With Finite Memory Consensus Is Easier Than Reliable Broadcast

Carole Delporte-Gallet[1], Stéphane Devismes[2], Hugues Fauconnier[1], Franck Petit[3,⋆], and Sam Toueg[4]

[1] LIAFA, Université D. Diderot, Paris, France
{cd,hf}@liafa.jussieu.fr
[2] VERIMAG, Université Joseph Fourier, Grenoble, France
stephane.devismes@imag.fr
[3] INRIA/LIP Laboratory,Univ. of Lyon/ENS Lyon, Lyon, France
franck.petit@ens-lyon.fr
[4] Department of Computer Science, University of Toronto, Toronto, Canada
sam@cs.toronto.edu

Abstract. We consider asynchronous distributed systems with message losses and process crashes. We study the impact of finite process memory on the solution to *consensus*, *repeated consensus* and *reliable broadcast*. With finite process memory, we show that in some sense consensus is easier to solve than reliable broadcast, and that reliable broadcast is as difficult to solve as repeated consensus: More precisely, with finite memory, consensus can be solved with failure detector $\mathcal{S}$, and $\mathcal{P}^-$ (a variant of the perfect failure detector which is stronger than $\mathcal{S}$) is necessary and sufficient to solve reliable broadcast and repeated consensus.

1 Introduction

Designing fault-tolerant protocols for asynchronous systems is highly desirable but also highly complex. Some classical agreement problems such as *consensus* and *reliable broadcast* are well-known tools for solving more sophisticated tasks in faulty environments (e.g., [1,2]). Roughly speaking, with consensus processes must reach a common decision on their inputs, and with reliable broadcast processes must deliver the same set of messages.

It is well known that consensus cannot be solved in asynchronous systems with failures [3], and several mechanisms were introduced to circumvent this impossibility result: *randomization* [4], *partial synchrony* [5,6] and *(unreliable) failure detectors* [7].

Informally, a failure detector is a distributed oracle that gives (possibly incorrect) hints about the process crashes. Each process can access a local failure detector module that monitors the processes of the system and maintains a list of processes that are suspected of having crashed.

⋆ This work was initiated while Franck Petit was with MIS Lab., Université of Picardie, France. Research partially supported by Région Picardie, Proj. APREDY.

T.P. Baker, A. Bui, and S. Tixeuil (Eds.): OPODIS 2008, LNCS 5401, pp. 41–57, 2008.

Several classes of failure detectors have been introduced, *e.g.*, $\mathcal{P}$, $\mathcal{S}$, Ω, etc. Failure detectors classes can be compared by reduction algorithms, so for any given problem P, a natural question is "*What is the weakest failure detector (class) that can solve P ?*". This question has been extensively studied for several problems in systems *with infinite process memory* (*e.g.*, uniform and non-uniform versions of consensus [8,9,10], non-blocking atomic commit [11], uniform reliable broadcast [12,13], implementing an atomic register in a message-passing system [11], mutual exclusion [14], boosting obstruction-freedom [15], set consensus [16,17], etc.). This question, however, has not been as extensively studied in the context of systems *with finite process memory*.

In this paper, we consider systems where processes have finite memory, processes can crash and links can lose messages (more precisely, links are fair lossy and FIFO[1]). Such environments can be found in many systems, for example in sensor networks, sensors are typically equipped with small memories, they can crash when their batteries run out, and they can experience message losses if they use wireless communication.

In such systems, we consider (the uniform versions of) reliable broadcast, consensus and repeated consensus. Our contribution is threefold: First, we establish that the weakest failure detector for reliable broadcast is $\mathcal{P}^-$ — a failure detector that is almost as powerful than the perfect failure detector $\mathcal{P}$. Next, we show that consensus can be solved using failure detector $\mathcal{S}$. Finally, we prove that $\mathcal{P}^-$ is the weakest failure detector for repeated consensus. Since $\mathcal{S}$ is strictly weaker than $\mathcal{P}^-$, in some precise sense these results imply that, in the systems that we consider here, consensus is easier to solve than reliable broadcast, and reliable broadcast is as difficult to solve as repeated consensus.

The above results are somewhat surprising because, when processes have infinite memory, reliable broadcast is easier to solve than consensus[2], and repeated consensus is not more difficult to solve than consensus.

Roadmap. The rest of the paper is organized as follows: In the next section, we present the model considered in this paper. In Section 4, we show that in case of process memory limitation and possibility of crashes, $\mathcal{P}^-$ is necessary and sufficient to solve reliable broadcast. In Section 5, we show that consensus can be solved using a failure detector of type $\mathcal{S}$ in our systems. In Section 6, we show that $\mathcal{P}^-$ is necessary and sufficient to solve repeated consensus in this context.

For space considerations, all the proofs are omitted, see the technical report for details ([20], `http://hal.archives-ouvertes.fr/hal-00325470/fr/`).

[1] The FIFO assumption is necessary because, from the results in [18], if lossy links are not FIFO, reliable broadcast requires unbounded message headers.

[2] With infinite memory and fair lossy links, (uniform) reliable broadcast can be solved using Θ [19], and Θ is strictly weaker than (Σ, Ω) which is necessary to solve consensus.

2 Model

Distributed System. A system consists of a set $\Pi = \{p_1, ..., p_n\}$ of processes. We consider *asynchronous* distributed systems where each process can communicate with each other through *directed links*.[3] By asynchronous, we mean that there is no bound on message delay, clock drift, or process execution rate.

A process has a local memory, a local sequential and deterministic algorithm, and input/output capabilities. In this paper we consider systems of processes having either a *finite* or an *infinite* memory. In the sequel, we denote such systems by $\Phi^{\mathcal{F}}$ and $\Phi^{\mathcal{I}}$, respectively.

We consider links with unbounded capacities. We assume that the messages sent from p to q are *distinguishable*, *i.e.*, if necessary, the messages can be numbered with a non-negative integer. These numbers are used for notational purpose only, and are unknown to the processes. Every link satisfies the *integrity*, *i.e.*, if a message m from p is received by q, m is received by q at most once, and only if p previously sent m to q. Links are also *unreliable* and *fair*. Unreliable means that the messages can be lost. Fairness means that for each message m, if process p sends infinitely often m to process q and if q tries to receive infinitely often a message from p, then q receives infinitely often m from p. Each link are *FIFO*, *i.e.*, the messages are received in the same order as they were sent.

