# The Weakest Failure Detector for Non-Blocking Atomic Commit

Rachid Guerraoui  Petr Kouznetsov

Distributed Programming Laboratory
EPFL

**Abstract**

This paper addresses the question of the weakest failure detector for solving the Non-Blocking Atomic Commit problem (NBAC) in a message passing system where processes can fail by crashing.

We define a failure detector, denoted by $\mathcal{X}$, which we show to be *sufficient* to solve NBAC with a majority of correct processes. Then we give an algorithm which, no matter how many processes may crash, uses any failure detector that solves NBAC to emulate $\mathcal{X}$, i.e., we prove that $\mathcal{X}$ is also *necessary* for solving NBAC.

A similar result was obtained concurrently and independently by Hadzilacos and Toueg [11].

**Keywords:** atomic commitment, failure detector, non-blocking, asynchronous distributed system, consensus.

## 1 Introduction

**Problem.** *Non-Blocking Atomic Commit (NBAC)* is a fundamental *agreement* problem in distributed computing [12, 6, 10]. The problem occurs in distributed transactional systems. To ensure the atomicity of a distributed transaction, processes must agree on a common outcome: *commit* or *abort*. Every process that does not crash during the execution of the algorithm (i.e., a correct process), should eventually decide on an outcome without waiting for crashed processes to recover. Moreover, *commit* is decided only if no process proposes to abort, and *abort* is decided only if a process proposes to abort or crashes.

More precisely, the *Non-Blocking Atomic Commit (NBAC)* problem [12] consists for a set of processes to reach a common *decision*, *commit* or *abort*, according to some initial votes of the processes, *yes* or *no*, such that the following properties are satisfied: *(1) Agreement:* no two processes decide differently; *(2) Termination:* every correct process eventually decides; *(3) C-Validity: abort* can only be decided if some process votes *no* or crashes; and *(4) A-Validity: commit* can only be decided if all processes vote *yes*. For brevity, we denote *yes* and *commit* by 1, while *no* and *abort* by 0.

In this paper, we discuss the solvability of the problem in a *distributed crash-stop asynchronous message-passing* model. Informally, the model is one in which processes exchange messages through reliable communication channels, processes can fail by crashing, and there are no bounds on message transmission time and relative processor speeds.

**Failure detectors.** Clearly, NBAC is not solvable in this model if at least one process can crash. Indeed, there is no way to distinguish a single slow process proposing 1 from a crashed one. As a

result, a correct process might never be able to decide. Thus, NBAC belongs to the class of agreement problems that are not solvable in an asynchronous system where at least one process can crash [4] [1].

To circumvent the impossibility of agreement problems, Chandra and Toueg [2] introduced the notion of *failure detector*. Informally, a failure detector is a distributed oracle that gives (possibly incorrect) hints about the crashes of processes. Each process has access to a local *failure detector module* that monitors other processes in the system. A failure detector can be implemented with synchrony assumptions in a message passing system. It was shown in [2] that a rather weak failure detector $\diamond\mathcal{S}$ is *sufficient* to solve Consensus in an asynchronous system with a majority of correct processes, and in [1] that $\diamond\mathcal{S}$ is in some sense *necessary* to solve Consensus. More precisely, it was shown in [1] that any failure detector solving Consensus can emulate $\diamond\mathcal{S}$. In the parlance of [1], $\diamond\mathcal{S}$ is *the weakest* failure detector to solve Consensus: $\diamond\mathcal{S}$ encapsulates the exact information about failures needed to solve Consensus in a system with a majority of correct processes. $\diamond\mathcal{S}$ guarantees that, in any execution, the following properties are satisfied: (*strong completeness*) every crashed process is eventually suspected by every correct process, and (*eventual weak accuracy*) there is a time after which at least one correct process is never suspected.

In this paper we address the question of the weakest failure detector to solve NBAC. That is, we aim at determining the minimal amount of information about failures needed to solve NBAC.

**Background.** It was pointed out in [1, 7] that $\diamond\mathcal{S}$ was not sufficient to solve NBAC and the question for the weakest remained open since then.

In [8] the *anonymously perfect* failure detector $?\mathcal{P}$ was introduced, and shown to be *necessary* to solve NBAC. Each module of $?\mathcal{P}$ at a given process outputs either the empty set or the identifier of the process. When the failure detector module of $?\mathcal{P}$ at a process $p_i$ outputs $p_i$, we say that $p_i$ *detects a crash*. $?\mathcal{P}$ satisfies the following properties: (*anonymous completeness:*) if some process crashes, then there is a time after which every correct process permanently detects a crash, and (*anonymous accuracy:*) no crash is detected unless some process crashes. In other words, $?\mathcal{P}$ correctly detects that *some* process has crashed, but does not tell *which* process has actually crashed.

It was stated in [5] that $\mathcal{P}$ is the weakest failure detector to solve NBAC, but [8] pointed out that [5] assumes NBAC to be solved among any subset of the processes in the system, then showed that $\mathcal{P}$ is not the weakest failure detector to solve NBAC without that assumption. In fact, an algorithm that transforms Consensus into NBAC using $?\mathcal{P}$ was presented in [8]. Since $\diamond\mathcal{S}$ is sufficient to solve Consensus in a system with a majority of correct processes, $?\mathcal{P} + \diamond\mathcal{S}$ is sufficient to solve NBAC. It was also shown in [8] that (a) $?\mathcal{P} + \diamond\mathcal{S}$ is strictly weaker than $\mathcal{P}$(if more than one process can crash), and (b) yet $?\mathcal{P} + \diamond\mathcal{S}$ is not the weakest for NBAC (It is shown in [9] that $?\mathcal{P} + \diamond\mathcal{S}$ is necessary for NBAC among the class of *time-free* failure detectors, i.e., all failure detectors are not allowed to provide timing information).

