The Weakest Failure Detectors to Solve Certain Fundamental Problems in Distributed Computing*

[Extended Abstract]†

Carole Delporte-Gallet
LIAFA,
Université Paris 7
cd@liafa.jussieu.fr

Hugues Fauconnier
LIAFA,
Université Paris 7
hf@liafa.jussieu.fr

Rachid Guerraoui
Distributed Programming Laboratory, EPFL
rachid.guerraoui@epfl.ch

Vassos Hadzilacos
Department of Computer Science, University of Toronto
vassos@cs.toronto.edu

Petr Kouznetsov
Distributed Programming Laboratory, EPFL
petr.kouznetsov@epfl.ch

Sam Toueg
Department of Computer Science, University of Toronto
sam@cs.toronto.edu

ABSTRACT

We determine the weakest failure detectors to solve several fundamental problems in distributed message-passing systems, for *all* environments — i.e., regardless of the number and timing of crashes. The problems that we consider are: implementing an atomic register, solving consensus, solving quittable consensus (a variant of consensus in which processes have the option to decide 'quit' if a failure occurs), and solving non-blocking atomic commit.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed applications, distributed databases, network operating systems*; D.4.5 [**Operating Systems**]: Reliability—*fault tolerance*; F.1.1 [**Computation by Abstract Devices**]: Models of Computation—*automata, relations among models*; H.2.4 [**Database Management**]: Systems—*concurrency, distributed databases, transaction processing*

General Terms

Algorithms, Reliability, Theory

Keywords

register, consensus, quittable consensus, non-blocking atomic commit, weakest failure detector

*This work was partially supported by the Swiss National Science Foundation (OFES FN - Projet 2100-66768.01/1).

†This paper combines results of two technical reports [7] and [12].

PODC'04, July 25–28, 2004, St. Johns, Newfoundland, Canada.

1. INTRODUCTION

Consensus is a classical problem in fault tolerant distributed computing. Informally, each process in the system initially proposes a value, and eventually processes must reach a common decision on one of the proposed values. Non-blocking atomic commit (NBAC) is a closely related problem that arises in distributed transaction processing [10]. Informally, the set of processes that participate in a transaction must agree on whether to commit or abort that transaction. Initially each process votes Yes ("I am willing to commit") or No ("we must abort"), and eventually processes must reach a common decision, Commit or Abort. The decision to Commit can be reached only if all processes voted Yes. Furthermore, if all processes voted Yes and no failure occurs, then the decision *must* be Commit.

It is well-known that consensus and NBAC are unsolvable in asynchronous systems with process crashes (even if communication is reliable) [8]. One way to circumvent such impossibility results is through the use of *unreliable failure detectors* [4]: In this model, each process has access to a failure detector module that provides some (possibly incomplete and inaccurate) information about failures, e.g., a list of processes currently suspected to have crashed. Chandra *et al.* [3] determined the *weakest* failure detector to solve consensus in sytems with a majority of correct processes.[1] It was not known, however, whether there is a weakest failure detector to solve consensus regardless of the number of faulty processes — and, if so, what that failure detector is.

One of the results of this paper is to determine the weakest failure detector to solve consensus in *any* environment whatsoever. (We will formally define the term "environment" in Section 2. Informally, however, an environment encapsulates an arbitrary assumption about which processes crash and when they do. Examples of environments are: a majority of the processes are correct; process p never fails before process q; no process crashes after it has taken at least one step.) To obtain this result, we exploit a close relationship

[1] We say that $\mathcal{D}$ is the weakest failure detector to solve problem P if (sufficient part) there is an algorithm that uses $\mathcal{D}$ to solve P, and (necessary part) any failure detector $\mathcal{D}'$ that can be used to solve P can be transformed to $\mathcal{D}$.

between consensus and atomic registers. We determine the weakest failure detector to implement such registers in any environment — a previously unexplored question of independent interest. We then leverage this result to determine the weakest failure detector to solve *consensus* in any environment.

As with consensus, failure detectors can be used to solve NBAC [9, 11]. It was an open problem, however, whether there is a weakest failure detector to solve NBAC and, if so, what that failure detector is. The answer to this question was not known even for environments with a majority of correct processes. In this paper we resolve this problem, again for any environment. To do so, we first introduce a natural variation of consensus, called *quittable consensus* (QC) — a problem that is interesting in its own right. We then determine the weakest failure detector to solve QC in all environments, and establish a close relationship between QC and NBAC. Using this, we determine the weakest failure detector to solve NBAC in any environment.

We now describe in more detail our results; these involve the following three failure detectors.

- The *quorum failure detector* Σ outputs a set of processes at each process. Any two sets (output at any times and by any processes) intersect, and eventually every set output at correct processes consists only of correct processes [7].
- The *leader failure detector* Ω outputs the id of a process at each process. There is a time after which it outputs the id of the same correct process at all correct processes [3].
- The *failure signal* failure detector $\mathcal{FS}$ outputs **green** or **red** at each process. As long as there are no failures, $\mathcal{FS}$ outputs **green** at every process; after a failure occurs, and only if it does, $\mathcal{FS}$ must eventually output **red** permanently at every correct process [5, 11].

The weakest failure detector to implement an atomic register. It is well-known that in asynchronous message-passing systems it is possible to implement atomic registers if and only if a majority of processes are correct [1]. The question naturally arises whether using some sort of failure detectors it is possible to implement atomic registers even in environments where fewer than a majority of processes can be relied upon to be correct. We prove that the quorum failure detector Σ is the weakest one to do so in *any* environment.

One may wonder how this can be the case, since in environments with a majority of correct processes atomic registers can be implemented without *any* failure detector. The answer is that, in such environments, we can easily implement Σ *ex nihilo* as follows: Each process periodically sends "join-quorum" messages, and takes as its present quorum any majority of processes that respond to that message. Thus, to implement registers in environments with a majority of correct processes we "need" something that we can get for free!

Like any weakest failure detector result, this one has two aspects: We must prove that in any environment,

(1) using Σ we can implement registers; and

(2) any failure detector that can be used to implement registers can be transformed to Σ.

(1) will not surprise anyone familiar with the algorithm of Attiya *et al.* for implementing registers in a message-passing system [1]: Where that algorithm uses majorities to ensure that a read operation returns the most recently written value, we can use the quorums provided by Σ to the same effect. (2) is far less obvious; it is discussed in Section 3.

The weakest failure detector to solve consensus. It is known that the leader failure detector Ω is the weakest failure detector to solve consensus in environments with a majority of correct processes. More specifically,

(3) using Ω we can solve consensus if a majority of processes are correct [4]; and

(4) any failure detector that can be used to solve consensus can be transformed to Ω [3].

It is important to note that (4) holds for *all* environments.