To simplify the presentation, we assume the existence of a discrete global clock. This is merely a fictional device: the processes do not have access to it. We take the range $\mathcal{T}$ of the clock's ticks to be the set of natural numbers.

Failures and Failure Patterns. Every process can fail by permanently *crashing*, in which case it definitively stops to execute its local algorithm. A *failure pattern* F is a function from $\mathcal{T}$ to 2^{Π}, where $F(t)$ denotes the set of processes that have crashed through time t. Once crashed, a process never recoves, *i.e.*, $\forall t : F(t) \subseteq F(t+1)$. We define $crashed(F) = \bigcup_{t \in \mathcal{T}} F(t)$ and $correct(F) = \Pi \backslash crashed(F)$. If $p \in crashed(F)$ we say that p *crashes in* F (or simply *crashed* when it is clear in the context) and if $p \in correct(F)$ we say that p is *correct in* F (or simply *correct* when it is clear in the context). An environment is a set of failure patterns. We do not restrict here the number of crash and we consider as environment $\mathcal{E}$ the set of all failure patterns.

Failure Detectors. A failure detector [7] is a local module that outputs a set of processes that are currently suspected of having crashed. A *failure detector history* H is a function from $\Pi \times \mathcal{T}$ to 2^{Π}. $H(p,t)$ is the value of the failure detector module of process p at time t. If $q \in H(p,t)$, we say that p *suspects* q *at time* t *in* H. We omit references to H when it is obvious from the context.

Formally, *failure detector* $\mathcal{D}$ is a function that maps each failure pattern F to a set of failure detector histories $\mathcal{D}(F)$.

A failure detector can be defined in terms of two *abstract properties*: *Completeness* and *Accuracy* [7] . Let us recall one type of *completeness* and two types of *accuracy*.

[3] We assume that each process knows the set of processes that are in the system; some papers related to failure detectors do not make this assumption e.g. [21,22,23].

Definition 1 (Strong Completeness). *Eventually every process that crashes is permanently suspected by* every correct process*. Formally, $\mathcal{D}$ satisfies strong completeness if:* $\forall F \in \mathcal{E}, \forall H \in \mathcal{D}(F), \exists t \in \mathcal{T}, \forall p \in crashed(F), \forall q \in correct(F), \forall t' \geq t : p \in H(q, t')$

Definition 2 (Strong Accuracy). *No process is suspected before it crashes. Formally, $\mathcal{D}$ satisfies strong accuracy if:* $\forall F \in \mathcal{E}, \forall H \in \mathcal{D}(F), \forall t \in \mathcal{T}, \forall p, q \in \Pi \setminus F(t) : p \notin H(q, t)$

Definition 3 (Weak Accuracy). *A correct process is never suspected. Formally, $\mathcal{D}$ satisfies weak accuracy if:* $\forall F \in \mathcal{E}, \forall H \in \mathcal{D}(F), \forall t \in \mathcal{T}, \exists p \in correct(F), \forall q \in \Pi : p \notin H(q, t)$

We introduce a last type of accuracy:

Definition 4 (Almost Strong Accuracy). *No correct process is suspected. Formally, $\mathcal{D}$ satisfies almost strong accuracy if:* $\forall F \in \mathcal{E}, \forall H \in \mathcal{D}(F), \forall t \in \mathcal{T}, \forall p \in correct(F), \forall q \in \Pi : p \notin H(q, t)$

This definition was the definition of strong accuracy in [24].

For all these aformentioned properties, we can assume, without loss of generality, that when a process is suspected it remains suspected forever.

We now recall the definition of the perfect and the strong failure detectors [7] and we introduce our almost perfect failure detector:

Definition 5 (Perfect). *A failure detector is said to be* perfect *if it satisfies the strong completeness and the strong accuracy properties. This failure detector is denoted by $\mathcal{P}$.*

Definition 6 (Almost Perfect). *A failure detector is said to be* almost perfect *if it satisfies the strong completeness and the almost strong accuracy properties. This failure detector is denoted by $\mathcal{P}^-$.*

Note that $\mathcal{P}-$ was given as the definition of the perfect failure detector in the very first paper on unreliable failure detector in [24]. In fact, failure detector in $\mathcal{P}^-$ can suspect faulty processes before they crash and be *unrealistic* according to the definition in [25].

Definition 7 (Strong). *A failure detector is said to be* strong *if it satisfies the strong completeness and the weak accuracy properties. This failure detector is denoted by $\mathcal{S}$.*

Algorithms, Runs, and Specification. A distributed algorithm is a collection of n sequential and deterministic algorithms, one for each process in Π. Computations of distributed algorithm $\mathcal{A}$ proceed in *atomic steps*.

In a step, a process p executes each of the following actions at most once: p try to receive a message from another process, p queries its failure detector module, p modifies its (local) state. and p sends a message to another process.

A *run* of Algorithm $\mathcal{A}$ using a failure detector $\mathcal{D}$ is a tuple $\langle F, H_{\mathcal{D}}, \gamma_{init}, E, T\rangle$ where F is a failure pattern, $H_{\mathcal{D}} \in \mathcal{D}(F)$ is an history of failure detector $\mathcal{D}$ for the failure pattern F, γ_{init} is an *initial configuration* of $\mathcal{A}$, E is an infinite sequence of steps of A, and T is a list of increasing time values indicating when each step in E occurred. A run must satisfy certain well-formedness and fairness properties. In particular:

1. E is applicable to γ_{init}.
2. A process cannot take steps after it crashes.
3. When a process takes a step and queries its failure detector module, it gets the current value output by its local failure detector module.
4. Every correct process takes an infinite number of local steps in E.
5. Any message sent is eventually received or lost.

A *problem* P is defined by a set of properties that runs must satisfy. An algorithm A *solves a problem* P using a failure detector $\mathcal{D}$ if and only if all the runs of A using $\mathcal{D}$ satisfy the properties required by P.