If at most one process can crash, then $?\mathcal{P}$ can easily emulate $\diamond\mathcal{S}$. Given that $?\mathcal{P} + \diamond\mathcal{S}$ solves NBAC [8] in a system with a majority of correct processes, $?\mathcal{P}$ is hence the weakest failure detector to solve NBAC in a system of $n > 2$ processes where one process can crash.

This leaves open the general case however: what is the weakest failure detector, among all possible failure detectors (in the original sense of [2]), in a system where more than one process can crash? This paper answer the question for systems in which a majority of processes is correct..

**Contribution.** In this paper, we precisely define the weakest failure detector to solve NBAC with a majority of correct processes. This failure detector, denoted by $\mathcal{X}$, is strictly weaker than $?\mathcal{P} + \diamond\mathcal{S}$,

[1] In fact, the impossibility result of [4] is shown for *Weak Consensus* and applies to both *Consensus* and NBAC (though a simpler proof can clearly be devised for NBAC because of its validity properties). In Consensus, the processes need to decide on one out of two values, 0 or 1, based on proposed values, 0 or 1, so that, in addition to the *agreement* and *termination* properties of NBAC, the following *validity* property holds: A value decided must be a value proposed.

and thus than $\mathcal{P}$. Roughly speaking, $\mathcal{X}$ eventually *either* (1) accurately and uniformly *detects a crash or* (2) uniformly *misses a crash* (no crash is detected unless it happens and no two processes disagree). If a crash is detected, no further communication is needed to solve NBAC: processes can safely decide 0. Otherwise, $\mathcal{X}$ provides the same information as $?\mathcal{P} + \diamond\mathcal{S}$ provides. We show that $\mathcal{X}$ is sufficient to solve NBAC when a majority of processes is correct. We show in this paper that $\mathcal{X}$ is also necessary to solve NBAC (in any environment). That is, we prove that there is an algorithm that emulates $\mathcal{X}$ using any failure detector that solves NBAC.

The construction of our emulation algorithm is slightly more involved than the one of [1]. Intuitively, the emulation algorithm exploits the failure detection flavor of the NBAC problem itself. Indeed, if every process proposes 1 and the decision value of an NBAC algorithm using a failure detector $\mathcal{D}$ is 0, then, by the *c-validity* property of the problem, there is a failure. Moreover, by *termination* and *agreement*, it is guaranteed that any process either detects the failure or crashes. As a result, if the failure is detected, no further failure detection is necessary. Otherwise, if no failure is detected, it turns out that the information provided by $\mathcal{D}$ is sufficient to emulate $\diamond\mathcal{S}$. Indeed, under the condition that 1 is decided in some execution of the NBAC algorithm using $\mathcal{D}$, there exists an algorithm that eventually, at every correct process, outputs the same correct process identifier, which is sufficient to emulate $\diamond\mathcal{S}$ [1].

Our weakest failure detector for solving NBAC $\mathcal{X}$ provides an insight to the implementation aspect of the problem. For instance, in [3] a transaction commit algorithm was proposed. The algorithm ensures *a-validity*, *agreement* and, with probability 1, *termination* of NBAC. The *c-validity* property is guaranteed under the condition that the underlying system "behaves well": e.g., all messages arrive within some known time bound. (Note that there is no purely randomized solution to NBAC due to the "failure-sensitive" nature of the problem.) In a precise sense, $\mathcal{X}$ captures what "behaves well" means to make the algorithm of [3] solve the original NBAC problem.

**Important remark.** Concurrently and independently of this work, Vassos Hadzilacos and Sam Toueg came out with a similar result [11].

In fact, Hadzilacos and Toueg determined the weakest failure detecor for a slightly stronger version of the NBAC problem. With respect to the original definition of the problem given in [10], the definition of [11] does not allow processes to abort because of *future* failures. However, this limitation does not seem to be important from the practical point of view.

Another difference between our approach and the one of [11] has to do with the proof technique. Our proof directly attacks the NBAC problem whereas the proof of [11] goes through defining first the weakest failure detector for another problem, a weaker variant of consensus, which is by itself an interesting result. That makes the proof of [11] a bit more technically involved than our proof.

**Roadmap.** Section 2 defines our system model. Section 3 defines our candidate failure detector and shows that it is sufficient to solve NBAC. Section 4 gives a brief reminder of the technique used in [1] to show that $\diamond\mathcal{S}$ is necessary for solving Consensus and extends the technique to the NBAC problem. Section 5 proves the necessity part of our result.

## 2 Model

We consider in this paper a crash-prone asynchronous message passing model augmented with the failure detector abstraction. We recall here what in the model is needed to state and prove our results. More details on the model can be found in [2].

**System.** We assume the existence of a global clock to simplify the presentation. The processes do not have *direct* access to the clock (timing assumptions are captured within failure detectors).

We take the range $\mathcal{T}$ of the clock output values to be the set of natural numbers and the integer 0, $(\{0\} \cup \mathbb{N})$. The system consists of a set of $n$ processes $\Pi = \{p_1, .., p_n\} (n > 1)$. Every pair of processes is connected by a reliable communication channel. The systems is *asynchronous*: there is no time bound on message delay, clock drift, or the time necessary to execute a step [4].