In this paper we show that the failure detector (Ω, Σ) is the weakest failure detector to solve consensus in all environments.[2] This result is obtained in the following way. It is known that using registers and Ω we can solve consensus in *any* environment [19]. By (1), it follows that using (Ω, Σ) we can solve consensus in any environment. Conversely, let $\mathcal{D}$ be a failure detector that can be used to solve consensus in an arbitrary environment $\mathcal{E}$. From Lamport's work on the state-machine approach we know that by using consensus we can implement any object, and in particular registers [17, 21]. Thus, using $\mathcal{D}$ we can implement registers in $\mathcal{E}$. By (2), $\mathcal{D}$ can be transformed to Σ in $\mathcal{E}$. Since, by (4), $\mathcal{D}$ can also be transformed to Ω in $\mathcal{E}$, we conclude that $\mathcal{D}$ can be transformed to (Ω, Σ) in $\mathcal{E}$.

The fact that (Ω, Σ) is the weakest failure detector to solve consensus in any environment, implies that Ω is the weakest failure detector to solve consensus when a majority of processes are correct, since as we noted earlier, in such environments Σ can be implemented *ex nihilo*. Thus the result we show in this paper is a natural generalisation of the previously known result about the weakest failure detector to solve consensus in environments with a majority of correct processes [3, 4].

The weakest failure detectors to solve QC and NBAC. Informally, QC is like consensus except that, in case a failure occurs, processes have the option (but not the obligation) to agree on a special value Q (for "quit"). This weakening of consensus is appropriate for applications where, when a failure occurs, processes are allowed to agree on that fact (rather than on an input value) and resort to a default action.

Despite their apparent similarity, QC and NBAC are different in important ways. In NBAC the two possible input values Yes and No are not symmetric: A single vote of No is enough to force the decision to abort. In contrast, in QC (as in consensus) no input value has a privileged role. Another way in which the two problems differ is that the semantics of the decision to abort (in NBAC) and the decision to quit (in QC) are different. In NBAC the decision to abort is sometimes inevitable (e.g., if a process crashes before voting); in contrast, in QC the decision to quit is never inevitable, it is only an option. Moreover, in NBAC the decision to abort signifies that either a failure has occurred *or* someone voted No; in contrast, in QC the decision to quit is allowed only if a failure has occurred.

We first show that there is a weakest failure detector to solve QC. This failure detector, which we denote Ψ, behaves as follows: For an initial period of time the output of Ψ at each process is $\perp$. Eventually, however, Ψ either behaves like the failure detector (Ω, Σ) at all processes or, if a failure occurs, it may instead behave like the failure detector $\mathcal{FS}$ at all processes. The switch from $\perp$ to (Ω, Σ) or $\mathcal{FS}$ need not occur simultaneously at all processes, but the same choice is made by all processes. This result has an intuitively appealing interpretation: To solve QC, a failure detector

[2] If $\mathcal{D}$ and $\mathcal{D}'$ are failure detectors, $(\mathcal{D}, \mathcal{D}')$ is the failure detector that outputs a vector with two components, the first being the output of $\mathcal{D}$ and the second being the output of $\mathcal{D}'$.

must eventually either truthfully inform all processes that a failure has occurred, in which case the processes can decide Q, or it must be powerful enough to allow processes to solve consensus on their proposed values. This matches the behavior of Ψ.

We then prove that NBAC is equivalent to QC "modulo" the failure detector $\mathcal{FS}$. More precisely we show that: (a) given $\mathcal{FS}$, any QC algorithm can be transformed into an algorithm for NBAC, and (b) any algorithm for NBAC can be transformed into an algorithm for QC, and can also be used to implement $\mathcal{FS}$.

Finally, we use this equivalence to prove that $(\Psi, \mathcal{FS})$ is the weakest failure detector to solve NBAC. This result applies to any system, regardless of the number of faulty processes.

Related work. The question of the weakest failure detector for consensus was first addressed in [3], where it was shown that failure detector Ω is the weakest to solve consensus with a majority of correct processes. The generalization of the result to other environments was partially addressed in [6]. The same paper addressed the question of the weakest failure detector to implement atomic registers for specific environments and within a restricted class of failure detectors.

NBAC has been studied extensively in the context of transaction processing [10, 22]. Its relation to consensus was first explored in [14]. Charron-Bost and Toueg [5] and Guerraoui [11] showed that despite some apparent similarities, in asynchronous systems NBAC and consensus are in general incomparable — i.e., a solution for one problem cannot be used to solve the other.[3] The problem of determining the weakest failure detector to solve NBAC was explored and settled in special settings. Fromentin *et al.* [9] determine that to solve NBAC between *every* pair of processes in the system, one needs a *perfect failure detector* [4]. Guerraoui and Kouznetsov [13] determine the weakest failure detector for NBAC within a restricted class of failure detectors; while from results of [5] and [11] it follows that in the special case where at most one process may crash, $\mathcal{FS}$ is the weakest failure detector to solve NBAC. The general question, however, was open until the present paper.

Several versions of the consensus problem have been studied before but, to the best of our knowledge, this is the first paper to propose quittable consensus.

Roadmap. The rest of the paper is organised as follows: In Section 2 we briefly review the model of computation, and define formally the failure detectors Ω, Σ and $\mathcal{FS}$. In Section 3 we prove that Σ is the weakest failure detector to implement registers in any environment.In Section 4 we prove that (Ω, Σ) is the weakest failure detector to solve consensus in any environment. In Section 5 we give the exact specification of QC, and in Section 6 we prove that Ψ is the weakest failure detector to solve QC. In Section 7 we give the specification of NBAC, we prove that NBAC is equivalent to QC modulo $\mathcal{FS}$, and we use this to show that $(\Psi, \mathcal{FS})$ is the weakest failure detector to solve NBAC. The missing proofs can be found in [7, 12].

2. MODEL

We consider the asynchronous message-passing model in which processes have access to failure detectors [3, 4]. In this section we summarise the relevant terminology and notation.

The system consists of a set Π of n processes. Processes can fail only by crashing, i.e., prematurely halting. Each process executes steps asynchronously: the delays between steps are finite but unbounded and variable. Processes are connected via reliable links that transmit messages with finite but unbounded and variable delay.

A *failure pattern* is a function $F : \mathbb{N} \to 2^{\Pi}$, where $F(t)$ denotes the set of processes that have crashed through time t. (We assume a discrete global clock used only for presentational convenience, and not accessible by the processes.) Crashed processes do not recover and so, for all $t \in \mathbb{N}$, $F(t) \subseteq F(t+1)$. $\mathit{faulty}(F) = \bigcup_{t \in \mathbb{N}} F(t)$ denotes the set of processes that crash in F, and $\mathit{correct}(F) = \Pi - \mathit{faulty}(F)$ denotes the set of processes that are correct in F.