A failure detector $\mathcal{D}$ is said to be *weaker* than another failure detector $\mathcal{D}'$ (denote $\mathcal{D} \leq \mathcal{D}'$) if there is an algorithm that uses only $\mathcal{D}'$ to emulate the output of $\mathcal{D}$ for every failure pattern. If $\mathcal{D}$ is weaker than $\mathcal{D}'$ but $\mathcal{D}'$ is not weaker than $\mathcal{D}$ we say that $\mathcal{D}$ is *strictly weaker* than $\mathcal{D}'$ (denote $\mathcal{D} < \mathcal{D}'$).

From [7] and our definition of $\mathcal{P}^-$, we get:

Proposition 1

$$\mathcal{S} < \mathcal{P}^- < \mathcal{P}$$

The *weakest* [8] failure detector $\mathcal{D}$ to solve a given problem is a failure detector $\mathcal{D}$ that is sufficient to solve the problem and that is also necessary to solve the problem, i.e. $\mathcal{D}$ is weaker than any failure detector that solves the problem.

Notations. In the sequel, v_p denotes the value of the variable v at process p. Finally, a datum in a message can be replaced by "$-$" when this value has no impact on the reasonning.

3 Problem Specifications

Reliable Broadcast. The reliable broadcast [26] is defined with two primitives: `BROADCAST`(m) and `DELIVER`(m). Informally, any reliable broadcast algorithm guarantees that after a process p invokes `BROADCAST`(m), every correct process eventually executes `DELIVER`(m). In the formal definition below, we denote by $sender(m)$ the process that invokes `BROADCAST`(m).

Specification 1 (Reliable Broadcast). *A run R satisfies the specification Reliable Broadcast if and only if the following three requirements are satisfied in R:*

- Validity: *If a correct process invokes* `BROADCAST`(m)*, then it eventually executes* `DELIVER`(m)*.*

- (Uniform) Agreement: *If a process executes* DELIVER(m)*, then all other correct processes eventually execute* DELIVER(m).
- Integrity: *For every message* m*, every process executes* DELIVER(m) *at most once, and only if sender*(m) *previously invokes* BROADCAST(m).

Consensus. In the consensus problem, all correct processes *propose* a value and must reach a unanimous and irrevocable *decision* on some value that is chosen between the proposed values. We define the consensus problem in terms of two primitives, PROPOSE(v) and DECIDE(u). When a process executes PROPOSE(v), we say that it *proposes* v; similarly, when a process executes DECIDE(u), we say that it *decides* u.

Specification 2 (Consensus). *A run R satisfies the specification Consensus if and only if the following three requirements are satisfied in R:*

- (Uniform) Agreement: *No two processes* decide *differently.*
- Termination: *Every correct process eventually* decides *some value.*
- Validity: *If a process* decides v*, then* v *was* proposed *by some process.*

Repeated Consensus. We now define repeated consensus. Each correct process has as input an infinite sequence of proposed values, and outputs an infinite sequence of decision values such that:

1. Two correct processes have the same output. (The output of a faulty process is a prefix of this output.)
2. The i^{th} value of the output is the i^{th} value of the input of some process.

We define the repeated consensus in terms of two primitives, R-PROPOSE(v) and R-DECIDE(u). When a process executes the i^{th} R-PROPOSE(v), v is the i^{th} value of its input (we say that it *proposes* v for the i^{th} consensus); similarly, when a process executes the i^{th} R-DECIDE(u) u is the i^{th} value of its output (we say that it *decides* v for the i^{th} consensus).

Specification 3 (Repeated Consensus). *A run R satisfies the specification Repeated Consensus if and only if the following three requirements are satisfied in R:*

- Agreement: *If u and v are the outputs of two processes, then u is a prefix of v or v is a prefix of u.*
- Termination: *Every correct process has an infinite output.*
- Validity: *If the i^{th} value of the output of a process is v, then v is the i^{th} value of the input of some process.*

4 Reliable Broadcast in $\Phi^{\mathcal{F}}$

In this section, we show that $\mathcal{P}^-$ is the weakest failure detector to solve the reliable broadcast in $\Phi^{\mathcal{F}}$.

$\mathcal{P}^-$ *is Necessary.* To show that $\mathcal{P}^-$ is necessary to solve the reliable broadcast the following lemma is central to the proof:

Lemma 1. *Let $\mathcal{A}$ be an algorithm solving Reliable Broadcast in $\Phi^{\mathcal{F}}$ with a failure detector $\mathcal{D}$. There exists an integer k such that for every process p and every correct process q, for every run R of $\mathcal{A}$ where process p* BROADCAST*s and* DELIVER*s k messages, at least one message from q has been received by some process.*

Assume now that there exists an algorithm $\mathcal{A}$ that implements the *reliable broadcast* in $\Phi^{\mathcal{F}}$ using the failure detector $\mathcal{D}$. To show our result we have to give an algorithm that uses only $\mathcal{D}$ to emulate the output of $\mathcal{P}^-$ for every failure pattern.

Actually, we give an algorithm $\mathcal{A}_{(p,q)}$ (Figure 1) where a given process p monitors a given process q. This algorithm uses one instance of $\mathcal{A}$ with $\mathcal{D}$. Note that all processes except q participate to this algorithm following the code of $\mathcal{A}$. In this algorithm $Output\_q$ is equal to either $\{q\}$ (q is faulty) or $\emptyset$ (q is correct).

The algorithm $\mathcal{A}_{(p,q)}$ works as follows: p tries to BROADCAST k messages, all processes execute the code of the algorithm $\mathcal{A}$ using $\mathcal{D}$ except q that does nothing. If p DELIVERs k messages, it sets $Output\_q$ to q and never changes the values of $Output\_q$. By lemma 1, if q is correct p can't DELIVER k messages and so it never sets $Output\_q$ to $\{q\}$. If q is faulty and p is correct: as $\mathcal{A}$ solve reliable broadcast, p has to deliver DELIVER k messages and so p sets $Output\_q$ to $\{q\}$.[4]

To emulate $\mathcal{P}^-$, each process p uses algorithm $\mathcal{A}_{(p,q)}$ for every process q. As $\mathcal{D}$ is a failure detector it can be used for each instance. The output of $\mathcal{P}^-$ at p (variable $Output$) is then the union of $Output\_q$ for every process q.

```
1:      /* Code for process p */
2: begin
3:    Output_q ← ∅
4:    for i = 1 to k do
5:       BROADCAST(m)      /* using A with D */
6:       wait for DELIVER(m)
7:    end for
8:    Output_q ← {q}
9: end
10:     /* Code for process q */
11: begin
12: end
13:     /* Code for every process Π − {p, q} */
14: begin
15:    execute the code of A with D for these messages
16: end
```