**Failures and failure patterns.** Processes are subject to *crash* failures. A *failure pattern* $F$ is a function from the global time range $\mathcal{T}$ to $2^{\Pi}$, where $F(t)$ denotes the set of processes that have crashed by time $t$. Once a process crashes, it does not recover, i.e., $\forall t < t' : F(t) \subseteq F(t')$. We define $correct(F) = \Pi - \cup_{t \in \mathcal{T}} F(t)$ to be the set of *correct* processes. A process $p_i \notin F(t)$ is said to be *up* at time $t$. A process $p_i \in F(t)$ is said to be *crashed* (or *incorrect*) at time $t$. We do not consider Byzantine failures: a process either correctly executes the algorithm assigned to it, or crashes and stops executing any action forever. An *environment* $\mathcal{E}$ is a set of failure patterns. $\mathbb{E}_f$ denotes the set of all environments that include only failure patterns in which up to $f$ processes can fail. By default, we consider environments in $\mathbb{E}_f$ with at least one correct process: $0 \leq f < n$. We denote by $F_0$ the failure-free failure pattern $(correct(F_0) = \Pi)$.

**Failure detectors.** A *failure detector history $H$ with range $\mathcal{R}$* is a function from $\Pi \times \mathcal{T}$ to $\mathcal{R}$. $H(p_i, t)$ is the output of the failure detector module of process $p_i$ at time $t$. A *failure detector* $\mathcal{D}$ is a function that maps each failure pattern $F$ to a set of failure detector histories $\mathcal{D}(F)$ with range $\mathcal{R}_{\mathcal{D}}$.

Every process $p_i$ has a failure detector module $\mathcal{D}_i$ that $p_i$ queries to obtain information about the failures in the system. Typically, this information includes the set of processes that a process currently suspects to have crashed.[2] Among the failure detectors defined in [1, 2], we consider the following one:

Perfect ($\mathcal{P}$): the output of every $\mathcal{P}_i$ is a set of suspected processes satisfying *strong completeness* (i.e., every incorrect process is eventually suspected by *every* correct process) and *strong accuracy* (i.e., no process is suspected before it crashes);

Eventually strong ($\diamond\mathcal{S}$): the output of every $\diamond\mathcal{S}_i$ is a set of suspected processes satisfying strong completeness and *eventual weak accuracy* (i.e., there is a time after which one correct process is never suspected).

Eventual leader ($\Omega$): the output of each failure detector module $\Omega_i$ is a single process $p_j$, that $p_i$ currently *trusts*, i.e., that $p_i$ considers to be correct ($\mathcal{R}_\Omega = \Pi$). For every failure pattern, there is a time after which all correct processes always trust the same correct process. Obviously, $\Omega$ provides at least as much information as $\diamond\mathcal{S}$: if every process $p_i$ always suspects $\Pi - \{\Omega_i\}$, the properties of $\diamond\mathcal{S}$ are guaranteed [1].

We consider also the anonymously perfect failure detector $?\mathcal{P}$ [8], such that each module of $?\mathcal{P}$ at a given process outputs either 0 or 1 ($\mathcal{R}_{?\mathcal{P}} = \{0, 1\}$). When the failure detector module of $?\mathcal{P}$ at a process $p_i$ outputs 1, we say that $p_i$ *detects a crash*. $?\mathcal{P}$ satisfies the following properties: *anonymous completeness* (i.e., if some process crashes, then there is a time after which every correct process permanently detects a crash), and *anonymous accuracy* (i.e., no crash is detected unless some process crashes).

For any failure pattern $F$, $\mathcal{P}(F)$, $\diamond\mathcal{S}(F)$, $\Omega(F)$ and $?\mathcal{P}(F)$ denote the sets of *all* histories satisfying the corresponding properties. Recalling the notion of failure detector *classes* introduced in [2], every failure detector above denotes here the weakest element in the classes of failure detectors satisfying the corresponding properties.

[2] In [1], failure detectors can output values from an arbitrary range. In determining the weakest failure detector for NBAC, we also do not make any assumption a priori on the range of a failure detector.

**Algorithms.** We model the set of asynchronous communication channels as a message buffer which contains messages not yet received by their destinations. An algorithm $A$ is a collection of $n$ (possibly infinite state) deterministic automata, one for each of the processes. $A(p_i)$ denotes the automaton running on process $p_i$. In each step of $A$, process $p_i$ performs atomically the following three actions: (receive phase) $p_i$ chooses *non-deterministically* a single message addressed to $p_i$ from the message buffer, or a null message, denoted $\lambda$; (failure detector query phase) $p_i$ queries and receives a value from its failure detector module; (local state update phase) $p_i$ changes its state; and (send phase) sends a message to all processes according to the automaton $A(p_i)$, based on its state at the beginning of the step, the message received in the receive action, and the value obtained by $p_i$ from its failure detector module.[3]

**Configurations, schedules and runs.** A *configuration* defines the current state of each process in the system and the set of messages currently in the message buffer. Initially, the message buffer is empty. A step $(p_i, m, d, A)$ of an algorithm $A$ is uniquely determined by the identity of the process $p_i$ that takes the step, the message $m$ received by $p_i$ during the step ($m$ might be the null message $\lambda$), and the failure detector value $d$ seen by $p_i$ during the step. We say that a step $e = (p_i, m, d, A)$ *is applicable* to the current configuration if and only if $m = \lambda$ or $m$ is a message from the current message buffer destined to $p_i$. $e(C)$ denotes the unique configuration that results when $e$ is applied to $C$. A *schedule* $S$ of algorithm $A$ is a (finite or infinite) sequence of steps of $A$. $S_\perp$ denotes the empty schedule. We say that a schedule $S$ *is applicable to a configuration* $C$ if and only if (a) $S = S_\perp$, or (b) $S[1]$ is applicable to $C$, $S[2]$ is applicable to $S[1](C)$, etc. For a finite schedule $S$ applicable to $C$, $S(C)$ denotes the unique configuration that results from applying $S$ to $C$.