A *failure detector history with range* R describes the behavior of a failure detector during an execution; formally, it is a function $H : \Pi \times \mathbb{N} \to R$, where $H(p, t)$ is the value of the failure detector module of process p at time t. A *failure detector* $\mathcal{D}$ *with range* R is a function that maps each failure pattern F to a *set* of failure detector histories with range R. Intuitively, $\mathcal{D}(F)$ is the set of behaviors that $\mathcal{D}$ can exhibit when the failure pattern is F.

An algorithm $\mathcal{A}$ is an automaton that specifies, for each process p, (a) the set of messages that p can send; (b) the set of states that p can occupy (a subset of which are identified as p's possible initial states); and (c) a transition function which determines the messages that p sends and the new state it occupies when it takes a step. In one atomic step, p performs the following actions: it receives a message addressed to it (possibly the empty message λ), it queries the failure detector and receives its present value, it sends messages to other processes and changes its state.[4] The messages that p sends and the new state it occupies are specified by its transition function based on its present state, the message it receives and the value it was given by the failure detector. Formally, a step is a triple $\langle p, m, d \rangle$, where p is the process taking the step, m is the message p receives in that step, and d is the failure detector value it sees in that step. A *schedule* S is a finite or infinite sequence of steps.

A *configuration of algorithm* $\mathcal{A}$ specifies the global state of the system, i.e., the state of each process and of the "message buffer" which contains the set of messages that have been sent but have not yet been received. An *initial configuration of* $\mathcal{A}$ is a configuration in which every process occupies an initial state and the message buffer is empty. The step $e = \langle p, m, d \rangle$ is *applicable* to configuration C if m is λ or m belongs to the message buffer of C and its recipient is p; in this case $e(C)$ denotes the configuration that results if the present configuration is C, and process p executes the step in which it receives message m and sees failure detector value d. A schedule $S = e_1 e_2 \ldots$ is *applicable* to configuration C if and only if S is empty or e_1 is applicable to C, e_2 is applicable to $e_1(C)$ and so on. If S is a finite schedule applicable to C, $S(C)$ denotes the configuration that results from applying S to C.

A run of algorithm $\mathcal{A}$ using a failure detector $\mathcal{D}$ describes an execution of $\mathcal{A}$ where processes have access to $\mathcal{D}$. Formally, a *run* (respectively, *partial run*) of algorithm $\mathcal{A}$ using a failure detector $\mathcal{D}$ is a tuple $R = \langle F, H, I, S, T \rangle$ where F is a failure pattern, $H \in \mathcal{D}(F)$ is a failure detector history, I is an initial configuration of A, S is an infinite (respectively, finite) schedule of A that is applicable to I, and T is an infinite (respectively, finite) increasing list of times indicating when each step in S occurred. A number of straightforward conditions are imposed on the components of runs and partial runs to ensure that the failure detector values that appear in the steps of the schedule are consistent with the failure detection history H, that processes don't take steps after crashing, and that (in runs) correct processes take infinitely many steps and messages are not lost. For details see [3].

Some algorithms meet their specification only under some as-

[3] An exception is the case where at most one process may fail. In this case, NBAC can be transformed into consensus, but the reverse does not hold.

[4] Our result also applies to models where steps have finer granularity. For simplicity we focus on this one.

sumptions about the "environment" — e.g., that a majority of the processes are correct, or that two particular processes do not both crash. Formally, an *environment* $\mathcal{E}$ is a set of possible failure patterns. Intuitively, these are the failure patterns in which an algorithm of interest works correctly.

In Section 1 we informally introduced the failure detectors Ω, Σ, $\mathcal{FS}$. We now define these formally.

- The range of Ω is Π. For every failure pattern F,

$$H \in \Omega(F) \quad \Leftrightarrow \quad \exists p \in correct(F)\, \forall q \in correct(F)\, \exists t \in \mathbb{N}\, \forall t' \geq t\; H(q, t') = p.$$

- The range of Σ is 2^{Π}. For every failure pattern F,

$$H \in \Sigma(F) \quad \Leftrightarrow \quad \big(\forall p, p' \in \Pi\, \forall t, t' \in \mathbb{N}\; H(p,t) \cap H(p',t') \neq \emptyset\big) \wedge \big(\forall p \in correct(F)\, \exists t \in \mathbb{N}\, \forall t' \geq t\; H(p,t') \subseteq correct(F)\big).$$

- The range of $\mathcal{FS}$ is $\{\mathbf{green}, \mathbf{red}\}$. For every failure pattern F,

$$H \in \mathcal{FS}(F) \quad \Leftrightarrow \quad \forall p \in \Pi\, \forall t \in \mathbb{N}\, \big(H(p,t) = \mathbf{red} \Rightarrow F(t) \neq \emptyset\big) \wedge \big(faulty(F) \neq \emptyset \Rightarrow \forall p \in correct(F)\, \exists t \in \mathbb{N}\, \forall t' \geq t\; H(p,t') = \mathbf{red}\big).$$

3. THE WEAKEST FAILURE DETECTOR TO IMPLEMENT AN ATOMIC REGISTER

A register is a shared object accessed through two operations: *read* and *write*. The write operation takes as an input parameter a specific value to be stored in the register and returns a simple indication *ok* that the operation has been executed. The read operation returns the last value written in the register. The registers we consider are *fault-tolerant* and *atomic* [18] (*linearizable* [15]): they ensure that, despite concurrent invocations and possible crashes of the processes, every correct process that invokes an operation eventually gets a reply (a value for the read and an *ok* indication for the write), and any operation appears to be executed instantaneously between its invocation and reply time events. (A precise definition is given in [2, 15].)

Theorem 1 *For all environments $\mathcal{E}$, Σ is the weakest failure detector to implement an atomic register in $\mathcal{E}$.*

PROOF SKETCH. (A detailed proof is presented in [7].)

To prove the sufficiency of Σ, we adapt the algorithm of [1] to show how an atomic register with one reader and one writer can be implemented with Σ. Then, using the classical of results [16, 23], we deduce that atomic registers with multiple readers and writers can be implemented with Σ.

The proof that Σ is necessary to implement atomic registers is more intricate. We must show that any algorithm A that implements a register in some environment $\mathcal{E}$ using a failure detector $\mathcal{D}$ can be used to emulate the output of Σ. The transformation algorithm that does this is presented in Figure 1.

In the transformation algorithm, every process p_i maintains a variable Σ-$output_i$. We now prove that for any two processes p_k and p_l, Σ-$output_k$ and Σ-$output_l$ (taken at any times) intersect (*intersection*, and, at every correct process p_j, Σ-$output_j$ eventually consists only of correct processes (*completeness*).

The transformation algorithm uses n copies of the algorithm A that, using failure detector $\mathcal{D}$, implement n atomic registers, denoted by $Reg_1, \ldots Reg_n$. Every process p_i is associated with register Reg_i which can be written only by p_i and read by all processes.

