ 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 $\text{crashed}(F) = \bigcup_{t \in \mathcal{T}} F(t)$ and
correct\((F) = \Pi \setminus \text{crashed}(F)\). If \(p \in \text{crashed}(F)\) we say \(p\) \textit{crashes (or is faulty) in } F \textit{ and if } p \in \text{correct}(F)\) we say \(p\) \textit{is correct in } F \textit{.}

An environment \(E\) is a set of failure patterns. As we explained in the introduction, environments describe the crashes that can occur in a system.

Links can fail by dropping messages, but we assume that links are \textit{fair}. Roughly speaking, a fair link from \(p\) to \(q\) may intermittently drop messages, and may do so infinitely often, but it must satisfy the following “fairness” property: if \(p\) repeatedly sends some message to \(q\) and \(q\) does not crash, then \(q\) eventually receives that message. This is made more precise in Section 2.3.

2.2 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 \textit{failure detector history} \(H\) with range \(\mathcal{R}\) is a function from \(\Pi \times T\) to \(\mathcal{R}\). \(H(p, t)\) is the output value of the failure detector module of process \(p\) at time \(t\). A \textit{failure detector} \(D\) is a function that maps each failure pattern \(F\) to a non-empty set of failure detector histories with range \(\mathcal{R}_D\) (where \(\mathcal{R}_D\) denotes the range of the failure detector output of \(D\)). \(D(F)\) denotes the set of possible failure detector histories permitted by \(D\) for the failure pattern \(F\).

2.3 Runs of Algorithms

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

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

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

A run must also satisfy the following “fair link properties” for every pair of processes \(p\) and \(q\):

- \textit{Fairness}: If \(p\) sends a message \(m\) to \(q\) an infinite number of times and \(q\) is correct, then \(q\) eventually receives \(m\) from \(p\).
- \textit{Uniform Integrity}: If \(q\) receives a message \(m\) from \(p\) then \(p\) previously sent \(m\) to \(q\); and if \(q\) receives \(m\) infinitely often from \(p\), then \(p\) sends \(m\) infinitely often to \(q\).

3 Failure Detector Transformations

As explained in [CT96, CHT96], failure detectors can be compared via algorithmic transformations. We now explain what it means for an algorithm \(T_{D \rightarrow D'}\) to transform a failure detector \(D\) into another failure
detector $D'$ in an environment $E$. Algorithm $T_{D \rightarrow D'}$ uses $D$ to maintain a variable $D'_p$ at every process $p$. This variable, reflected in the local state of $p$, emulates the output of $D'$ at $p$. Let $H_{D'}$ be the history of all the $D'$ variables in a run $R$ of $T_{D \rightarrow D'}$, i.e., $H_{D'}(p, t)$ is the value of $D'_p$ at time $t$ in run $R$. Algorithm $T_{D \rightarrow D'}$ transforms $D$ into $D'$ in $E$ if and only if for every $F \in E$ and every run $R = (F, H_D, I, S, T)$ of $T_{D \rightarrow D'}$ using $D$, we have $H_{D'} \in D'(F)$. Intuitively, since $T_{D \rightarrow D'}$ is able to use $D$ to emulate $D'$, $D$ provides at least as much information about process failures as $D'$ does, and we say that $D'$ is weaker than $D$ in $E$.

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

4 Uniform Reliable Broadcast

Uniform Reliable Broadcast (URB) is defined in terms of two primitives: $\text{broadcast}(m)$ and $\text{deliver}(m)$. We say that process $p$ broadcasts message $m$ if $p$ invokes $\text{broadcast}(m)$. We assume that every broadcast message $m$ includes the following fields: the identity of its sender, denoted $\text{sender}(m)$, and a sequence number, denoted $\text{seq}(m)$. These fields make every message unique. We say that $q$ delivers message $m$ if $q$ returns from the invocation of $\text{deliver}(m)$. Primitives $\text{broadcast}$ and $\text{deliver}$ satisfy the following properties [HT94]:

- **Validity**: If a correct process broadcasts a message $m$, then it eventually delivers $m$.
- **Uniform Agreement**: If some 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 $\text{sender}(m)$.

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

5 Failure Detector $\Theta$

We now define failure detector $\Theta$. Each failure detector module of $\Theta$ outputs a set of processes that are trusted to be up, i.e., $R_{\Theta} = 2^\Pi$. For each failure pattern $F$, $\Theta(F)$ is the set of all failure detector histories $H$ with range $R_{\Theta}$ that satisfy the following properties:

- **$\Theta$-completeness**: There is a time after which correct processes do not trust any process that crashes. More precisely:
  $$\exists t \in T, \forall p \in \text{correct}(F), \forall q \in \text{crashed}(F), \forall t' \geq t : q \notin H(p, t')$$

- **$\Theta$-accuracy**: If there is a correct process then, at every time, every process trusts at least one correct process. More precisely:
  $$\text{crashed}(F) \neq \Pi \Rightarrow \forall t \in T, \forall p \in \Pi \setminus F(t), \exists q \in \text{correct}(F) : q \in H(p, t)$$
For every process $p$:

To execute $\text{broadcast}(m)$:

$\text{got}[m] \leftarrow \{p\}$

fork task $\text{diffuse}(m)$

return

task $\text{diffuse}(m)$:

while true do

send $m$ to all processes

$d \leftarrow D_p$ \{ $d$ is the list of processes trusted to be up \}

if $d \subseteq \text{got}[m]$ and $p$ has not delivered $m$

then deliver($m$)

upon receive $m$ from $q$ do

if task $\text{diffuse}(m)$ has not been started yet then

$\text{got}[m] \leftarrow \{p,q\}$

fork task $\text{diffuse}(m)$

else $\text{got}[m] \leftarrow \text{got}[m] \cup \{q\}$

Figure 2: Implementing Uniform Reliable Broadcast using $D = \Theta$

Note that a process may be trusted even if it has actually crashed. Moreover, the correct processes trusted by a process $p$ is allowed to change over time (in fact, it can change infinitely often), and it is not necessarily the same as the correct process trusted by another process $q$.