A *partial run of algorithm $A$ in an environment $\mathcal{E}$ using a failure detector $\mathcal{D}$* is a tuple $R = \langle F, H_\mathcal{D}, I, S, T\rangle$, where $F \in \mathcal{E}$ is a failure pattern, $H_\mathcal{D} \in \mathcal{D}(F)$ is a failure detector history, $I$ is an initial configuration of $A$, $S$ is a *finite* schedule of $A$, and $T \subset \mathcal{T}$ is a *finite* list of increasing time values, such that $|S| = |T|$, $S$ is applicable to $I$, and for all $t \leq |S|$, if $S[t]$ is of the form $(p_i, m, d, A)$ then: (1) $p_i$ has not crashed by time $T[t]$, i.e., $p_i \notin F(T[t])$ and (2) $d$ is the value of the failure detector module of $p_i$ at time $T[t]$, i.e., $d = H_\mathcal{D}(p_i, T[t])$.

A *run of algorithm $A$ in an environment $\mathcal{E}$ using a failure detector $\mathcal{D}$* is a tuple $R = \langle F, H_\mathcal{D}, I, S, T\rangle$, where $S$ is an *infinite* schedule of $A$ and $T \subseteq \mathcal{T}$ is an *infinite* list of increasing time values indicating when each step of $S$ occurred. In addition to satisfying the properties (1) and (2) of a partial run, run $R$ should guarantee that (3) every correct process in $F$ takes an infinite number of steps in $S$ and eventually receives every message sent to it (this conveys the reliability of the communication channels).

**Weakest failure detector.** A *problem* (e.g., NBAC or Consensus) is a set of runs (usually defined by a set of properties that these runs should satisfy). We say that a failure detector $\mathcal{D}$ *solves a problem $M$ in an environment $\mathcal{E}$* if there is an algorithm $A$, such that all the runs of $A$ in $\mathcal{E}$ using $\mathcal{D}$ are in $M$ (i.e., they satisfy the properties of $M$).

Let $\mathcal{D}$ and $\mathcal{D}'$ be any two failure detectors and $\mathcal{E}$ be any environment. If there is an algorithm $T_{\mathcal{D}'\rightarrow\mathcal{D}}$ that emulates $\mathcal{D}$ with $\mathcal{D}'$ in $\mathcal{E}$ ($T_{\mathcal{D}'\rightarrow\mathcal{D}}$ is called a *reduction* algorithm), we say that $\mathcal{D}$ *is weaker than $\mathcal{D}'$ in $\mathcal{E}$*, or $\mathcal{D} \preceq_\mathcal{E} \mathcal{D}'$. If $\mathcal{D} \preceq_\mathcal{E} \mathcal{D}'$ but $\mathcal{D}' \not\preceq_\mathcal{E} \mathcal{D}$ we say that $\mathcal{D}$ *is strictly weaker than $\mathcal{D}'$ in $\mathcal{E}$*, or $\mathcal{D} \prec_\mathcal{E} \mathcal{D}'$.[4] Note that $T_{\mathcal{D}'\rightarrow\mathcal{D}}$ does not need to emulate *all* histories of $\mathcal{D}$; it is required that all the failure detector histories it emulates be histories of $\mathcal{D}$.

We say that a failure detector $\mathcal{D}$ is *the weakest failure detector to solve a problem $M$ in an*

[3] The necessary part of our result also applies to weaker models where a step can atomically comprise at most one phase and where a process can atomically send at most one message to a single process per step. The sufficient part also holds in these models, since the algorithm of Section 3 is still correct there.

[4] Later we omit $\mathcal{E}$ in $\prec_\mathcal{E}$ and $\preceq_\mathcal{E}$ when there is no ambiguity on the environment $\mathcal{E}$.

*environment* $\mathcal{E}$ if two conditions are satisfied: (1) Sufficiency: $\mathcal{D}$ solves $M$ in $\mathcal{E}$, and (2) Necessity: if a failure detector $\mathcal{D}'$ solves $M$ in $\mathcal{E}$ then $\mathcal{D} \preceq_{\mathcal{E}} \mathcal{D}'$.

# 3 The candidate

Informally, our "XOR" failure detector $\mathcal{X}$ works as follows. For any failure pattern $F$, $\mathcal{X}$ eventually either outputs value 1 at every process or provides a history $H'$ of the failure detector $?\mathcal{P} + \diamond\mathcal{S}$, $H' \in (?\mathcal{P}+\diamond\mathcal{S})(F)$. When $\mathcal{X}$ eventually outputs 1 we say that $\mathcal{X}$ *detects a crash.* When $\mathcal{X}$ eventually provides $H' \in (?\mathcal{P} + \diamond\mathcal{S})(F)$, we say that $\mathcal{X}$ *misses a crash.* $\mathcal{X}$ guarantees that no crash is detected unless some process crashes. However, $\mathcal{X}$ does not guarantee that every crash failure is detected. If $\mathcal{X}$ does not detect a crash, then $\mathcal{X}$ behaves like $?\mathcal{P} + \diamond\mathcal{S}$.