There are three key ideas underlying our transformation algorithm:

(1) Every process p_i periodically writes in Reg_i a counter k, together with a specific value that we will discuss below in item (3): the counter k is incremented for every new write performed by p_i. Process p_i determines the processes that *participate* in every *write(k,*)*:

Let w_b, respectively w_e, denote the beginning event, respectively the termination event of this write operation, and let $\preceq$ be the causality relation of [17], the set of *participants* in w, is the set of processes:

$$\{p_j \in \Pi | \exists e \text{ event of } p_j : w_b \preceq e \preceq w_e\}$$

This set of participants is denoted by $P_i(k)$.

Roughly speaking, this set is determined by having every process p_j that receives some message m in the context of the k-th write from p_i, tag every message that causally follows m, with k, p_j, as well as the list of processes from which messages have been received with those tags. When p_i terminates *write(k,*)*, p_i gathers in $P_i(k)$ the participants in *write(k,*)*.

An important property of $P_i(k)$ is that it necessarily contains at least one correct process. Indeed, assume that $P_i(k)$ contains no correct processes. Consider a read operation that takes place after every process in $P_i(k)$ has crashed. Furthermore, assume that no message from processes in $P_i(k)$, sent after the first event of *write(k,*)*, reaches any process in $\Pi - P_i(k)$ before the read operation completes. This scenario is possible, since no process in $\Pi - P_i(k)$ participated in *write(k,*)*. Then the read operation cannot return the value written by *write(k,*)* — a contradiction.

(2) Every process p_i maintains a set of process sets, E_i, where each set within E_i consists of the processes that participated in some previous write performed by p_i. Basically, before *write(k,*)* in Reg_i, $E_i := \{P_i(0), P_i(1), \ldots, P_i(k-1)\}$. Initially, E_i contains exactly one set: the set of all processes Π, i.e., we assume that $P_i(0) = \Pi$. Then, whenever p_i terminates a *write*, it updates E_i.

An important property of the set E_i is that, eventually, all new sets $(P_i(k))$ that are added to E_i contain only correct processes. This is because after all faulty processes have crashed, the participants in new write operations are necessarily correct.

(3) The value that process p_i writes in Reg_i, together with k (i.e., its k-th *write*), is the value of E_i after the $(k-1)$-th *write*. After p_i writes E_i in Reg_i, p_i reads Reg_j for every $j \in \{1, \ldots, n\}$. Process p_i selects at least one process p_t from every set it reads (in some register Reg_j) by sending a message to all processes in this set and waiting for at least one reply. The value of the variable Σ-$output_i$ is the value of $P_i(k-1)$ augmented with every process p_t that p_i selected.

An important property of variable Σ-$output_i$ is that it is permanently updated if process p_i is correct. This is because p_i only waits for a message from one process in every set that it reads in a register, and every such set contains at least one correct process.

```
Code for each process p_i:

on initialization:
  P_i(0) := Π
  E_i := {P_i(0)}
{ E_i is the set of subsets of processes that participate in write on Reg_i }
  k := 0
{ k represents the number of times a write on Reg_i was invoked by p_i }
  F_i := ∅ { F_i is a temporary value of trusted processes }
  Σ-output_i := Π { Initially, all processes are trusted }

task 1:
  do forever
    k := k + 1
    Reg_i.write(k, E_i); let P_i(k) be the set of participants in the write
    E_i := E_i ∪ {P_i(k)}
    F_i := P_i(k − 1)
    forall p_j ∈ Π do
      L_j := Reg_j.read()
      forall X ∈ L_j do
        send(k, ?) to all  processes in X
        wait until receive(k, ok) from at least one process p_t ∈ X
        F_i := F_i ∪ {p_t}
    Σ-output_i := F_i
task 2:
    upon receive(l, ?) from p_j send(l, ok) to  p_j
```

Figure 1: Extracting Σ from $\mathcal{D}$ and an atomic register implementation.

In short, the *completeness* property of Σ is ensured since, for every correct process p_i, Σ-*output*$_i$ eventually contains only correct processes. The key idea that ensures the *intersection* property of Σ is that every process p_i writes in its register before reading all other registers and updating Σ-*output*$_i$. □

4. THE WEAKEST FAILURE DETECTOR TO SOLVE CONSENSUS

4.1 Specification of consensus

In the *consensus* problem, each process invokes the operation PROPOSE(v), where $v \in \{0, 1\}$, which returns a value $v' \in \{0, 1\}$. It is required that:

Termination: If every correct process proposes, then every correct process eventually returns a value.

Uniform Agreement: No two processes (whether correct or faulty) return different values.

Validity: If a process returns a value v, then v was proposed by some process.

4.2 The weakest failure detector to solve consensus

The following corollary states that consensus can be implemented using failure detector (Ω, Σ) in every environment. Such implementation can be achieved by first implementing registers out of Σ, and then consensus out of registers and Ω [19]:

Corollary 2 *For all environments $\mathcal{E}$, (Ω, Σ) can be used to solve consensus in $\mathcal{E}$.*

The following corollary follows from [3], Theorem 1 and the fact that consensus can be used to implement atomic registers in every environment [17, 21]:

Corollary 3 *For all environments $\mathcal{E}$, if $\mathcal{D}$ can be used to solve consensus in $\mathcal{E}$, then $\mathcal{D}$ can be transformed in (Ω, Σ) in $\mathcal{E}$.*

From the above two corollaries, we deduce the following one:

Corollary 4 *For all environments $\mathcal{E}$, (Ω, Σ) is the weakest failure detector to solve consensus in $\mathcal{E}$.*

5. QUITTABLE CONSENSUS (QC)

Informally, quittable consensus is a weaker version of consensus where, if a failure has occurred, processes can also agree on the special value Q. In the *quittable consensus* problem (QC), each process invokes the operation PROPOSE(v), where $v \in \{0, 1\}$, which returns a value of 0, 1 or Q (for "quit"). It is required that:

Termination: If every correct process proposes a value, then every correct process eventually returns a value.

Uniform Agreement: No two processes (whether correct or faulty) return different values.

Validity: A process may only return a value $v \in \{0, 1, \mathrm{Q}\}$. Moreover,
(a) If $v \in \{0, 1\}$ then some process previously proposed v.
(b) If $v = \mathrm{Q}$ then a failure previously occurred.

Here we defined the binary version of QC, where processes can propose values in the set $\{0, 1\}$. It is straightforward to generalise QC so that processes can propose values from an arbitrary set of at least two values that does not include the special value Q.

6. THE WEAKEST FAILURE DETECTOR TO SOLVE QC

We define a new failure detector denoted Ψ and show that it is the weakest failure detector to solve QC in any environment. To prove this, we first show that Ψ can be used to solve QC in any environment. We then prove that, for every environment $\mathcal{E}$, any failure detector that can be used to solve QC in $\mathcal{E}$ can be transformed into Ψ in $\mathcal{E}$.