Fig. 1. $\mathcal{A}_{(p,q)}$

Theorem 1. *$\mathcal{P}^-$ is necessary to solve Reliable Broadcast in $\Phi^{\mathcal{F}}$.*

$\mathcal{P}^-$ *is Sufficient.* In Algorithm $\mathcal{B}$ (Figure 2), every process uses a failure detector module of type $\mathcal{P}^-$ and a finite memory. Theorem 2 shows that Algorithm $\mathcal{B}$ solves the reliable broadcast in $\Phi^{\mathcal{F}}$ and directly implies that $\mathcal{P}^-$ is sufficient to solve the reliable broadcast in $\Phi^{\mathcal{F}}$ (Corollary 1).

[4] If q is faulty and p is faulty, the property of failure detector is trivially ensured.

```
1:      /* Code for every process q */
2: variables:
3:      Flag[1...n][1...n] ∈ {0,1}^(n^2); ∀(i,j) ∈ Π^2, Flag[i][j] is initialized to 0
4:      FD: failure detector of type P^-
5:      Mes[1...n]: array of data messages; ∀i ∈ Π, Mes[i] is initialized to ⊥
6: function:
7:      MesToBrd(): returns a message or ⊥
8: begin
9:    repeat forever
10:       if Mes[p] =⊥ then
11:          Mes[p] ← MesToBrd()
12:          if Mes[p] ≠⊥ then
13:             Flag[p][p] ← (Flag[p][p] + 1) mod 2
14:          end if
15:       end if
16:       for all i ∈ Π \ FD do
17:          for all j∈Π\(FD∪{p,i}),Flag[i][p]≠Flag[i][j] do
18:             if (receive⟨i-ACK, F⟩ from j) ∧ (F=Flag[i][p]) then
19:                Flag[i][j] ← F
20:             else
21:                send⟨i-BRD, Mes[i], Flag[i][p]⟩ to j
22:             end if
23:          end for
24:          if (Mes[i] ≠⊥) ∧ (∀q ∈ Π \ FD, Flag[i][i] = Flag[i][q]) then
25:             DELIVER(Mes[i]); Mes[i] ←⊥
26:          end if
27:       end for
28:       for all i ∈ Π \ FD \ {p} do
29:          for all j ∈ Π \ (FD ∪ {p}) do
30:             if (receive⟨i-BRD, m, F⟩ from j) then
31:                if (∀q ∈ Π \ FD, Flag[i][q] = Flag[i][i]) ∧ (F ≠ Flag[i][p]) then
32:                   Mes[i] ← m; Flag[i][p] ← F
33:                end if
34:                if i = j then
35:                   Flag[i][i] ← F
36:                end if
37:                if (i ≠ j) ∨ (∀q ∈ Π \ FD, Flag[i][q] = Flag[i][i]) then
38:                   send⟨i-ACK, Flag[i][p]⟩ to j
39:                end if
40:             end if
41:          end for
42:       end for
43:    end repeat
44: end
```

Fig. 2. Algorithm $\mathcal{B}$

In Algorithm $\mathcal{B}$, each process p executes broadcasts sequentially: p starts a new broadcast only after the termination of the previous one. To that goal, any process p initializes $\texttt{Mes}[p]$ to $\perp$. Then, p periodically checks if an external application invokes $\texttt{BROADCAST}(-)$. In this case, $\texttt{MesToBrd}()$ returns the message to broadcast, say m. When this event occurs, $\texttt{Mes}[p]$ is set to m and the broadcast procedure starts. $\texttt{Mes}[p]$ is set to $\perp$ at the end of the broadcast, p checks again, and so on.

Algorithm $\mathcal{B}$ has to deal with two types of faults: process crashes and message loss.

- *Dealing with process crashes.* Every process uses a failure detector of type $\mathcal{P}^-$ to detect the process crashes. Note that, as mentionned in Section 2,

we assume that when a process is suspected by some process it remains suspected forever.

Assume that a process p broadcasts the message m: p sends a broadcast message (p-BRD) with the datum m to any process it believes to be correct.

In Algorithm $\mathcal{B}$, p executes DELIVER(m) only after all other processes it does not suspect receive m. To that goal, we use acknowledgment mechanisms. When p received an acknowledgment for m (p-ACK) from every other process it does not suspect, p executes DELIVER(m) and the broadcast of m terminates (*i.e.*, $\texttt{Mes}[p]$ is set to $\perp$).

To ensure the *agreement* property, we must guarantee that if p crashes but another process q already executes DELIVER(m), then any correct process eventually executes DELIVER(m). To that goal, any process can execute DELIVER(m) only after all other processes it does not suspect except p receive m. Once again, we use acknowledgment mechanisms to that end: q also broadcasts m to every other process it does not suspect except p (this induces that a process can now receive m from a process different of p) until receiving an acknowledgment for m from all these processes and the broadcast message from p if q does not suspect it.

To prevent m to be still broadcasted when p broadcasts the next message, we synchronize the system as follows: any process acknowledges m to p only after it received an acknowledgment for m from every other process it does not suspect except p. By contrast, if a process i receives a message broadcasted by p (p-BRD) from the process $j \neq p$, i directly acknowledges the message to j.

- *Dealing with message loss.* The broadcast messages have to be periodically retransmitted until they are acknowledged. To that goal, any process q stores the last broadcasted message from p into its variable $\texttt{Mes}_q[p]$ (initialized to $\perp$). However, some copies of previously received messages can be now in transit at any time in the network. So, each process must be able to distinguish if a message it receives is copy of a previously received message or a new one, say *valid*. To circumvent this problem, we use the traditional alternating-bit mechanism [27,28]: a flag value (0 or 1) is stored into any message and a two-dimentionnal array, noted $\texttt{Flag}[1 \ldots n][1 \ldots n]$, allows us to distinguish if the messages are *valid* or not. Initially, any process sets $\texttt{Flag}[i][j]$ to 0 for all pairs of processes (i,j). In the code of process p, the value $\texttt{Flag}_p[p][p]$ is used to mark every p-BRD messages sent by p. In the code of every process $q \neq p$, $\texttt{Flag}_q[p][q]$ is equal to the flag value of the last valid p-BRD message q receives (not necessarily from p). For all $q' \neq q$, $\texttt{Flag}_q[p][q']$ is equal to the flag value of the last valid p-BRD message q receives from q'.