Formally, $\mathcal{R}_{\mathcal{X}} = \{\perp\} \cup \{1\} \cup \mathcal{R}_{?\mathcal{P}+\diamond\mathcal{S}}$. Let $1_{\mathcal{X}} : \Pi \times \mathcal{T} \rightarrow \{1\}$, be a function, such that $\forall p_i \in \Pi$, $\forall t \in \mathcal{T}$: $1_{\mathcal{X}}(p_i, t) = 1$. $\forall F \in \mathcal{E}$ and $\forall H \in \mathcal{X}(F)$, there exist a function $H' \in \{1_{\mathcal{X}}\} \cup (?\mathcal{P} + \diamond\mathcal{S})(F)$ and a tuple $t_i \in \mathcal{T}, i = 1..n$, such that $\forall p_i \in \Pi$ and $t \in T$:

$$H(p_i, t) = \begin{cases} \perp & \text{if} \quad t < t_i \\ H'(p_i, t) & \text{if} \quad t \geq t_i \end{cases}$$

If $H' = 1_{\mathcal{X}}$, we say that $\mathcal{X}$ *detects a crash.* Otherwise, we say that $\mathcal{X}$ *misses a crash.* $\mathcal{X}$ guarantees the following accuracy property:

***$\mathcal{X}$-accuracy*** No crash is detected in a failure-free failure pattern: $H' = 1_{\mathcal{X}} \Rightarrow F \neq F_0$.

**Lemma 1** *$\mathcal{X}$ solves NBAC in any $\mathcal{E} \in \mathbb{E}_f$ with $0 \leq f < n/2$.*

**Proof:** An algorithm that solves NBAC using $\mathcal{X}$ is presented in Figure 1. Each process waits until its failure detector module outputs a value $x \neq \perp$. If 1 is output ($\mathcal{X}$ detects a crash), every process decides 0. Otherwise ($\mathcal{X}$ misses a crash and, thus, behaves like $?\mathcal{P}+\diamond\mathcal{S}$), the process runs the NBAC algorithm $\mathsf{NBAC}_{?\mathcal{P}+\diamond\mathcal{S}}$ [8] using $\mathcal{X}$ instead of $?\mathcal{P} + \diamond\mathcal{S}$.

```
1: function nbac(v_i)
2:    wait until X_i outputs x ≠ ⊥        { Wait until X detects a crash }
                                                  { or misses it }
3:    if x = 1 then
4:       return(0)                          { Abort if X detects a crash }
5:    else
6:       v_i ← nbac_{?P+◇S}(v_i)   { Otherwise, run the NBAC_{?P+◇S} algorithm }
7:       return(v_i)
```

Figure 1: An NBAC algorithm using $\mathcal{X}$ for process $p_i$.

To prove the correctness of the algorithm, we consider two cases:

1. $\mathcal{X}$ detects a crash: every process either decides 0 or crashes. Thus, *termination*, *agreement* and *a-validity* are ensured. If $\mathcal{X}$ detects a crash, then some process crashes. Thus, *c-validity* is ensured.

2. $\mathcal{X}$ misses a crash: the correctness follows from the correctness of the NBAC algorithm using $?\mathcal{P} + \diamond\mathcal{S}$ when a majority of processes is correct [8].

In both cases, $\mathcal{X}$ provides sufficient information about failures to solve NBAC. □

# 4 Proof technique

In this section, we slightly generalize the technique used in [1] in order to show the necessity part of our result.

**DAGs.** Let $\mathcal{E}$ be any environment. Let $F$ be any failure pattern in $\mathcal{E}$. Let $\mathsf{NBAC}_{\mathcal{D}}$ be any NBAC algorithm using a failure detector $\mathcal{D}$, $H$ be any history in $\mathcal{D}(F)$, and $I_1$ be any initial configuration of $\mathsf{NBAC}_{\mathcal{D}}$ in which every process proposes 1.

Let $G$ be an infinite directed acyclic graph (DAG) defined by the set of vertices $V(G)$ and a set of directed edges $E(G)$ of the form $v \rightarrow v'$, where $v \in V(G)$ and $v' \in V(G)$, with the following properties:

(1) The vertices of $G$ are of the form $[p_i, d, k]$ where $p_i \in \Pi$, $d \in \mathcal{R}_{\mathcal{D}}$ and $k \in \mathbb{N}$. There is a mapping $f : V(G) \rightarrow \mathcal{T}$ that associates a time with each vertex of $G$, such that:

  (a) For any $v = [p_i, d, k] \in V(G)$, $p_i \notin F(f(v))$ and $d = H(p_i, f(v))$.

  (b) For any edge $v \rightarrow v' \in E(G)$, $f(v) < f(v')$.

(2) If $[p_i, d, k] \in V(G)$,$[p_i, d', k'] \in V(G)$ and $k < k'$ then $[p_i, d, k] \rightarrow [p_i, d', k'] \in E(G)$.

(3) $G$ is transitively closed.