6.1 Specification of failure detector Ψ

Roughly speaking Ψ behaves as follows: For an initial period of time the output of Ψ at each process is $\perp$. Eventually, however, Ψ behaves either like the failure detector (Ω, Σ) at all processes, or, *in case a failure previously occurred*, it *may* instead behave like the failure detector $\mathcal{FS}$ at all processes. The switch from $\perp$ to (Ω, Σ) or $\mathcal{FS}$ need not occur simultaneously at all processes, but the same choice is made by all processes. Note that the switch from $\perp$ to $\mathcal{FS}$ is allowable *only* if a failure previously occurred. On the other hand, if a failure does occur, processes are not *required* to switch from $\perp$ to $\mathcal{FS}$; they may still switch to (Ω, Σ).

More precisely, Ψ is defined as follows. For each failure pattern F,

$$
\begin{aligned}
H \in \Psi(F) \quad \Leftrightarrow \quad & \Big(\exists H' \in (\Omega, \Sigma)(F)\, \forall p \in \Pi \exists t \in \mathbb{N} \\
& (\forall t' < t\, H(p, t') = \perp \wedge \forall t' \geq t\, H(p, t') = H'(p, t'))\Big) \vee \\
& \Big(\exists t^* \in \mathbb{N} \Big(F(t^*) \neq \emptyset \;\wedge\; \exists H' \in \mathcal{FS}(F)\, \forall p \in \Pi\, \exists t \geq t^* \\
& (\forall t' < t\, H(p, t') = \perp \wedge \forall t' \geq t\, H(p, t') = H'(p, t'))\Big)\Big)
\end{aligned}
$$

Code for each process p:

Procedure PROPOSE(v): { *v is 1 or 0* }
while $\Psi_p = \perp$ **do nop**
if $\Psi_p \in \{\textbf{green}, \textbf{red}\}$
  then { *henceforth Ψ behaves like $\mathcal{FS}$* }
    return Q
  else { *henceforth Ψ behaves like (Ω, Σ)* }
    $d :=$ CONSPROPOSE(v)
    { *use Ψ to run (Ω, Σ)-based consensus algorithm* }
    return d

Figure 2: Using Ψ to solve QC.

6.2 Using Ψ to solve QC

It is easy to use Ψ to solve QC in any environment $\mathcal{E}$ (see Figure 2). Each process p waits until the output of Ψ becomes different from $\perp$. At that time, either Ψ starts behaving like $\mathcal{FS}$ or it starts behaving like (Ω, Σ). If Ψ starts behaving like $\mathcal{FS}$ (Ψ can do so *only* if a failure previously occurred), p returns Q. The remaining case is that Ψ starts behaving like (Ω, Σ). It is shown in [7] that there is an algorithm that uses (Ω, Σ) to solve consensus in any environment. Therefore, in this case, processes propose their initial values to that consensus algorithm and return the value decided by that algorithm. In Figure 2, CONSPROPOSE() denotes p's invocation of the algorithm that solves consensus using (Ω, Σ). Hence the following result:

Theorem 5 *For all environments $\mathcal{E}$, Ψ can be used to solve QC in $\mathcal{E}$.*

6.3 Extracting Ψ from any failure detector that solves QC

Let $\mathcal{D}$ be an arbitrary failure detector that can be used to solve QC in some environment $\mathcal{E}$; i.e., there is an algorithm $\mathcal{A}$ that uses $\mathcal{D}$ to solve QC in environment $\mathcal{E}$. We must prove that Ψ can be "extracted" from $\mathcal{D}$ in environment $\mathcal{E}$, i.e., processes can run in $\mathcal{E}$ a transformation algorithm that uses $\mathcal{D}$ and $\mathcal{A}$ to generate the output of Ψ — a failure detector that initially outputs $\perp$ and later behaves either like (Ω, Σ) or like $\mathcal{FS}$. The transformation algorithm that does this is shown in Figure 3 and is explained below.

Each process p starts by outputting $\perp$ (line 1). While doing so, p determines whether in the current run it is possible to extract (Ω, Σ), or it is legitimate to start behaving like $\mathcal{FS}$ and output **red** because a failure occurred, as follows.

In task 1, p simulates runs of $\mathcal{A}$ that could have occurred in the current failure detector history of $\mathcal{D}$ and the current failure pattern F, exactly as in [3]. It does this by "sampling" its local module of $\mathcal{D}$ and exchanging failure detector samples with the other processes (line 4). Process p organizes these samples into an ever-increasing DAG G_p whose edges are consistent with the order in which the failure detector samples were actually taken. Using G_p, p simulates ever-increasing partial runs of algorithm $\mathcal{A}$ that are compatible with paths in G_p (line 6).[5] Each process p organizes these runs into a forest of $n+1$ trees, denoted Υ_p. For any i, $0 \le i \le n$, the i-th tree of this forest, denoted Υ_p^i, corresponds to simulated runs of $\mathcal{A}$,

[5]Each failure detector sample in G_p includes the name of the process that took this sample. Roughly speaking, we say that a run or schedule of $\mathcal{A}$ is compatible with a path in G_p if the sequence of processes that take steps in this run or schedule and the failure detector values that they see match the sequence of processes and failure detector values in this path [3].

Code for each process p:

on initialization:
1 Ψ-*output*$_p := \perp$ { *Ψ-output$_p$ is the output of p's module of Ψ* }

task 1:
2 **do forever** { *This is done exactly as in [3]* }
3 **cobegin**
4   p builds an ever-increasing DAG G_p of failure detectors samples by repeatedly sampling its failure detector and exchanging samples with other processes.
5 $||$
6   p uses G_p and the $n+1$ initial configurations to construct a forest Υ_p of ever-increasing simulated runs of algorithm $\mathcal{A}$ using $\mathcal{D}$ that could have occurred with the current failure pattern F and the current failure detector history $H \in D(F)$.
7 **coend**