  At the beginning of any broadcast at p, p increments $\texttt{Flag}_p[p][p]$ modulus 2. The broadcast terminates at p when for every other process q that p does not suspect, $\texttt{Flag}_p[p][q] = \texttt{Flag}_p[p][p]$, $\texttt{Flag}_p[p][q]$ being set to $\texttt{Flag}_p[p][p]$ only when p received a valid acknowledgement from q, *i.e.*, an acknowledgment marked with the value $\texttt{Flag}_p[p][p]$.

  Upon receiving a p-BRD message marked with the value F, a process $q \neq p$ detects that it is a new valid message broadcasted by p (but not necessarily

sent by p) if for every non-suspected process j, ($\texttt{Flag}_q[p][j] = \texttt{Flag}_q[p][p]$) and ($F \neq \texttt{Flag}_q[p][q]$). In this case, p sets $\texttt{Mes}_q[p]$ to m and sets $\texttt{Flag}_q[p][q]$ to F. From this point on, q periodically sends $\langle p\text{-}\texttt{BRD}, \texttt{Mes}_q[p], \texttt{Flag}_q[p][q]\rangle$ to any other process it does not suspect except p until receiving a valid acknowledgment (*i.e.*, an acknowledgment marked with the value $\texttt{Flag}_q[p][q]$) from all these processes. For any non-suspected process j different from p and q, $\texttt{Flag}_q[p][j]$ is set to $\texttt{Flag}_q[p][q]$ when q received an acknowledgment marked with the value $\texttt{Flag}_q[p][q]$ from j. Finally, $\texttt{Flag}_q[p][p]$ is set to $\texttt{Flag}_q[p][q]$ when q received the broadcast message from p (marked with the value $\texttt{Flag}_q[p][q]$). Hence, q can execute $\texttt{DELIVER}(\texttt{Mes}[p])$ when ($\texttt{Mes}[p] \neq \perp$) and ($\forall j \in \Pi \setminus \texttt{FD}$, $\texttt{Flag}_q[p][j] = \texttt{Flag}_q[p][p]$) because (1) it receives a valid broadcast message from p if p was not suspected and it has the guarantee that any non-suspected process different of p receives m in a valid message. To ensure that q executes $\texttt{DELIVER}(\texttt{Mes}[p])$ at most one, q just has to set $\texttt{Mes}[p]$ to $\perp$ after.

It is important to note that q acknowledges the valid p-$\texttt{BRD}$ messages it receives from p only when the predicate ($\forall j \in \Pi \setminus \texttt{FD}$, $\texttt{Flag}_q[p][j] = \texttt{Flag}_q[p][p]$) holds. However, to guarantee the liveness, q acknowledges any p-$\texttt{BRD}$ message that it receives from any other process. Every p-$\texttt{ACK}$ messages sent by q is marked with the value $\texttt{Flag}_q[p][q]$.

Finally, p stops its current broadcast when the following condition holds: ($\texttt{Mes}_p[p] \neq \perp$) $\wedge$ ($\forall q \in \Pi \setminus \texttt{FD}$, $\texttt{Flag}_p[p][p] = \texttt{Flag}_p[p][q]$), *i.e.*, any non-suspected process has acknowledged $\texttt{Mes}_p[p]$. In this case, p sets $\texttt{Mes}[p]$ to $\perp$.

Theorem 2. *Algorithm $\mathcal{B}$ is a Reliable Broadcast algorithm in $\Phi^{\mathcal{F}}$ with $\mathcal{P}^-$.*

Corollary 1. *$\mathcal{P}^-$ is sufficient for solving Reliable Broadcast in $\Phi^{\mathcal{F}}$.*

5 Consensus in $\Phi^{\mathcal{F}}$

In this section, we show that we can solve consensus in system $\Phi^{\mathcal{F}}$ with a failure detector that is strictly weaker than the failure detector necessary to solve reliable broadcast and repeated consensus. We solve consensus with the strong failure detector $\mathcal{S}$. $\mathcal{S}$ is not the weakest failure detector to solve consensus whatever the number of crash but it is strictly weaker than $\mathcal{P}^-$ and so enough to show our results.

We customize the algorithm of Chandra and Toueg [7] that works in an asynchronous message-passing system with reliable links and augmented with a strong failure detector ($\mathcal{S}$), to our model.

In this algorithm, called $\mathcal{CS}$ in the following (Figure 3), the processes execute n asynchronous rounds. First, processes execute $n-1$ asynchronous rounds (r denotes the current round number) during which they broadcast and relay their proposed values. Each process p waits until it receives a round r message from every other non-suspected process (*n.b.* as mentionned in Section 2, we assume that when a process is suspected it remains suspected forever) before proceeding

to round $r+1$. Then, processes execute a last asynchronous round during which they eliminate some proposed values. Again each process p waits until it receives a round n message from every other process it does not suspected. By the end of these n rounds, correct processes *agree* on a vector based on the proposed values of all processes. The i^{th} element of this vector either contains the proposed value of process i or $\perp$. This vector contains the proposed value of *at least one process*: a process that is never suspected by all processes. Correct processes decide the first nontrivial component of this vector.

To customize this algorithm to our model, we have to ensure the progress of each process: While a process has not ended the asynchronous round r it must be able to retransmit all the messages[5] that it has previously sent in order to allow others processes to progress despite the link failures. As we used a strong failure detector and unreliable but fair links, it is possible that one process has decided and the other ones still wait messages of asynchronous rounds. When a process has terminated the n asynchronous rounds, it uses a special `Decide` message to allow others processes to progress.

In the algorithm, we first use a function *consensus*. This function takes the proposed value in parameter and returns the decision value and uses a failure detector. Then, at processes request, we propagate the `Decide` message.