(4) Let $U \subseteq V(G)$ be a finite set of vertices and $p_i$ be any correct process in $F$. There is $d \in \mathcal{R}_{\mathcal{D}}$ and $k \in \mathbb{N}$, such that for every vertex $v \in V(G)$, $v \rightarrow [p_i, d, k]$ is an edge of $G$.

**Simulation trees.** Let $g = [q_1, d_1, k_1], [q_2, d_2, k_2], ...$ be a path of $G$. A schedule $S$ is *compatible with* $g$ if $|S| = |g|$ and $S = (q_1, m_1, d_1), (q_2, m_2, d_2), ...$, for some messages $m_1, m_2, ...$ . We say that $S$ is *compatible with* $G$ if $S$ is compatible with some path in $G$.

The set of all schedules of $\mathsf{NBAC}_{\mathcal{D}}$ that are compatible with $G$ and applicable to $I_1$ are modeled as a tree $\Upsilon$, called the *simulation tree induced by* $G$, defined as follows. The set of vertices of $\Upsilon$ is the set of *finite* schedules $S$ that are compatible with $G$ and applicable to $I_1$. The root of $\Upsilon$ is an empty schedule $S_{bot}$. There is an edge from vertex $S$ to a vertex $S'$ if and only if $S' = S \cdot e$ for a step $e$. Thus, for each (finite or infinite) path of $\Upsilon$, there is a unique schedule $S = e_1, e_2, ....$ Every vertex $S$ of $\Upsilon$ corresponds to a partial run $\langle F, H, I_1, S, T \rangle$ of $\mathsf{NBAC}_{\mathcal{D}}$.

It can also be easily shown that for any path in $\Upsilon$ associated with an infinite schedule $S$, in which every correct process takes an infinite number of steps and receives every message addressed to it, there is a sequence of times $T$, such that $\langle F, H, I_1, S, T \rangle$ is a run of $\mathsf{NBAC}_{\mathcal{D}}$ (Lemma 6.2.2 of [1]). Moreover, each vertex $S$ in $\Upsilon$ has a descendant $S'$ in $\Upsilon$, such that all correct processes have decided in $S'(I_1)$ (Lemma 6.2.6 of [1]). Thus, we can assign a non-empty set of *tags* to each vertex of $\Upsilon$. Vertex $S$ gets a tag $k \in \{0, 1\}$ if and only if it has a descendant $S'$, such that some correct process has decided $k$ in $S'(I_1)$. A vertex of $\Upsilon$ is called *$k$-valent* if it has only one tag $k$. Otherwise, it is called *bivalent*. A 0-valent or 1-valent vertex is called *monovalent*. We say that the simulation tree $\Upsilon$ is bivalent if its root $S_\perp$ is bivalent. Respectively, $\Upsilon$ is $k$-valent if $S_\perp$ is $k$-valent.

**Lemma 2** *$\Upsilon$ is 1-valent if and only if $F = F_0$ ($F_0$ is the failure-free pattern).*[5]

**Proof:** Assume that the root of $\Upsilon$ is 1-valent. Assume, by contradiction, that $F \neq F_0$. Let $p_i$ be a process in $\Pi - correct(F)$ and $I_0$ be an initial configuration of $\mathsf{NBAC}_{\mathcal{D}}$ in which $p_i$ proposes 0 and all other processes propose 1. By Lemma 6.2.6 of [1], there is a finite schedule $S$ in $\Upsilon$ containing steps

[5]Note that the statement is not true for the consensus problem: by the consensus validity, $\Upsilon$ for consensus is always 1-valent independently of the failure pattern.

of correct processes only, such that some correct process $p_j$ has decided in $S(I_1)$. By the assumption, the decision value is 1. Let $S$ corresponds to a partial run $R = \langle F, H, I_1, S, T\rangle$. Since $p_i$ does not take a single step in $S$, $S$ also corresponds to the run $R' = \langle F, H, I_0, S, T\rangle$ in which $p_i$ proposes 0. No correct process can distinguish $R$ and $R'$, thus $p_j$ decides 1 in $S(I_0)$, contradicting *a-validity* of NBAC.

Assume now, by contradiction, that $F = F_0$ and there is a schedule $S$ of $\Upsilon$, such that a process $p_j$ decides 0 in $S(I_1)$. By Lemma 6.2.2 of [1], there is a partial run $\langle F_0, H, I_1, S, T\rangle$ of $\mathsf{NBAC}_{\mathcal{D}}$. In $R$, $p_j$ decides 0 even though every process is correct and proposes 1, contradicting the *c-validity* property of NBAC. □

**Decision gadgets and deciding processes.** We also use the notion of *decision gadgets* introduced in [1]. A decision gadget is a finite subtree of $\Upsilon$ rooted at $S_\perp$. Any decision gadget has exactly two leaves: one 0-valent and one 1-valent. Any decision gadget has a *deciding process* that defines the valence of each descendant of the gadget in $\Upsilon$. (The formal definitions of decision gadgets and deciding processes are not central for this paper and can be found in [1].) We also make use of the following lemmas (Lemma 6.4.1 and Lemma 6.5.3 of [1]).