6 Using $\Theta$ to Implement Uniform Reliable Broadcast

The algorithm that implements URB using $\Theta$ is shown in Figure 2. When ambiguities may arise, a variable local to process $p$ is subscripted by $p$. To broadcast a message $m$, a process $p$ first initializes $\text{got}_p[m] = \{p\}$; this variable represents the processes that $p$ knows to have received $m$ so far. Process $p$ then forks task $\text{diffuse}(m)$. In $\text{diffuse}(m)$, process $p$ periodically sends $m$ to all processes, and checks if $\text{got}[m]$ contains all processes that are currently trusted by $p$; when that happens, $p$ delivers $m$ if it has not done so already. When process $p$ receives $m$ from a process $q$, it starts task $\text{diffuse}(m)$ if it has not done so already.

**Theorem 1.** Consider an asynchronous distributed system with process crashes and fair links, and with environment $E$. The algorithm in Figure 2 implements URB using $\Theta$ in $E$.

The proof is straightforward and can be found in Appendix A.

7 The Weakest Failure Detector for Uniform Reliable Broadcast

We now show that, in any environment, a failure detector $D$ that can be used to solve URB can be transformed to $\Theta$. We first give a rough outline of how the transformation works, and then give the detailed transformation algorithm and its proof.
7.1 Outline of the Transformation Algorithm

Let $E$ be an environment, $D$ be a failure detector that can be used to solve URB in $E$, and $A_{urb}$ be the URB algorithm that uses $D$. Intuitively, the algorithm that transforms $D$ into $Θ$ in $E$ works as follows.

Processes periodically query their failure detector $D$ and exchange information about the values of $D$ that they see. Using this information, processes construct a directed acyclic graph (DAG) that represents a “sampling” of the failure detector output and some temporal relationships between the values sampled.

To illustrate this, suppose that $q_0$ queries its failure detector $D$ for the $k_0$-th time and sees value $d_0$; $q_0$ then reliably broadcasts the message $[q_0, d_0, k_0]$ (it can use $A_{urb}$ to do so). When a process $q_1$ receives $[q_0, d_0, k_0]$, it can add vertex $[q_0, d_0, k_0]$ to its (current) version of the DAG. When $q_1$ later queries $D$ and sees the value $d_1$ (say this is its $k_1$-th query), it adds vertex $[q_1, d_1, k_1]$ and edge $[q_0, d_0, k_0] → [q_1, d_1, k_1]$ to its DAG: This edge indicates that $q_0$ saw $d_0$ (in its $k_0$-th query) before $q_1$ saw $d_1$ (in its $k_1$-th query). By periodically sending its current version of the DAG to all processes, and incorporating all the DAGs that it receives into its own DAG, a process can construct an ever increasing DAG that includes the failure detector values seen by processes and some of their temporal relationships.

Consider a run of the transformation algorithm above in which the failure pattern is $F ∈ E$, and the failure detector history is $H ∈ D(F)$. In this run, a process $p$ can use its DAG to simulate runs of $A_{urb}$ with failure pattern $F$ and failure detector history $H$. These are runs that could have occurred if processes were running $A_{urb}$ instead of the transformation algorithm.

To illustrate this, let $p$ be a process, and consider a path in its DAG, say $[q_0, d_0, k_0], [q_1, d_1, k_1], \ldots, [q_t, d_t, k_t]$. In the transformation algorithm, process $p$ uses this path to simulate a run $R_{urb}$ of $A_{urb}$. In $R_{urb}$, $q_0$ takes the 0-th step, $q_1$ takes the 1-st step, $q_2$ takes the 2-nd step, and so on. In the 0-th step, $q_0$ broadcasts $m_0$. Moreover, for every $j$, in the $j$-th step process $q_j$ sees failure detector value $d_j$ and receives the oldest message sent to it that it has not yet received (if there are no such messages, it receives nothing).

It turns out that, if failure pattern $F$ has some correct process, then process $p$ can extract from $R_{urb}$ a list of processes that contains at least one such a correct process. To see how, consider the step of $R_{urb}$ when a process first delivers $m_0$, and suppose this is the $k$-th step. Then, among processes $\{q_0, \ldots, q_k\}$ (those that took steps before the delivery of $m_0$), there is at least one that never crashes in $F$. If that were not the case, we could construct another run $R_{urb}^{bad}$ of $A_{urb}$ with failure pattern $F$ and failure detector history $H$, where (1) up to the $k$-th step, processes behave as in $R_{urb}$, (2) after the $k$-th step, processes $\{q_0, \ldots, q_k\}$ all crash, and all messages sent by these processes to other processes are lost and (3) from the $(k + 1)$-st step onwards, the correct processes (in $F$) take steps in a round-robin fashion. Note that in $R_{urb}^{bad}$, (1) process $q_k$ delivers $m_0$ at the $k$-th step, (2) correct processes (in $F$) only take steps after the $k$-th step, (3) these processes never receive a message sent by $k$-th step, and so (4) correct processes (in $F$) never deliver $m_0$ — a contradiction.