task 2:
8 **wait until** p decides in some run of *every* tree of the forest Υ_p
9 **if** p decides Q in some run
10   **then**
11   p executes $\mathcal{A}$ by proposing 0
12   **else** { *every tree of Υ_p has a run where p decides 0 or 1* }
13   let I and I' be initial configurations that differ only in the proposal of one process and S and S' be schedules in Υ_p so that p decides 0 in $S(I)$ and 1 in $S'(I')$
14   p executes $\mathcal{A}$ by proposing (I, I', S, S')
15 **wait until** p decides in this execution of $\mathcal{A}$
16 **if** p decides 0 or Q
17   **then** { *extract $\mathcal{FS}$* }
18   Ψ-*output*$_p :=$ **red**
19   **else** { *p's decision is of the form (I_0, I_1, S_0, S_1)* }
20   Ω-*output*$_p := p$; Σ-*output*$_p := \Pi$ { *extract (Ω, Σ)* }
21   **cobegin**
       { *extract Ω* }
22   **do forever** Ω-*output*$_p :=$ id of the process that p extracts using Υ_p and the procedure described in [3]
23     $||$
       { *extract Σ* }
24   let (I_0, I_1, S_0, S_1) be the decision value of p
25   let $\mathcal{C}$ be the set of configurations reached by applying all prefixes of S_0, S_1 to I_0, I_1, respectively
26   **do forever**
27   **wait until** p adds a new failure detector sample u to its DAG G_p
28   **repeat**
29   let $G_p(u)$ be the subgraph induced by the descendants of u in G_p
30   **for each** $C \in \mathcal{C}$ construct the set $\mathcal{S}_C$ of all schedules compatible with some path of $G_p(u)$ and applicable to C
31   **until** for each $C \in \mathcal{C}$ there is a schedule $S \in \mathcal{S}_C$ such that p decides in $S(C)$
32   Σ-*output*$_p := \bigcup_{C \in \mathcal{C}}$ set of processes that take steps in the schedule $S \in \mathcal{S}_C$ such that p decides in $S(C)$
33     $||$
       { *combine Ω and Σ to Ψ* }
34   **do forever** Ψ-*output*$_p := (\Omega$-*output*$_p, \Sigma$-*output*$_p)$
35   **coend**

Figure 3: Extracting Ψ from $\mathcal{D}$ and QC algorithm $\mathcal{A}$

all starting with the same initial configuration, namely the one in which processes $p_1, \ldots, p_i$ propose 1, and $p_{i+1}, \ldots, p_n$ propose 0. A path from the root of a tree to a node x in this tree corresponds to (the schedule of) a partial run of $\mathcal{A}$, where every edge along the path corresponds to a step of some process.

In task 2, p waits until it decides in some run of every tree of the forest Υ_p (line 8). If p decides Q in any of these runs, then a failure must have occurred (in the current failure pattern), and so p knows that it is legitimate to output **red** in this run. Otherwise (p's decisions in the simulated runs are 0s or 1s), p determines that it is possible to extract (Ω, Σ) in the current run.

At this point, p executes the given QC algorithm $\mathcal{A}$ (using failure detector $\mathcal{D}$) to *agree* with all the other processes on whether to output **red** or to extract (Ω, Σ). Specifically, if p has determined that it is legitimate to output **red** (p decides Q in some run of Υ_p) then it proposes 0 to $\mathcal{A}$ (line 11). Otherwise, in each tree of Υ_p, p has a run in which it decides 0 or 1. In the tree where every process proposes 0 (respectively, 1), p's decision must be 0 (respectively, 1). Thus, there exist initial configuration I and I', and schedules S and S' in Υ_p, such that I and I' are initial configurations that differ only in the proposal of one process, and S and S' are schedules in Υ_p such that p decides 0 in $S(I)$ and 1 in $S'(I')$. Then p proposes (I, I', S, S') to $\mathcal{A}$(line 14).[6]

If $\mathcal{A}$ returns 0 or Q, then p stops outputting $\perp$ and outputs **red** from that time on (line 18). If $\mathcal{A}$ returns a value of the form (I_0, I_1, S_0, S_1), then p stops outputting $\perp$ and starts extracting Ω (line 22) and Σ (lines 24-32). Ω is extracted as in [3] (see Section 6.3.1). Σ is extracted using novel techniques explained in Section 6.3.2.

Note that processes use the given QC algorithm $\mathcal{A}$ and failure detector $\mathcal{D}$ in two different ways and for different purposes. First each process *simulates* many runs of $\mathcal{A}$ to determine whether it is legitimate to output **red** or it is possible to extract (Ω, Σ) in the current run. Then processes actually execute $\mathcal{A}$ (this is a *real* execution, not a simulated one) to reach a common decision on whether to output **red** or to extract (Ω, Σ). Finally, if processes decide to extract (Ω, Σ), they resume the simulation of runs of $\mathcal{A}$ to do this extraction.

6.3.1 Extracting Ω

To extract Ω, p must continuously output the id of a process such that, after some time, correct processes output the id of the same correct process. This is done using the procedure of [3], with some minor differences explained below.

As in [3], because of the way each process p constructs its ever-increasing forest Υ_p of simulated runs, the forests of correct processes tend to the same infinite limit forest, denoted Υ. The limit tree of Υ_p^i is denoted Υ^i. Each node x of the limit forest Υ is tagged by the set of decisions reached by correct processes in partial runs that correspond to descendants of x.

In [3] the only possible decisions were 0 or 1, and so these were the only possible tags. Consequently, each node was 0-*valent*, 1-*valent* or *bivalent* (with two tags). Here there are three possible decisions (0, 1 or Q) so each node is 0-*valent*, 1-*valent*, *Q-valent* or *multivalent* (with two or three tags).

In [3] (and here) the extraction of the id of a common correct process relies on the existence of a *critical index* i in the limit forest Υ. Here we define i to be critical if the root of Υ^i is multivalent (in which case it is called *multivalent critical*), or if the root of Υ^{i-1} is u-valent and the root of Υ^i is v-valent, where $u, v \in \{0, 1, \mathrm{Q}\}$ and $u \neq v$ (in which case it is called *univalent critical*).

[6] We assume here that $\mathcal{A}$ can solve multivalued QC. This causes no loss of generality: by using the technique of [20] one can transform any binary QC algorithm into a multivalued one.

In [3] it is shown that a critical index always exists. In this paper, however, this is not necessarily the case. If some process crashes (in the current failure pattern), it is possible that in all the simulated runs of QC algorithm $\mathcal{A}$ in Υ all decisions are Q. In this case, the roots of all trees in the limit forest Υ are tagged only with Q. So there is no critical index, and we cannot apply the techniques of [3] to extract the id of a correct process! This is why, in our transformation algorithm, processes do not always attempt to extract Ω from $\mathcal{D}$. We will show, however, that if a process actually attempts to extract Ω (in line 22) then a critical index does exist in the limit forest Υ, and so Ω can indeed be extracted (Lemma 8 in [12]).

6.3.2 Extracting Σ

To extract Σ, p must continuously output a set of processes (quorum) such that the quorums of all processes always intersect, and eventually they contain only correct processes. This is done in lines 24-32 as follows.

When process p reaches line 24, it has agreed with other processes on two initial configurations I_0 and I_1 and two schedules S_0 and S_1 that are applicable to I_0 and I_1, respectively. Consider the set $\mathcal{C}$ of configurations of $\mathcal{A}$ obtained by applying all the prefixes of S_0 and S_1 to I_0 and I_1 (line 25).

To determine its next quorum, p uses "fresh" failure detector samples to simulate runs of $\mathcal{A}$ that extend each configuration in $\mathcal{C}$ (lines 29-30). It does so until, for each configuration in $\mathcal{C}$, it has simulated an extension in which it has decided (line 31). The quorum of p is the set of all processes that take steps in these "deciding" extensions (line 32).