Theorem 3 shows that Algorithm $\mathcal{CS}$ solves the consensus in $\Phi^{\mathcal{F}}$ and directly implies that $\mathcal{S}$ is sufficient to solve te consensus problem in $\Phi^{\mathcal{F}}$ (Corollary 2).

Theorem 3. *Algorithm $\mathcal{CS}$ is a Consensus algorithm in $\Phi^{\mathcal{F}}$ with $\mathcal{S}$.*

Corollary 2. *$\mathcal{S}$ is sufficient for solving Consensus in $\Phi^{\mathcal{F}}$.*

6 Repeated Consensus in $\Phi^{\mathcal{F}}$

We show in this section that $\mathcal{P}^-$ is the weakest failure detector to solve the reliable consensus problem in $\Phi^{\mathcal{F}}$.

$\mathcal{P}^-$ is Necessary. The proof is similar to the one in Section 4, and here the following lemma is central to the proof:

Lemma 2. *Let $\mathcal{A}$ be an algorithm solving Repeated Consensus in $\Phi^{\mathcal{F}}$ with a failure detector $\mathcal{D}$. There exists an integer k such that for every process p and every correct process q for every run R of $\mathcal{A}$ where process p* `R-PROPOSE`*s and* `R-DECIDE`*s k times, at least one message from q has been received by some process.*

Assume that there exists an algorithm $\mathcal{A}$ that implements *Repeated Consensus* in $\Phi^{\mathcal{F}}$ using the failure detector $\mathcal{D}$. To show our result we have to give an algorithm that uses only $\mathcal{D}$ to emulate the output of $\mathcal{P}^-$ for every failure pattern.

In fact we give an algorithm $\mathcal{A}_q$ (Figure 4) where processes monitor a given process q. This algorithm uses one instance of $\mathcal{A}$ with $\mathcal{D}$. Note that all processes except q participate to this algorithm following the code of $\mathcal{A}$. In this algorithm *Output_q* is equal to either $\{q\}$ (q is crashed) or $\emptyset$ (q is correct).

[5] Notice that they are in finite number.

```
1:      /* Code for process p */
2: function consensus(v) with the failure detector fd
3:     variables:
4:         Flag[1 . . . n] ∈ {true,false}^n; ∀i ∈ Π, Flag[i] is initialized to false
5:         V[1 . . . n]: array of propositions; ∀i ∈ Π, V[i] is initialized to ⊥
6:         Mes[1 . . . n]: array of arrays of propositions; ∀i ∈ Π, Mes[i] is initialized to ⊥
7:         r: integer; r is initialized to 1
8:     begin
9:         V[p] ← v the proposed values
10:        Mes[1] ← V
11:        while (r ≤ n) do
12:            send⟨R-r, Mes[r]⟩ to every process, except p
13:            for all i ∈ Π \ (fd ∪ {p}), Flag[i] = false do
14:                if (receive⟨R-r, W⟩ from i) then
15:                    Flag[i] ← true
16:                    if r < n then
17:                        if V[i] = ⊥ then
18:                            V[i] ← W[i]; Mes[r + 1][i] ← W[i]
19:                        end if
20:                    else
21:                        if V[i] ≠ ⊥ then
22:                            V[i] ← W[i]
23:                        end if
24:                    end if
25:                end if
26:            end for
27:            for all i ∈ Π \ {p} do
28:                if (receive⟨R-x, W⟩ from i), x < r then
29:                    send⟨R-x, Mes[x]⟩ to i
30:                end if
31:            end for
32:            if ∀q ∈ Π \ (fd ∪ {p}), Flag[q] = true then
33:                if r < n then
34:                    for all i ∈ Π do
35:                        Flag[i] ← false
36:                    end for
37:                end if
38:                if r = n − 1 then
39:                    Mes[n] ← V
40:                end if
41:                r ← r + 1
42:            end if
43:            for all i ∈ Π \ {p} do
44:                if (receive⟨Decide, d⟩ from i) then
45:                    return(d)
46:                end if
47:            end for
48:        end while
49:        d ← the first component of V different from ⊥; return(d)
50:    end
51: end function
52: procedure PROPOSE(v)
53:    variables: u: integer; FD: failure detector of type S
54:    begin
55:        u ←consensus(v) with FD
56:        DECIDE(u)
57:        repeat forever
58:            for all j ∈ Π \ {p}, x ∈ {1, ..., n} do
59:                if (receive⟨R-x, W⟩ from j) then
60:                    send⟨Decide, u⟩ to j
61:                end if
62:            end for
63:        end repeat
64:    end
65: end procedure
```

Fig. 3. Algorithm $\mathcal{CS}$, *Consensus* with $\mathcal{S}$

The algorithm $\mathcal{A}_q$ works as follows: processes try to R-DECIDE k times, all processes execute the code of the algorithm $\mathcal{A}$ using $\mathcal{D}$ except q that does nothing. If p R-DECIDE k messages, it sets $Output\_q$ to q and never changes the values of $Output\_q$.

By lemma 2, if q is correct p cannot decides k times and so it never sets $Output\_q$ to q. If q is faulty and p is correct[6]: as $\mathcal{A}$ solve *Repeated Consensus*, p has to R-DECIDE k times and so p sets $Output\_q$ to $\{q\}$.

To emulate $\mathcal{P}^-$, each process p uses Algorithm $\mathcal{A}_q$ for every process q. As $\mathcal{D}$ is a failure detector it can be used for each instance. The output of $\mathcal{P}^-$ at p (variable $Output$) is then the union of $Output\_q$ for every process q.

```
1:      /* Code for process p of Π \ q */
2: begin
3:    Output_q ← ∅
4:    for i = 1 to k do
5:       R-PROPOSED(v)      /* using A with D */
6:       wait for R-DECIDE(v)
7:    end for
8:    Output_q ← {q}
9: end
10:     /* Code for process q */
11: begin
12: end
```

Fig. 4. $\mathcal{A}_q$

Theorem 4. *$\mathcal{P}^-$ is necessary to solve Repeated Consensus problem in $\Phi^{\mathcal{F}}$.*

$\mathcal{P}^-$ is Sufficient. In this section, we show that $\mathcal{P}^-$ is sufficient to solve the repeated consensus in $\Phi^{\mathcal{F}}$. To that goal, we consider an algorithm called Algorithm $\mathcal{RCP}$ (Figures 5 and 6). In this algorithm, any process uses a failure detector module of type $\mathcal{P}^-$ (again we assume that since a process is suspected by some process it is suspected forever) and a finite memory. Theorem 5 shows that Algorithm $\mathcal{RCP}$ solves the repeated consensus in $\Phi^{\mathcal{F}}$ and directly implies that $\mathcal{P}^-$ is sufficient to solve the repeated consensus in $\Phi^{\mathcal{F}}$ (Corollary 3).