Thus, the list $\{q_0, \ldots, q_k\}$ contains at least one correct process (in $F$), and so $p$ can achieve the $Θ$-accuracy property by outputting this list.

The list $\{q_0, \ldots, q_k\}$ that $p$ generates, however, may contain processes that crash (in $F$). Thus, to achieve $Θ$-completeness, $p$ must continuously repeat the simulation above to generate new $\{q_0, \ldots, q_k\}$ lists, such that eventually the lists contain only correct processes (in $F$). In order to guarantee that, $p$ must ensure that the path $[q_0, d_0, k_0], [q_1, d_1, k_1], \ldots, [q_t, d_t, k_t]$ that it uses to extract $\{q_0, \ldots, q_k\}$ eventually includes only vertices of processes that do not crash. That will be true if all the processes that crash in $F$, do so before $q_0$ obtains $d_0$ at its $k_0$-th step. Therefore, process $p$ can achieve $Θ$-completeness (as well as $Θ$-accuracy) by simply periodically reselecting a new path $[q_0, d_0, k_0], [q_1, d_1, k_1], \ldots, [q_t, d_t, k_t]$ so that $[q_0, d_0, k_0]$ is a “recent” vertex in its DAG.
7.2 The Algorithm $T_{\mathcal{D} \rightarrow \Theta}$ and Its Proof

Having given an outline of the transformation algorithm, we now explain it in more detail. In what follows, let $S$ be a sequence of pairs consisting of a process name and a failure detector value, that is, $S := ([q_0, d_0], [q_1, d_1], \ldots, [q_k, d_k])$. Let $m_0$ be an arbitrary fixed message. Given $S$, we can simulate an execution of $A_{\text{urb}}$ in which (1) process $q_0$ initially invokes broadcast$(m_0)$ and (2) for $j = 0, \ldots, k$, the $j$-th step of $A_{\text{urb}}$ is taken by process $q_j$; in that step, $q_j$ obtains $d_j$ from its local failure detector module, and receives the oldest message addressed to it that it has not yet received (if there are no such messages, it receives nothing). We define Delivered$(S)$ to be true if process $q_k$ delivers $m_0$ in the $k$-th step of this simulation.

The detailed algorithm $T_{\mathcal{D} \rightarrow \Theta}$ that transforms $\mathcal{D}$ to $\Theta$ is given in Figure 3. As we explained in the outline, each process $p$ maintains a directed acyclic graph $DAG_p$, whose nodes are triples $[q, d, \text{curr}]$. The transformation algorithm has three tasks; in the first task, a process $p$ periodically queries its local failure detector, creates a new node $[p, d, \text{curr}]$ in $DAG_p$ and adds an edge from all other nodes in $DAG_p$ to this new node. Then, $p$ uses $A_{\text{urb}}$ to broadcast its new $DAG_p$ to all processes. In the second task, upon the delivery of $DAG_q$ from a process $q$, process $p$ merges its own $DAG_p$ with $DAG_q$. In the third task, process $p$ loops forever. In the loop, $p$ first waits until its Task 1 adds a new node to $DAG_p$, and then waits until there is a path starting at this new node that truthifies Delivered. Once $p$ finds such a path, it sets the output of $\mathcal{D'}$ to the set of all processes that appear in the path. Then, process $p$ restarts the loop.

Let $\mathcal{E}$ be an environment, $\mathcal{D}$ be a failure detector that can be used to solve URB in $\mathcal{E}$, and $A_{\text{urb}}$ be the URB algorithm that uses $\mathcal{D}$. Consider a run $R_{\text{trans}}$ of the transformation algorithm of Figure 3, and we let $C$ be the set of correct processes in this run. In order to differentiate between the suspicions of $\mathcal{D}$ and of $\mathcal{D'}$, we henceforth prefix the word “trust” by the failure detector to which it refers. For example, we say “$p \mathcal{D'}$-trusts”.

Lemma 2. If there is a correct process then, at every time, every process $p \mathcal{D'}$-trusts at least one correct process.

Proof. Initially, $p$ sets $\mathcal{D'}_p$ to $\Pi$, so clearly if there is a correct process $q$, then $p \mathcal{D'}$-trusts $q$.

Now suppose $p$ sets $\mathcal{D'}_p$ to a set $S$ in line 27 at time $t$. In order to obtain a contradiction, assume that $S$ contains no correct process, i.e., $S \cap C = \emptyset$. At time $t$, $DAG_p$ contains a path $P = ([q_0, d_0, \text{seq}_0], \ldots, [q_k, d_k, \text{seq}_k])$ such that (1) $q_0 = p$, (2) Delivered$([q_0, d_0], \ldots, [q_k, d_k]) = \text{true}$, and (3) $\{q_0, \ldots, q_k\} = S$. Since $DAG_p$ contains $P$, we claim that there exists a of run $R_{\text{adv}}$ of $A_{\text{urb}}$ in environment $\mathcal{E}$ such that