**Lemma 3** *If $\Upsilon$ is bivalent, then $\Upsilon$ has at least one decision gadget (and hence a deciding process).*

**Lemma 4** *The deciding process of a decision gadget is correct.*

Thus, if $\Upsilon$ is bivalent, it is possible to locate in $\Upsilon$ a decision gadget and compute a correct process out of it. This provides a way to design a distributed reduction algorithm $T_{\mathcal{D}\to\Omega}$ that, for every failure pattern $F$ and history $H \in \mathcal{D}(F)$, for which a bivalent simulation tree can be constructed, eventually outputs at every correct process the identifier of the same correct process $p_c$. This is sufficient to emulate $\Omega$ and, thus, $\diamond\mathcal{S}$ [1]. Even though the simulation tree $\Upsilon$ is infinite and cannot be computed in a finite time, there exists a *finite subtree* of $\Upsilon$ that gives sufficient information to identify $p_c$.

# 5 Necessary condition: the reduction algorithm

The algorithm $\mathsf{NBAC}_{\mathcal{D}}$ provides a single operation $\mathsf{nbac}()$, that takes as a parameter a value in $\{0, 1\}$ and returns a value in $\{0, 1\}$. We present the algorithm $T_{\mathcal{D}\to\mathcal{X}}$, any run of which emulates an output of $\mathcal{X}$. First, we describe some general properties of the algorithm.

Each process $p_i$ maintains a local variable $G_i$, a finite ever growing DAG, that contains a sample of failure detector output at the processes in the system as well as temporal relationships between them. $G_i$ is actually a subgraph of some infinite DAG $G$ discussed in Section 4. The variable $G_i$ is attached to every message sent by $p_i$. Let $k$ be any natural number and $(p_i, m, d)$ be the $k$-th step of process $p_i$ in any run of the algorithm $T_{\mathcal{D}\to\mathcal{X}}$. At the beginning of the local state update phase, $p_i$ performs the procedure $\mathsf{update}_i(m, d, k)$ presented in Figure 2.

Note that the resulting algorithm conforms to our model, where every step atomically comprises a receive phase, a failure detector query, an update of the local state and a send phase.

Let $G_i(t)$ denote the value of $G_i$ at the end of the last step of $p_i$ before time $t \in \mathcal{T}$. There exists a DAG $G$ and a map $f : V(G) \to T$ satisfying the properties (1)-(4), such that for any correct process $p_i$, $\lim_{t\to\infty} G_i(t) = G$ (Lemma 6.6.1.2 and Lemma 6.6.1.4 of [1]).

$\Upsilon_i$ denotes the ever growing finite simulation tree induced by its finite subgraph $G_i$. Clearly, $\lim_{t\to\infty} \Upsilon_i(t) = \Upsilon$. We tag $\Upsilon_i$ by adding a tag $k$ to any $S$ in $\Upsilon_i$ if and only if $S$ has a descendant $S'$ in $\Upsilon_i$, such that $p_i$ has decided $k$ in $S'(I_1)$. (If $p_i$ is correct, then the tagging scheme is equivalent to the one used to tag $\Upsilon$ in Section 4; if $p_i$ is not correct, then the tagging scheme used by $p_i$ does not matter for emulating $\mathcal{X}$.)

```
1: procedure update_i(m, d, k)
2:   if m ≠ ⊥ is a message from a process p_j then
3:     G_j ← DAG extracted from m
4:     G_i ← G_i ∪ G_j                { Merge vertices and edges of G_i and G_j }
5:   V(G_i) ← V(G_i) ∪ [p_i, d, k]    { Add to G_i the current value seen by p_i }
6:   for all v ∈ V(G_i), v ≠ [p_i, d, k] do
7:     E(G_i) ← E(G_i) ∪ (v → [p_i, d, k])   { Add to G_i an edge to [p_i, d, k] }
```

Figure 2: The $\mathsf{update}_i(m, d, k)$ procedure.

Further, we modify the $\mathsf{NBAC}_{\mathcal{D}}$ algorithm by (a) adding the $\mathsf{update}_i(m)$ procedure to the end of the local state update phase of every step and (b) attaching the value of $G_i$ to every message sent by the algorithm. As a result, we obtain an algorithm $\mathsf{NBAC}'_{\mathcal{D}}$ that, in addition to solving NBAC, computes at every process $p_i$ some finite DAG $G_i$. Denote by $\mathsf{nbac}'_{\mathcal{D}}()$ the operation provided by $\mathsf{NBAC}'_{\mathcal{D}}$.

The reduction algorithm $T_{\mathcal{D}\to\mathcal{X}}$ is presented in Figure 3. In the algorithm, every process $p_i$ maintains a variable $output_i$ the value of which is returned each time the failure detector module $\mathcal{X}_i$ is queried.

```
1:  output_i ← ⊥
2:  G_i ← ∅
3:  k ← 1
4:  res ← nbac'_D(1)                  { Run the NBAC'_D algorithm maintaining G_i }
5:  if res = 0 then
6:    output_i ← 1
7:  else
8:    while true do
9:      p_i receives message m                                  { Receive phase }
10:     d ← D_i                                  { Failure detector query phase }
11:     update_i(m, d, k)                          { Local state update phase }
12:     k ← k + 1
13:     add [p_i, d, k] to G_i and edges from all other vertices of G_i to [p_i, d, k]
14:     Υ_i ← simulation tree induced by G_i
15:     if Υ_i has no decision gadgets then
16:       output_i ← (0, Π − {p_1})
17:     else
18:       p_c ← deciding process of the smallest decision gadget of Υ_i
19:       output_i ← (1, Π − {p_c})
20:     send (p_i, G_i) to all p_j ∈ Π                                 { Send phase }
```

Figure 3: Reduction algorithm $T_{\mathcal{D}\to\mathcal{X}}$ for process $p_i$.