We assume that each correct processes has an infinite sequence of input and when it terminates R-PROPOSED(v) where v is the i^{th} value of its input, it executes R-PROPOSED(w) where w is the $(i+1)^{th}$ value of its input.

When a process executes R-PROPOSED(v), it first executes a consensus in which it proposes v. The decision of this consensus is then outputted. Then, processes have to avoid that the messages of two consecutive consensus are mixed up. We construct a synchronization barrier. Without message loss, and with a perfect failure detector, it is sufficient that each process waits a `Decide` message from every process trusted by its failure detector module. By FIFO property, no message $\langle$R-$x, -\rangle$ sent before this `Decide` message can be received in the next consensus.

[6] If q is faulty and p is faulty, the property of failure detector is trivially ensured.

```
1:      /* Code for process p */
2: function consensus(v) with the failure detector fd
3:    variables:
4:       Flag[1 . . . n] ∈ {true,false}^n; ∀i ∈ Π, Flag[i] is initialized to false
5:       V[1 . . . n]: array of propositions; ∀i ∈ Π, V[i] is initialized to ⊥
6:       Mes[1 . . . n]: array of arrays of propositions; ∀i ∈ Π, Mes[i] is initialized to ⊥
7:       r: integer; r is initialized to 1
8:    begin
9:       V[p] ← v the proposed values; Mes[1] ← V
10:      while (r ≤ n) do
11:         send⟨R-r, Mes[r]⟩ to every process, except {p} ∪ fd
12:         for all i ∈ Π \ (fd ∪ {p}), Flag[i] = false do
13:            if (receive⟨R-r, W⟩ from i) then
14:               Flag[i] ← true
15:               if r < n then
16:                  if V[i] = ⊥ then
17:                     V[i] ← W[i]; Mes[r + 1][i] ← W[i]
18:                  end if
19:               else
20:                  if V[i] ≠ ⊥ then
21:                     V[i] ← W[i]
22:                  end if
23:               end if
24:            end if
25:         end for
26:         if ∀q ∈ Π \ (fd ∪ {p}), Flag[q] = true then
27:            if r < n then
28:               for all i ∈ Π do
29:                  Flag[i] ← false
30:               end for
31:            end if
32:            if r = n − 1 then
33:               Mes[n] ← V
34:            end if
35:            r ← r + 1
36:         end if
37:         for all i ∈ Π \ (fd ∪ {p}) do
38:            if (receive⟨Decide, d⟩ from i) then
39:               return(d)
40:            end if
41:         end for
42:      end while
43:      d ← the first component of V different from ⊥; return(d)
44:   end
45: end function
```

Fig. 5. Algorithm $\mathcal{RCP}$, *Repeated Consensus* with $\mathcal{P}^-$. Part 1: function consensus().

To deal with message loss, the synchronization barrier is obtained by two asynchronous rounds: In the first asynchronous rounds, each process sends a `Decide` message and waits to receive a `Decide` message or a `Start` message from every other process it does not suspect. In the second one, each process sends a `Decide` message and waits to receive a `Start` message or a $\langle \texttt{R-}x, - \rangle$ message. Actually, due to message loss it is possible that a process goes to its second round despite some process have not received its `Decide` message, but it cannot finish the second round before every correct processes have finished the first one.

As a faulty process can be suspected before it crashes (due to the quality of $\mathcal{P}^-$), it is possible that a faulty process will not be waited by other processes although it is still alive. To avoid that this process disturbs the round, since a process p suspects a process q, p stops to wait messages from q and to send messages to q.

```
1:      /* Code for process p */
2: variables:
3:     FD: failure detector of type P⁻
4: procedure R-PROPOSED(v)
5:     variables:
6:         FlagR[1 . . . n] ∈ {true,false}^n; ∀i ∈ Π, FlagR[i] is initialized to false
7:         stop: boolean; stop is initialized to false
8:         u: integer;
9:     begin
10:        u ←consensus(v) with FD
11:        R-DECIDE(u)
12:        repeat
13:            send⟨Decide, u⟩ to every process, except {p} ∪ FD
14:            for all i ∈ Π \ (FD ∪ {p}), FlagR[i] = false do
15:                if (receive⟨Decide, u⟩ from i) ∨ (receive⟨Start⟩ from i) then
16:                    FlagR[i] ← true
17:                end if
18:            end for
19:            if ∀q ∈ Π \ (FD ∪ {p}), FlagR[q] = true then
20:                stop ← true
21:            end if
22:        until stop
23:        for all i ∈ Π do
24:            FlagR[i] ← false
25:        end for
26:        stop ← false
27:        repeat
28:            send⟨Start⟩ to every process, except {p} ∪ FD
29:            for all i ∈ Π \ (FD ∪ {p}), FlagR[i] = false do
30:                if (receive⟨Start⟩ from i) ∨ (receive⟨R-1, W⟩ from j) then
31:                    FlagR[i] ← true
32:                end if
33:            end for
34:            if ∀q ∈ Π \ (FD ∪ {p}), FlagR[q] = true then
35:                stop ← true
36:            end if
37:        until stop
38:    end
39: end procedure
```

Fig. 6. Algorithm $\mathcal{RCP}$, *Repeated Consensus* with $\mathcal{P}^-$. Part 2.

Note also that if the consensus function is executed with $\mathcal{P}^-$, then there is no need to send $\langle \mathtt{R}\text{-}x, -\rangle$ in round $r > x$ again. We have rewritten the consensus function to take account of these facts, but the behaviour remains the same.