- the set of correct processes is $C$, and
- process $p = q_0$ initially invokes broadcast$(m_0)$, and
- for $j = 0, \ldots, k$, the $j$-th step is taken by process $q_j$; in that step, $q_j$ obtains $d_j$ from its local failure detector module, and receives the oldest message addressed to it that it has not yet received (if there are no such messages, it receives nothing), and
- after the $k$-th step, (1) the set of messages sent by the $k$-th step and not yet received are lost and (2) processes in $C$ take steps in round-robin fashion, obtain some value from their failure detector, and receive the oldest message not yet received that was sent after the $k$-th step

To see why the claims holds, note that $DAG_p$ was constructed in a run $R_{\text{trans}}$ with failure detector $\mathcal{D}$ in environment $\mathcal{E}$; since $DAG_p$ contains $P$, then in this run the following happened in chronological order: (1)
For every process $p$:

Initialization:

\[ \text{DAG} \leftarrow \emptyset \]

\[ \text{curr} \leftarrow -1 \]

\[ D'_p \leftarrow \Pi \quad \{ \text{trust all processes} \} \]

cobegin

|| Task 1:||

while true do

\[ d \leftarrow D_p \]

\[ \text{curr} \leftarrow \text{curr} + 1 \]

add to DAG the node $[p, d, \text{curr}]$ and edges from all other nodes to $[p, d, \text{curr}]$

broadcast(DAG) \quad \{ \text{use URB algorithm to broadcast} \}

|| Task 2:||

upon deliver(DAG$_q$) from $q$ do

\[ \text{DAG} \leftarrow \text{DAG} \cup \text{DAG}_q \]

|| Task 3:||

while true do

\[ \text{next} \leftarrow \text{curr} + 1 \]

wait until DAG contains a node of the form $[p, *, \text{next}]$

wait until DAG contains a path $P = ([q_0, d_0, seq_0], \ldots, [q_k, d_k, seq_k])$ such that

\begin{enumerate}
  \item $q_0 = p$ and $seq_0 = \text{next}$ and
  \item $\text{Delivered}([q_0, d_0], \ldots, [q_k, d_k])$ is true
\end{enumerate}

\[ D'_p \leftarrow \{q_0, \ldots, q_k\} \quad \{ \text{all processes in this path} \} \]

coen

Figure 3: Transformation of $\mathcal{D}$ to $\mathcal{D}' = \emptyset$

$q_0$ took a step and obtained $d_0$ from its local failure detector; (2) $q_1$ took a step and obtained $d_1$ from its local failure detector; (3) $q_2$ took a step and obtained $d_2$ from its local failure detector; and so on. Note that $C$ is the set of correct process in $R_{\text{trans}}$. Thus, there is a run $R_{\text{urb}}^{bad}$ of $\mathcal{A}_{\text{urb}}$ with $\mathcal{D}$ in environment $\mathcal{E}$ in which $q_0$ broadcasts $m_0$, such that: (1) $C$ is the set of correct processes; (2) for $j = 0, \ldots, k$, in the $j$-th step, $q_j$ obtains $d_j$ from its local failure detector and receives the oldest message addressed to it that it has not yet received; (3) messages sent by the $k$-th step that were not received by the $k$-th step are lost; (4) after the $k$-th step, processes in $C$ take steps in a round-robin fashion, obtain some value from their failure detector, and receive the oldest message not yet received that was sent after the $k$-th step. This is a valid run in our model because the correct processes $C$ take an infinite number of steps, and only a finite number of message are lost (the lost message are those that are sent, but not received by the $k$-th step). This shows the claim.

Now consider run $R_{\text{urb}}^{bad}$. Up to the $k$-th step, no process in $C$ takes a step (since only processes in $S$ take a step, and $S \cap C = \emptyset$ by assumption). At the $k$-th step, process $q_k$ delivers $m_0$, since $\text{Delivered}([q_0, d_0], \ldots, [q_k, d_k]) = \text{true}$. After the $k$-th step, only processes in $C \neq \emptyset$ take a step, and they never receive a message sent by the $k$-th step. It is easy to see that processes in $C$ do not deliver $m_0$. Since $q_k$ delivers $m_0$ and processes in $C$ are correct, $R_{\text{urb}}^{bad}$ violates the Uniform Agreement property of URB — a contradiction.

**Lemma 3.** If $p$ is a correct process and at some time $\text{DAG}_p$ contains a path $P$, then eventually for every
To show the claim, let $true$. This claim immediately implies that $p$ does not block forever in line 24.

Lemma 4. If $p$ is a correct process then $p$ does not block forever in lines 23 or 24.

Proof. Let $p$ be a correct process. Then $p$ does not block forever in line 23, since eventually its Task 2 adds to $DAG_p$ a node of the form $[p,*,next]$. To see that $p$ does not block forever in line 24, let seq$_0$ be the value of $next_p$ when $p$ starts line 24, and let d$_0$ be such that $[p,d_0,seq_0]$ belongs to $DAG_p$ (such a $d_0$ exists because $p$ has just executed past line 23). We claim that eventually $DAG_p$ contains a path $P = ([p,d_0,seq_0],[q_1,d_1,seq_1],\ldots,[q_k,d_k,seq_k])$ such that $Delivered([p,d_0],[q_1,d_1],\ldots,[q_k,d_k])$ is true. This claim immediately implies that $p$ does not block forever in line 24.