**Lemma 5** *For any environment $\mathcal{E}$, if a failure detector $\mathcal{D}$ solves NBAC in $\mathcal{E}$, then $\mathcal{X} \preceq_{\mathcal{E}} \mathcal{D}$.*

**Proof:** We show that the algorithm of Figure 3 emulates the output of $\mathcal{X}$ in any environment. Consider any run $R = \langle F, H, I, S, T\rangle$ of the algorithm. The following cases are possible:

**I.** $\mathsf{nbac}'_{\mathcal{D}}(1)$ returns 0 (line 4 of Figure 3). The *agreement* property of NBAC implies that every process $p_i$ either crashes or sets $output_i$ to 1. By *c-validity*, $F \neq F_0$, thus, the algorithm emulates a

valid history of $\mathcal{X}$ detecting a crash.

**II.** $\mathsf{nbac}'_{\mathcal{D}}(1)$ returns 1 (line 4 of Figure 3). Our construction of the $\mathsf{NBAC}'_{\mathcal{D}}$ algorithm implies that, for every process $p_i$ that reached line 5 of Figure 3, $\Upsilon_i$ includes at least one 1-valent vertex associated with the schedule of $\mathsf{NBAC}'_{\mathcal{D}}$ in which 1 is decided.

If $F = F_0$, then, by *c-validity*, the only tag that a vertex of every $\Upsilon_i$ can get is 1. Thus, every correct $p_i$ eventually permanently sets $output_i$ to $(0, \Pi - \{p_1\})$. No crash is ever detected in $F_0$ by the emulated module of $?\mathcal{P}$: anonymous completeness and anonymous accuracy of $?\mathcal{P}$ are ensured. Further, $p_1$ is correct in $F_0$. Thus, the algorithm emulates a valid history in $\mathcal{X}(F_0)$.

Let $F \neq F_0$. By Lemma 2, $\Upsilon$ is not 1-valent. It is not 0-valent either, because $\Upsilon_i \subset \Upsilon$ and $\Upsilon_i$ includes a 1-valent vertex. Thus, $\Upsilon$ is bivalent and, by Lemma 3, has at least one bivalent gadget.

Let $\varepsilon$ be the first decision gadget in $\Upsilon$ and $p_c$ be the deciding process of $\varepsilon$ (since the vertexes in $\Upsilon$ are countable, we can define a rule to compute the first decision gadget). Eventually, $\Upsilon_i$ of every correct process $p_i$ includes $\varepsilon$ ($\varepsilon$ is a finite subtree of $\Upsilon$). Thus, there is a time after which $p_i$ computes $p_c$ as the deciding process of $\varepsilon$, the first decision gadget of $\Upsilon_i$ (line 18 of of Figure 3). As a result, every correct $p_i$ eventually permanently sets $output_i$ to $(1, \Pi - \{p_c\})$. Since $F \neq F_0$ and the emulated module of $?\mathcal{P}$ detects a crash, the anonymous completeness and anonymous accuracy properties of $?\mathcal{P}$ are ensured. By Lemma 4, $p_c$ is correct. Thus, the algorithm emulates a valid history in $\mathcal{X}(F)$. □

Combining Lemma 1 and Lemma 5 together we obtain our main result:

**Theorem 6** *$\mathcal{X}$ is the weakest failure detector to solve NBAC in any $\mathcal{E} \in \mathbb{E}_f$ with $0 \leq f < n/2$.*

# References


[1] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM (JACM)*, 43(4):685–722, July 1996.

[2] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2):225–267, March 1996.

[3] B. A. Coan and J. L. Welch. Transaction commit in a realistic timing model. *Distributed Computing*, 4(2):87–103, 1990.

[4] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(3):374–382, April 1985.

[5] E. Fromentin, M. Raynal, and F. Tronel. On classes of problems in asynchronous distributed systems with process crashes. In *Proceedings of the IEEE International Conference on Distributed Systems (ICDCS)*, pages 470–477, 1999.

[6] J. Gray. A comparison of the byzantine agreement problem and the transaction commit problem. In *Proceedings of the Workshop on Fault-Tolerant Distributed Computing*, volume 448 of *LNCS*, pages 10–17. Springer-Verlag, 1986.

[7] R. Guerraoui. Revisiting the relationship between non-blocking atomic commit and consensus. In *Proceedings of the 9th International Workshop on Distributed Algorithms (WDAG)*, volume 972 of *LNCS*, pages 87–100. Springer Verlag, 1995.

[8] R. Guerraoui. Non-blocking atomic commit in asynchronous distributed systems with failure detectors. *Distributed Computing*, 15:17–25, January 2002.

[9] R. Guerraoui and P. Kouznetsov. On the weakest failure detector for non-blocking atomic commit. In *Proceedings of the 2nd IFIP International Conference on Theoretical Computer Science (TCS 2002)*, pages 461–473, August 2002.

[10] V. Hadzilacos. On the relationship between the atomic commitment and consensus problems. In *Proceedings of the Workshop on Fault-Tolerant Distributed Computing*, volume 448 of *LNCS*, pages 201–208. Springer-Verlag, 1986.

[11] V. Hadzilacos and S. Toueg. The weakest failure detector to solve quittable consensus and non-blocking atomic commit. Unpublished manuscript – private communication, May 2003.

[12] D. Skeen. NonBlocking commit protocols. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 133–142. ACM Press, May 1981.