Theorem 5. *Algorithm $\mathcal{RCP}$ (Figure 5 and 6) is a Repeated Consensus algorithm in $\Phi^{\mathcal{F}}$ with $\mathcal{P}^-$.*

Corollary 3. *$\mathcal{P}^-$ is sufficient for solving Repeated Consensus in $\Phi^{\mathcal{F}}$.*

Contrary to these results in system $\Phi^{\mathcal{F}}$, in system $\Phi^{\mathcal{I}}$, we have the same weakest failure detector to solve the consensus problem and the repeated consensus problem:

Proposition 2. *In system $\Phi^{\mathcal{I}}$, if there is an algorithm $\mathcal{A}$ with failure detector $\mathcal{D}$ solving Consensus, then there exists an algorithm solving Repeated Consensus with $\mathcal{D}$.*

References

1. Guerraoui, R., Schiper, A.: The generic consensus service. IEEE Transactions on Software Engineering 27(1), 29–41 (2001)
2. Gafni, E., Lamport, L.: Disk paxos. Distributed Computing 16(1), 1–20 (2003)
3. Fischer, M.J., Lynch, N.A., Paterson, M.: Impossibility of distributed consensus with one faulty process. Journal of the ACM 32(2), 374–382 (1985)
4. Chor, B., Coan, B.A.: A simple and efficient randomized byzantine agreement algorithm. IEEE Trans. Software Eng. 11(6), 531–539 (1985)
5. Dolev, D., Dwork, C., Stockmeyer, L.J.: On the minimal synchronism needed for distributed consensus. Journal of the ACM 34(1), 77–97 (1987)
6. Dwork, C., Lynch, N.A., Stockmeyer, L.J.: Consensus in the presence of partial synchrony. Journal of the ACM 35(2), 288–323 (1988)
7. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM 43(2), 225–267 (1996)
8. Chandra, T.D., Hadzilacos, V., Toueg, S.: The weakest failure detector for solving consensus. Journal of the ACM 43(4), 685–722 (1996)
9. Delporte-Gallet, C., Fauconnier, H., Guerraoui, R.: Shared memory vs message passing. Technical report, LPD-REPORT-2003-001 (2003)
10. Eisler, J., Hadzilacos, V., Toueg, S.: The weakest failure detector to solve nonuniform consensus. Distributed Computing 19(4), 335–359 (2007)
11. Delporte-Gallet, C., Fauconnier, H., Guerraoui, R., Hadzilacos, V., Kouznetsov, P., Toueg, S.: The weakest failure detectors to solve certain fundamental problems in distributed computing. In: Twenty-Third Annual ACM Symposium on Principles of Distributed Computing (PODC 2004), pp. 338–346 (2004)
12. Aguilera, M.K., Toueg, S., Deianov, B.: Revisiting the weakest failure detector for uniform reliable broadcast. In: Jayanti, P. (ed.) DISC 1999. LNCS, vol. 1693, pp. 13–33. Springer, Heidelberg (1999)
13. Halpern, J.Y., Ricciardi, A.: A knowledge-theoretic analysis of uniform distributed coordination and failure detectors. In: Eighteenth Annual ACM Symposium on Principles of Distributed Computing (PODC 1999), pp. 73–82 (1999)
14. Delporte-Gallet, C., Fauconnier, H., Guerraoui, R., Kouznetsov, P.: Mutual exclusion in asynchronous systems with failure detectors. Journal of Parallel and Distributed Computing 65(4), 492–505 (2005)
15. Guerraoui, R., Kapalka, M., Kouznetsov, P.: The weakest failure detectors to boost obstruction-freedom. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 399–412. Springer, Heidelberg (2006)
16. Raynal, M., Travers, C.: In search of the holy grail: Looking for the weakest failure detector for wait-free set agreement. In: Shvartsman, M.M.A.A. (ed.) OPODIS 2006. LNCS, vol. 4305, pp. 3–19. Springer, Heidelberg (2006)
17. Zielinski, P.: Anti-omega: the weakest failure detector for set agreement. Technical Report UCAM-CL-TR-694, Computer Laboratory, University of Cambridge, Cambridge, UK (July 2007)
18. Lynch, N.A., Mansour, Y., Fekete, A.: Data link layer: Two impossibility results. In: Symposium on Principles of Distributed Computing, pp. 149–170 (1988)
19. Bazzi, R.A., Neiger, G.: Simulating crash failures with many faulty processors (extended abstract). In: Segall, A., Zaks, S. (eds.) WDAG 1992. LNCS, vol. 647, pp. 166–184. Springer, Heidelberg (1992)
20. Delporte-Gallet, C., Devismes, S., Fauconnier, H., Petit, F., Toueg, S.: With finite memory consensus is easier than reliable broadcast. Technical Report hal-00325470, HAL (October 2008)

21. Cavin, D., Sasson, Y., Schiper, A.: Consensus with unknown participants or fundamental self-organization. In: Nikolaidis, I., Barbeau, M., Kranakis, E. (eds.) ADHOC-NOW 2004. LNCS, vol. 3158, pp. 135–148. Springer, Heidelberg (2004)
22. Greve, F., Tixeuil, S.: Knowledge connectivity vs. synchrony requirements for fault-tolerant agreement in unknown networks. In: DSN, pp. 82–91. IEEE Computer Society, Los Alamitos (2007)
23. Fernández, A., Jiménez, E., Raynal, M.: Eventual leader election with weak assumptions on initial knowledge, communication reliability, and synchrony. In: DSN, pp. 166–178. IEEE Computer Society, Los Alamitos (2006)
24. Chandra, T.D., Toueg, S.: Unreliable failure detectors for asynchronous systems (preliminary version). In: 10th Annual ACM Symposium on Principles of Distributed Computing (PODC 1991), pp. 325–340 (1991)
25. Delporte-Gallet, C., Fauconnier, H., Guerraoui, R.: A realistic look at failure detectors. In: DSN, pp. 345–353. IEEE Computer Society, Los Alamitos (2002)
26. Hadzilacos, V., Toueg, S.: A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR 94-1425, Department of Computer Science, Cornell University (1994)
27. Bartlett, K.A., Scantlebury, R.A., Wilkinson, P.T.: A note on reliable full-duplex transmission over halfduplex links. Journal of the ACM 12, 260–261 (1969)
28. Stenning, V.: A data transfer protocol. Computer Networks 1, 99–110 (1976)