To show the claim, let $r = |C|$, $q_0 = p$ and $q_1,\ldots,q_{|C|−1}$ be the processes in $C \setminus \{p\}$ in some arbitrary order. In order to obtain a contradiction, suppose that at every time, any path $P$ in $DAG_p$, whose first node is $[p,d_0,seq_0]$ satisfies $Delivered(P) = false$. We now define inductively an increasing sequence $P_0,P_1,\ldots$ of paths that are all eventually in $DAG_p$. Let $P_0$ be the singleton path $([p,d_0,seq_0])$ and note that $DAG_p$ contains $P_0$. For $j ≥ 1$, given that $P_{j−1}$ is in $DAG_p$, by Lemma 3 eventually $DAG_p$ contains a path $P_{j−1} = [q_{jmod[C]},d_j,seq_j]$ for some $d_j$ and seq$_j$. We set $P_j$ to be such a path.

We can now construct a run $R^0_{urw}$ of $A_{urw}$ using $D$ in environment $E$, where:

- the set of correct processes is $C$, and
- process $q_0$ initially invokes $broadcast(m_0)$, and
- for $j ≥ 0$, the $(j+1)$-th step is taken by process $q_{jmod[C]}$; in that step, $q_{jmod[C]}$ obtains $d_j$ from its local failure detector module, and receives the oldest message addressed to it that it has not yet received (if there are no such messages, it receives nothing).

Note that $R^0_{urw}$ is a valid run of $A_{urw}$ using $D$ in environment $E$. Since $q_0$ is correct in $R^0_{urw}$, it must eventually deliver $m_0$, say at some step $k$. Therefore, we have that $Delivered(P_k) = true$. This contradicts the assumption that $Delivered(P) = false$ for every path $P$ in $DAG_p$ whose first node is $[p,d_0,seq_0]$.

Lemma 5. There is a time after which correct processes do not $D'$-trust any process that crashes.

Proof. Let $p$ be a correct process and $q$ be a process that crashes. We now show that there is a time after which $p$ does not $D'$-trusts $q$. Let $t_0$ be the time when $q$ crashes. Let $curr^t_p$ be the value of $curr_p$ at time $t_0$.

We claim that for any seq $≥ curr^t_p + 1$, $DAG_p$ can never contain a path whose first node is $[p,*,seq]$ and that has a subsequent node of the form $[q,*,*]$. To see why, note that for any path $P = ([p,d_0,seq],[q_1,d_1,seq_1],\ldots,[q_k,d_k,seq_k])$ in $DAG_p$, it must be the case that after time $t_0$ $p$ obtains $d_0$ from $D_p$ and adds node $[p,d_0,seq]$ to $DAG_p$, which happens no later than $p$ broadcasts a $DAG_p$ containing node $[p,d_0,seq]$, which happens no later than $q$ delivers a $DAG_p$ containing node $[p,d_0,seq]$. 

11
Note that there is an edge \([p, d_0, seq] \rightarrow [q_1, d_1, seq_1]\) in \(DAG_p\), and so there must be such an edge in \(DAG_{q_1}\) (this is because for any process \(q\), an edge of the form \([*, *, *] \rightarrow [q, *, *]\) must appear \(DAG_q\) before it appears in the \(DAG\) of any other process). Therefore, \(q_1\) delivers a \(DAG_p\) containing node \([p, d_0, seq]\) no later than \(q_1\) adds node \([q_1, d_1, seq_1]\) to \(DAG_{q_1}\). Moreover, through a similar reasoning, we have that for any \(j : 1 \leq j < k\), \(q_j\) adds node \([q_j, d_j, seq_j]\) to \(DAG_{q_j}\) no later than \(p\) adds this same node to \(DAG_p\), which happens no later than \(q_{j+1}\) added node \([q_{j+1}, d_{j+1}, seq_{j+1}]\) to \(DAG_{q_{j+1}}\).

With all that, we conclude that for any \(j : 0 \leq j \leq k\), \(q_j\) added node \([q_j, d_j, seq_j]\) to \(DAG_{q_j}\) after time \(t_0\). Since \(q\) crashes at time \(t_0\), we conclude that \(q \neq q_j\). Thus \(DAG_p\) can never contain a path whose first node is \([p, *, seq]\) and that has a subsequent node of the form \([q, *, *]\).

Now let \(t_1 \geq t_0\) be the time when \(p\) reaches line 27 with \(next_p\) set to \(curr_p^0 + 1\). Note that eventually a path is selected in line 24 after time \(t_1\). By the claim, no path selected after time \(t_1\) contains a node of the form \([q, *, *]\). Thus, after time \(t_1\), \(p\) does not \(D^\prime\)-trust \(q\).

We can now prove our main result.

**Theorem 6.** Consider an asynchronous distributed system with process crashes and fair links, and with environment \(E\). Suppose failure detector \(D\) can be used to solve URB in \(E\). Then \(D\) can be transformed in \(E\) to the \(\Theta\) failure detector.

**Proof.** By Lemma 2, \(D^\prime\) satisfies \(\Theta\)-accuracy, and by Lemma 5, \(D^\prime\) satisfies \(\Theta\)-completeness. Therefore the algorithm in Figure 3 transforms \(D\) to \(\Theta\). □

## 8 On the Nature and Power of Failure Detectors

As we mentioned in the introduction, to understand the meaning of a statement such as “\(D\) is the weakest failure detector...”, or “\(D\) is necessary...”, we need to know the universe of failure detectors under consideration. For such minimality results to be significant, the universe of failure detectors should be reasonable. In particular, it should not include failure detectors that provide information that have nothing to do with failures, e.g., hints on which messages have been broadcast, information about the internal state of processes, etc. To see why, suppose that a “failure detector” is allowed to indicate whether a message \(m\) was broadcast; then processes could use it to solve the URB problem without ever sending any messages! Similarly, with the Consensus problem, if a “failure detector” could peek at the initial value of a process and provide this value to all processes, processes could use it to solve Consensus without messages and without \(\Diamond W\) [CHT96]. Thus, a failure detector should be defined as an oracle that provides information about failures only.