Note that in line 27, p waits until it gets a new sample u from its failure detector module (which happens in line 4 of task 1) and then it uses only samples that are more recent than u to extend the configurations in $\mathcal{C}$ (lines 29-30). This ensures the freshness of the failure detector samples that p uses to determine its quorums. Consequently, quorums eventually contain only correct processes (one of the two requirements of Σ).

Theorem 6 *For all environments $\mathcal{E}$, if failure detector $\mathcal{D}$ can be used to solve QC in $\mathcal{E}$, then the algorithm in Figure 3 transforms $\mathcal{D}$ into Ψ in environment $\mathcal{E}$.*

PROOF SKETCH. Let $\mathcal{A}$ be any algorithm that uses $\mathcal{D}$ to solve QC in environment $\mathcal{E}$. We show that the algorithm in Figure 3 uses $\mathcal{A}$ to transform $\mathcal{D}$ into Ψ in environment $\mathcal{E}$. In that algorithm, each process p maintains a variable Ψ-$output_p$. We now prove that the values that these variables take conform to the specification of Ψ. By inspection of Figure 3, it is clear that Ψ-$output_p$ is either $\perp$, or **red** (in which case we say it is of type $\mathcal{FS}$), or a pair (q, Q) where $q \in \Pi$ and $Q \subseteq \Pi$ (in which case we say it is of type (Ω, Σ)).

(1) *For each process p, Ψ-$output_p$ is initially $\perp$ (line 1). If Ψ-$output_p$ ever changes value, it becomes of type $\mathcal{FS}$ forever (line 18) or of type (Ω, Σ) forever (lines 20-34).*

(2) *For all processes p and q, it is impossible for Ψ-$output_p$ to be of type $\mathcal{FS}$ and Ψ-$output_q$ to be of type (Ω, Σ).* This is because, by the Uniform Agreement property of $\mathcal{A}$, p and q cannot decide different values in line 15.

(3) *For each correct process p, eventually Ψ-$output_p \neq \perp$.* To see this, let p be any correct process. Process p simulates a forest Υ_p of ever-increasing partial runs of $\mathcal{A}$ as in [3] (see line 6). In this simulation, every tree in Υ_p has runs in which all the correct processes take steps infinitely often. So, by the Termination property of QC, every tree in Υ_p has a run in which p

decides. Therefore, eventually *all* correct processes complete the wait statement in line 8, and execute $\mathcal{A}$ in line 11 or 14. By the Termination property of QC, eventually p decides in that execution of $\mathcal{A}$, and stops waiting in line 15. Thus, p eventually sets Ψ-$output_p$ to a value other than $\perp$ in line 18 or 34.

(4) *For each process p, if Ψ-$output_p$ is* **red** *then a process previously crashed in the current run.* To see this, let p be some process that sets Ψ-$output_p$ = **red** (line 18). Thus, p decides 0 or Q in the execution of $\mathcal{A}$ that it invoked in line 11 or 14. If p decides Q then the fact that some process has previously crashed in the current run follows immediately from part (b) of the Validity property of QC. If p decides 0 then from part (a) of the Validity property of QC, some process q proposed 0 in the execution of $\mathcal{A}$ that q invoked in line 11. This implies that q decided Q in one of the simulated runs of $\mathcal{A}$ that q has in its forest Υ_q. Recall that these are runs that could have occurred with the current failure pattern. By part (b) of the Validity property of QC, this means that some process has previously crashed in the current run.

(5) *If the Ψ-output variable of any process is ever of type (Ω, Σ), then there is a time after which, for every correct process p, Ω-$output_p$ is the id of the same correct process.* To see this, suppose some Ψ-*output* becomes of type (Ω, Σ). Then, by (2) and (3) above, eventually the Ψ-*output* variable of every correct process also become of type (Ω, Σ). So every correct process sets its Ω-*output* variable repeatedly in line 22 using the extraction procedure described in [3]. Since processes reach line 22, by Lemma 8 in [12], a critical index exists in the limit forest Υ. By following the proof of [3], it can now be shown that eventually all the correct processes extract the id of the same correct process. The only difference is that whenever [3] refers to a *bi*valent node, we now refer to a *multi*valent one, and whenever [3] refers to 0-valent versus 1-valent nodes, we refer here to u-valent and v-valent nodes where $u, v \in \{0, 1, \mathrm{Q}\}$ and $u \neq v$.

(6) *If the Ψ-output variable of any process is ever of type (Ω, Σ) then: (a) for every correct process p, there is a time after which Σ-$output_p$ contains only correct processes, and (b) for every processes p and q, Σ-$output_p$ and Σ-$output_q$ always intersect.* This is shown in Lemmas 11 and 12 in [12].

From the above, it is clear that the values of the variables Ψ-*output* conform to Ψ: For an initial period of time they are equal to $\perp$. Eventually, however, they behave either like the failure detector (Ω, Σ) at all processes or, if a failure occurs, they may instead behave like the failure detector $\mathcal{FS}$ at all processes. Moreover, this switch from $\perp$ to (Ω, Σ) or $\mathcal{FS}$ is consistent at all processes. □

From Theorems 5 and 6, we have:

Corollary 7 *For all environments $\mathcal{E}$, Ψ is the weakest failure detector to solve QC in $\mathcal{E}$.*

7. THE WEAKEST FAILURE DETECTOR TO SOLVE NBAC

7.1 Specification of NBAC

In the *non-blocking atomic commit* problem (NBAC), each process invokes the operation VOTE(v), where $v \in \{\text{Yes}, \text{No}\}$, which returns either Commit or Abort. It is required that:

```
Code for each process p:

Procedure VOTE(v): { v is Yes or No }
  send v to all
  wait until [(for each process q in Π, received q's vote) or FS = red]
  if the votes of all processes are received and are Yes then
     myproposal := 1
  else { some vote was No or there was a failure }
     myproposal := 0
  mydecision := PROPOSE(myproposal)
  if mydecision = 1 then
     return Commit
  else { mydecision = 0 or Q }
     return Abort
```

Figure 4: Using $\mathcal{FS}$ to transform QC into NBAC

Termination: If every correct process votes, then every correct process eventually returns a value.

Uniform Agreement: No two processes (whether correct or faulty) return different values.

Validity: A process may only return Commit or Abort. Moreover,
(a) If v = Commit then all processes previously voted Yes.
(b) If v = Abort then either some process previously voted No or a failure previously occurred.

7.2 Using $\mathcal{FS}$ to relate NBAC and QC

We first show that NBAC is equivalent to the combination of QC and failure detector $\mathcal{FS}$. We then use this result to establish a relationship between the weakest failure detector to solve QC and the one to solve NBAC.