In [HR99], it is not clear what information failure detectors are allowed to provide: On one hand, the formal model defines failure detectors as generic oracles;\(^6\) on the other hand, their behavior is implicitly restricted by a closure axiom (on the set of runs of the system) that is introduced later in the paper.\(^7\) The difficulty is that this axiom is technical and quite complex; furthermore, it does not mention failure detectors and it captures other assumptions that are not related to failure detection (e.g., the fact that processes are using a full-information protocol). Thus, the nature and power of the failure detectors that actually satisfy this axiom, and the universe of failure detectors under consideration, are not entirely clear.

\(^6\)Even though the definition of a failure detector states that it must output a set \(S\) of processes, and that \(S\) should be “interpreted” as processes suspected of being faulty, there is nothing in the definition to enforce this interpretation: the model does not tie the output \(S\) to the crashes that occur in a run. Thus, the formal definition allows a failure detector to use its output \(S\) to encode information that has nothing to do with failures.

\(^7\)This axiom, A4, is given in Appendix B.
For every process $p$:

Initialization:

\[ D'_p \leftarrow \Pi \]
\[ \text{got} \leftarrow \emptyset \]

\[ \text{cobegin} \]

\[ || \text{Task 1:} \]
\[ \text{while true do send } (I-am-alive) \text{ to all processes} \]

\[ || \text{Task 2:} \]
\[ \text{upon receive } (I-am-alive) \text{ from } q \text{ do} \]
\[ \text{got} \leftarrow \text{got} \cup \{q\} \]

\[ || \text{Task 3:} \]
\[ \text{while true do} \]
\[ \text{if there exists } S, k \text{ such that} \]
\[ (1) \text{ got event suspect}(S, k) \text{ (from } G^f), \]
\[ (2) k > |S| - n + \min(f, n - 1), \] and
\[ (3) \text{ got contains } \Pi \setminus S \]
\[ \text{ then } D'_p \leftarrow \text{got}; \text{got} \leftarrow \emptyset \]

\[ \text{coend} \]

Figure 4: Transformation of $G^f$ to $D' = \Theta$ in $E^f$.

9 Relation between $G^f$ and $\Theta$

[HR99] shows that failure detector $G^f$ is necessary and sufficient to solve URB in environment $E^f$ (recall that $E^f$ is the set of all failure patterns in which at most $f$ processes crash: in a system with environment $E^f$ any subset of up to $f$ processes may crash, and these crashes can occur at any time).

We now show that in environment $E^f$, $\Theta$ is at least as weak as $G^f$, that is, it is possible to transform $G^f$ to $\Theta$ in $E^f$. The transformation algorithm is given in Figure 4. Initially, each process $p$ sets its failure detector output to $\Pi$ (trust all processes). There are three concurrent tasks. In the first task, $p$ repeatedly sends “I-am-alive” to all processes in the system. In the second task, when $p$ receives one such message from process $q$, it adds $q$ to the set $\text{got}$. In the third task, process $p$ loops forever. In each iteration, $p$ checks whether at some time $G^f$ has output a pair $(S, k)$ such that $k > |S| - n + \min(f, n - 1)$ and $\text{got}$ contains the complement of $S$. In that case, $p$ sets its failure detector output to $\text{got}$, and then resets $\text{got}$ to the empty set.

We now show that the algorithm in Figure 4 transforms $G^f$ into $\Theta$ in systems with process crashes and link failures, and environment $E^f$. To do so, consider a run of this algorithm in such a system.

Lemma 7. If there is a correct process then, at every time, every process $D'$-trusts at least one correct process.

Proof. Assume that there is a correct process, and let $p$ be any process. Initially, $p$ $D'$-trusts all processes, so clearly there exists some correct process that is $D'$-trusted by $p$. Now assume that $p$ sets its failure detector output in line 21, and consider the value of some pair $(S, k)$ that truthifies lines 18 and 19. Then, in environment $E^f$, the definition of $G^f$ implies that $\Pi \setminus S$ contains at least one correct process. Thus, by
Lemma 8. If \( p \) is a correct process then \( p \) executes line 21 infinitely often.

Proof. In order to obtain a contradiction, assume that \( p \) executes line 21 only finitely often, and let \( t_{\text{final}} \) be the time when \( p \) executes this line for the last time (if \( p \) never executes this line, let \( t_{\text{final}} = 0 \)). Let \( C \) be the set of correct processes. After time \( t_{\text{final}} \), \( p \) never resets \( \text{got} \) to the empty set. Since \( p \) receives messages from correct processes infinitely often, then eventually \( \text{got} \) contains \( C \) after time \( t_{\text{final}} \). Moreover, by definition of \( G_f^\ell \), eventually \( p \) gets an \( f \)-useful event \( \text{suspect}(S,k) \). That means that \( S \) contains \( \Pi \setminus C \) and condition (2) in line 19 holds. Thus \( C \) contains \( \Pi \setminus S \). Since \( \text{got} \) contains \( C \) after time \( t_{\text{final}} \), then \( \text{got} \) contains \( \Pi \setminus S \) after time \( t_{\text{final}} \), and so condition (3) in line 20 holds. Thus, after time \( t_{\text{final}} \), \( p \) executes line 21 — a contradiction.

Lemma 9. There is a time after which correct processes do not \( \mathcal{D}' \)-trust any process that crashes.

Proof. Let \( p \) be a correct process. Let \( t_0 \) be the time when the last process crashes, and let \( t_1 > t_0 \) be the time after which no messages sent by \( t_0 \) are received (such a time \( t_1 \) exists because of the Uniform Integrity property of links and the fact that only a finite number of messages are sent by \( t_0 \)). Then, after \( t_1 \) all messages received were sent by correct processes. By Lemma 8, there exists a time \( t_2 > t_1 \) when \( p \) executes line 21. Note that after time \( t_2 \), variable \( \text{got} \) contains only correct processes. Let \( t_3 > t_2 \) be next time when \( p \) executes line 21 (such a time \( t_3 \) exists by Lemma 8). Then, after \( t_3 \) \( p \) does not \( \mathcal{D}' \)-trust any process that crashes.

Theorem 10. The algorithm in Figure 4 transforms \( G_f^\ell \) into \( \Theta \) in systems with process crashes and link failures, and environment \( \mathcal{E}_f^\ell \).

Proof. Lemmata 7 and 9 show that \( \mathcal{D}' \) satisfies \( \Theta \)-accuracy and \( \Theta \)-completeness, respectively.

10 Related Work

The difference between the concepts of Agreement and Uniform Agreement was first pointed out in [Had86] in a comparison of Consensus versus Atomic Commitment. The term “Uniform” was introduced in [GT89, NT90], where it was studied in the context of Reliable Broadcast. In these papers, it is shown that with send and receive omission failures, URB can be solved if and only if a majority of processes are correct.

[BCBT96] consider systems with process crashes and fair (lossy) links, and addresses the following question: given any problem \( P \) that can be solved in a system where the only possible failures are process crashes, is \( P \) still solvable if links can also fail by losing messages? [BCBT96] shows that if \( P \) can be solved in systems with only process crashes, then \( P \) can also be solved in systems with process crashes and fair links, provided that (a) \( P \) is correct-restricted\(^8\), or (b) a majority of processes are correct (i.e., \( n > 2f \)). As a corollary of this result (and the fact that URB is solvable in systems with only process crashes), we get that URB is solvable in systems with \( f < n/2 \) process crashes and fair links.

[HR99] is the first paper to consider solving URB in systems with fair links and \( f \geq n/2 \). As mentioned above, this cannot be done without failure detectors, and [HR99] determined that failure detector \( G_f^\ell \) is

\(^8\)Intuitively, a problem \( P \) is correct-restricted if its specification does not refer to the behavior of faulty processes [BN92, Gop92]. Note that URB is not correct-restricted.
necessary and sufficient to solve this problem in $E_f$. A discussion of the differences between [HR99] and this paper was given in Section 1.

11 Concluding Remarks

In some environments, URB can be solved without failure detectors at all, and this seems to contradict the fact that $\Theta$ is the weakest failure detector for URB in any environment. There is no contradiction, however, because in such environments $\Theta$ can be implemented.

For example, as we saw in the previous section, URB can be solved without failure detectors in an environment $E_{maj}$ where a majority of processes are correct. This does not contradict Theorem 6 because $\Theta$ can be implemented in $E_{maj}$, as we now explain.

To implement $\Theta$ in $E_{maj}$, processes periodically send an “I-am-alive” message to all processes, and each process $p$ keeps a list of processes $Order_p$. This list records the order in which the last “I-am-alive” message from each process is received. More precisely, $Order_p$ is initially an arbitrary permutation of the processes, and when $p$ receives an “I-am-Alive” message from $q$, $p$ moves $q$ to the front of $Order$. To obtain $\Theta$, a process $p$ repeatedly outputs the first $\lceil (n+1)/2 \rceil$ processes in $Order_p$ as the set of trusted processes. It is easy to see why this implementation works: any process that crashes stops sending “I-am-alive” messages and soon moves towards the end of $Order_p$. Since at most $\lfloor (n-1)/2 \rfloor$ processes crash, all processes that crash are eventually among the last $\lfloor (n-1)/2 \rfloor$ processes in $Order_p$ — so they do not appear among the first $\lceil (n+1)/2 \rceil$ processes. Thus our implementation satisfies $\Theta$-completeness. To see that it also satisfies $\Theta$-accuracy, note that among the first $\lceil (n+1)/2 \rceil$ processes in $Order_p$, there is always at least one correct process (since no majority of processes can crash in $E_{maj}$).

In general, from the transformation algorithm in Figure 3, the following obviously holds:

**Remark 1.** Consider an asynchronous distributed system with process crashes and fair links, and with environment $E$. If URB can be solved in $E$ without any failure detectors then $\Theta$ can be implemented in $E$.

References


Appendix

A Proof of Theorem 1

We now prove that the algorithm in Figure 2 implements URB in any system with process crashes and link failures, and with any environment. Consider a run of the algorithm using \( D = \Theta \) in any environment.

**Lemma 11.** If a correct process \( p \) starts task \( \text{diffuse}(m) \) then eventually all correct processes start task \( \text{diffuse}(m) \).