Theorem 8 *NBAC is equivalent to QC and $\mathcal{FS}$. That is, in every environment $\mathcal{E}$:*
(a) Given failure detector $\mathcal{FS}$, any solution to QC can be transformed into a solution to NBAC.
(b) Any solution to NBAC can be transformed into a solution to QC, and can be used to implement $\mathcal{FS}$.

PROOF SKETCH. Let $\mathcal{E}$ be an arbitrary environment.

(a) The algorithm in Figure 4 uses $\mathcal{FS}$ to transform QC into NBAC in $\mathcal{E}$.

(b) It is known that NBAC can be used to implement $\mathcal{FS}$ in any environment [5, 11]. Roughly speaking, processes use the given NBAC algorithm repeatedly (forever), voting Yes in each instance. At each process, the output of $\mathcal{FS}$ is initially **green**, and becomes permanently **red** if and when an instance of NBAC returns Abort. It remains to prove that any solution to NBAC in $\mathcal{E}$ can be transformed into a solution to QC in $\mathcal{E}$. This transformation is shown in Figure 5. □

7.3 The weakest failure detector to solve NBAC

Theorem 9 *For every environment $\mathcal{E}$, if $\mathcal{D}$ is the weakest failure detector to solve QC in $\mathcal{E}$, then $(\mathcal{D}, \mathcal{FS})$ is the weakest failure detector to solve NBAC in $\mathcal{E}$.*

PROOF SKETCH. Let $\mathcal{E}$ be an arbitrary environment, and $\mathcal{D}$ be the weakest failure detector to solve QC in $\mathcal{E}$. This means that: (i) $\mathcal{D}$

Code for each process p:

Procedure PROPOSE(v): { *v is* 1 *or* 0 }
send v **to** all
$d :=$ VOTE(Yes) { *use of the given NBAC algorithm* }
if $d =$ Abort **then**
  return Q
else
  wait until [(for each process $q \in \Pi$, received q's proposal)]
  return smallest proposal received

Figure 5: Transforming NBAC into QC

can be used to solve QC in $\mathcal{E}$ and (ii) any failure detector that solves QC in $\mathcal{E}$ can be transformed into $\mathcal{D}$ in $\mathcal{E}$.

Let $\mathcal{D}' = (\mathcal{D}, \mathcal{FS})$. We must show that: (a) $\mathcal{D}'$ can be used to solve NBAC in $\mathcal{E}$, and (b) any failure detector that solves NBAC in $\mathcal{E}$ can be transformed into $\mathcal{D}'$ in $\mathcal{E}$.

(a) Since the output of $\mathcal{D}'$ includes the output of $\mathcal{D}$, by (i), $\mathcal{D}'$ can be used to solve QC in $\mathcal{E}$. Since $\mathcal{D}'$ also includes $\mathcal{FS}$, by Theorem 8(a), $\mathcal{D}'$ can be used to solve NBAC in $\mathcal{E}$.

(b) Let $\mathcal{D}''$ be a failure detector that solves NBAC in $\mathcal{E}$. By Theorem 8(b), (1) $\mathcal{D}''$ can be used to solve QC in $\mathcal{E}$, and (2) $\mathcal{D}''$ can be used to implement $\mathcal{FS}$ in $\mathcal{E}$. From (1) and (ii), $\mathcal{D}''$ can be transformed into $\mathcal{D}$ in $\mathcal{E}$. By (2), $\mathcal{D}''$ can be transformed into $(\mathcal{D}, \mathcal{FS})$, i.e., into $\mathcal{D}'$, in $\mathcal{E}$. □

From Corollary 7 and Theorem 9, we immediately have:

Corollary 10 *For all environments $\mathcal{E}$, $(\Psi, \mathcal{FS})$ is the weakest failure detector to solve NBAC in $\mathcal{E}$.*

8. REFERENCES

[1] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, 1995.

[2] H. Attiya and J. L. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. UK: McGraw-Hill, 1998.

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

[4] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, Mar. 1996.

[5] B. Charron-Bost and S. Toueg. Unpublished notes, 2001.

[6] C. Delporte-Gallet, H. Fauconnier, and R. Guerraoui. Failure detection lower bounds on registers and consensus. In D. Malkhi, editor, *Distributed Computing*, LNCS 2508. 16th Internationl conference DISC 2002, 2002.

[7] C. Delporte-Gallet, H. Fauconnier, and R. Guerraoui. Shared memory vs. message passing. Technical Report IC/2003/77, EPFL, Dec. 2003. Availabe at http://icwww.epfl.ch/publications/.

[8] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, Apr. 1985.

[9] E. Fromentin, M. Raynal, and F. Tronel. On classes of problems in asynchronous distributed systems with process crashes. In *Proceedings of the 19th International Conference on Distributed Computing Systems*, pages 470–477. IEEE Computer Society Press, 1999.

[10] J. Gray. Notes on database operating systems. In R. Bayer, R. M. Graham, and G. Seegmuller, editors, *Operating Systems: An Advanced Course*, volume 60 of *LNCS*, pages 393–481. Springer-Verlag, 1978.

[11] R. Guerraoui. Non-blocking atomic commit in asynchronous distributed systems with failure detectors. *Distributed Computing*, 15(1):17–25, 2002.

[12] R. Guerraoui, V. Hadzilacos, P. Kouznetsov, and S. Toueg. The weakest failure detectors for quittable consensus and non-blocking atomic commit. Technical report, LPD, EPFL, 2004. Availabe at http://lpdwww.epfl.ch/publications/.

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

[14] V. Hadzilacos. On the relationship between the atomic commitment and consensus problems. *Fault-Tolerant Distributed Computing*, pages 201–208, 1987.

[15] M. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, June 1990.

[16] A. Israeli and M. Li. Bounded time-stamps. *Distributed Computing*, 6(4):205–209, July 1993.

[17] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[18] L. Lamport. On interprocess communication–Part I: Basic Formalism, Part II: Algorithms. *Distributed Computing*, 1,2(2):87–103, 1986.

[19] W.-K. Lo and V. Hadzilacos. Using failure detectors to solve consensus in asynchronous shared memory systems. In G. Tel and P. Vitányi, editors, *Proceedings of the eighth International Workshop on Distributed Algorithms*, volume 857 of *LNCS*, pages 280–295. Springer-Verlag, Sept. 1994.

[20] A. Mostefaoui, M. Raynal, and F. Tronel. From binary consensus to multivalued consensus in asynchronous message-passing systems. *Inf. Process. Lett.*, 73(5–6):207–212, March 2000.

[21] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.

[22] D. Skeen. *Crash Recovery in a Distributed Database System*. PhD thesis, University of California at Berkeley, May 1982. Technical Memorandum UCB/ERL M82/45.

[23] P. Vitanyi and B. Awerbuch. Atomic shared register access by asynchronous hardware. In *Proceedings of the 27th Symposium on Foundations of Computer Science*, pages 233–246, 1986.