*Proof.* Let \( q \) be a correct process. In task \( \text{diffuse}(m) \), process \( p \) repeatedly sends \( m \) to all processes, including \( q \). By the Fairness property of links, eventually \( q \) receives \( m \) from \( p \), and starts task \( \text{diffuse}(m) \) if it has not done so already. \( \square \)

**Lemma 12.** If a correct process \( p \) starts \( \text{diffuse}(m) \) then \( p \) eventually delivers \( m \).

*Proof.* Let \( q \) be a correct process; we first argue that eventually \( q \in \text{got}_p[m] \) holds forever. Indeed, by Lemma 11, \( q \) eventually starts task \( \text{diffuse}(m) \). In that task, \( q \) sends \( m \) to \( p \) an infinite number of times. By the Fairness property of links, \( p \) eventually receives \( m \) from \( q \) and adds \( q \) to \( \text{got}_p[m] \). Once that happens, \( q \) remains in \( \text{got}_p[m] \) forever.

We conclude that eventually every correct process is in \( \text{got}_p[m] \) forever. By the \( \Theta \)-completeness property, there is a time after which the output \( d \) of \( D_p \) contains only correct processes. Therefore, eventually \( d \subseteq \text{got}_p[m] \), and \( p \) delivers \( m \). \( \square \)

**Corollary 13.** If a correct process \( p \) broadcasts a message \( m \) then \( p \) eventually delivers \( m \).

**Corollary 14.** If some process \( p \) delivers a message \( m \) then all correct processes eventually deliver \( m \).

*Proof.* If there are no correct processes, the corollary is vacuously true, so assume there is some correct process. If \( p \) delivers \( m \) then for some \( d \), (1) \( p \) obtained \( d \) from \( D_p \) and (2) \( d \subseteq \text{got}_p[m] \). By (1) and the \( \Theta \)-accuracy property, \( d \) contains at least one correct process \( q \). By (2), \( q \in \text{got}_p[m] \). It is easy to show that this implies that \( q \) started task \( \text{diffuse}(m) \). Since \( q \) is correct, by Lemma 11 all correct processes start task \( \text{diffuse}(m) \). By Lemma 12, all correct processes deliver \( m \). \( \square \)

**Lemma 15.** For every message \( m \), every process delivers \( m \) at most once, and only if \( m \) was previously broadcast by \( \text{sender}(m) \).

*Proof.* From the code of the algorithm: (a) a process only delivers a message \( m \) if it has not done so previously, and so every process delivers \( m \) at most once; and (b) a process can only deliver \( m \) if it starts task \( \text{diffuse}(m) \). By the Uniform Integrity property of links, it is easy to show that a process only starts \( \text{diffuse}(m) \) if \( m \) was previously broadcast by \( \text{sender}(m) \). \( \square \)

**Theorem 16.** Consider an asynchronous distributed system with process crashes and fair links, and with environment \( \mathcal{E} \). The algorithm in Figure 2 implements URB using \( \Theta \) in \( \mathcal{E} \).

*Proof.* Validity, Uniform Agreement and Uniform Integrity follow from Corollary 13, Corollary 14 and Lemma 15, respectively. \( \square \)
B  The A4 Axiom of [HR99]

The axiom that implicitly restricts the behavior of failure detectors in [HR99] is:

(A4) If $\varphi$ is a stable formula local to some process $p$ in $\mathcal{R}$ that is insensitive to failure by $p$ and there is some $S \subseteq \text{Proc}$ such that $(\mathcal{R}, r, t) \models \bigwedge_{q \in S} \neg K_q \varphi$, then there exists a point $(r', t)$ such that (a) $r'_q(t) = r_q(t)$ for $q \in S$, (b) for $q \not\in S$, there is a prefix $h$ of $r_q(t)$ (not necessarily strict) such that $r'_q(t)$ is either $h$ or $h \cdot \text{crash}_q$, and $r'_q(t) = h \cdot \text{crash}_q$ only if $\text{crash}_q \in r_q(t)$, (c) $(\mathcal{R}, r', t) \models \neg \varphi$.

In this definition, $\mathcal{R}$ is a system (a set of runs), $\text{Proc}$ is the set of processes in the system (in our paper, this is denoted $\Pi$). The notation $(\mathcal{R}, r, t) \models \varphi$ means that formula $\varphi$ is true at point $(r, t)$ in system $\mathcal{R}$, where $r$ is a run in $\mathcal{R}$ and $t$ is a time. Roughly speaking, $r_q(t)$ is the prefix of run $r$ at $q$ up to time $t$ (for details, see [HR99]). $\text{crash}_q$ is an event that is in $q$’s history if $q$ crashes. The allowed formulas $\varphi$ are primitive propositions, closed off under Boolean combinations, $\Box$, and the epistemic operators $K_p$ for each process $p$. Among the primitive propositions are $\text{send}_p(q, \text{msg})$, $\text{recv}_q(p, \text{msg})$, $\text{crash}(p)$, $\text{do}_p(\alpha)$, and $\text{init}_p(\alpha)$. A formula $\varphi$ local to $q$ is said to be insensitive to failure by $q$ if for all runs $r, r' \in \mathcal{R}$, if $r'_q(t') = r_q(t) \cdot \text{crash}_q$, then $(\mathcal{R}, r, t) \models \varphi$ iff $(\mathcal{R}, r', t') \models \varphi$.

\[9\text{Recall than in [HR99], broadcast and deliver correspond to init and do, respectively.}\]