The weakest failure detectors to solve Quittable Consensus and Non-Blocking Atomic Commit[*]

Rachid Guerraoui[1] Vassos Hadzilacos[2] Petr Kouznetsov[1] Sam Toueg[2]

(1) Distributed Programming Laboratory, EPFL
(2) Department of Computer Science, University of Toronto

{rachid.guerraoui,petr.kouznetsov}@epfl.ch {vassos,sam}@cs.toronto.edu

Abstract

We introduce *quittable consensus*, a natural variation of the consensus problem, where processes have the option to agree on "quit" if failures occur, and we relate this problem to the well-known problem of non-blocking atomic commit. We then determine the weakest failure detectors for these two problems in all environments, regardless of the number of faulty processes.

1 Introduction

Non-blocking atomic commit (NBAC) is a well-known problem that arises in distributed transaction processing [8]. 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. NBAC is similar to the classical problem of consensus, where each process initially proposes a value, and eventually processes must reach a common decision on one of the proposed values.

It is well-known that NBAC and consensus are unsolvable in asynchronous systems with process crashes (even if communication is reliable) [6]. One way to circumvent such impossibility results is through the use of *unreliable failure detectors* [2]: 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 *at al.* [1] determined the *weakest* failure detector to solve consensus in systems with a majority of correct processes, while Delporte *et al.* [4]generalized this result to all systems, regardless of the number of correct processes. Informally, $\mathcal{D}$ is the weakest failure detector to solve problem P if (a) there is an algorithm that uses $\mathcal{D}$ to solve P, and (b) any failure detector $\mathcal{D}'$ that can be used to solve P can be transformed to $\mathcal{D}$.

As with consensus, failure detectors can be used to solve NBAC [9, 7]. It was an open problem, however, whether there is a weakest failure detector to solve NBAC and, if so, what that failure detector is. In this paper we resolve this problem. To do so,

(a) we introduce a natural variation of consensus, called *quittable consensus* (QC) — a problem that is interesting in its own right;

(b) we determine the weakest failure detector to solve QC;

[*]The results of this paper were first presented at the 23rd Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC 2004) [5].

(c) we establish a close relationship between QC and NBAC; and

(d) we use (b) and (c) to derive the weakest failure detector to solve 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 now describe in more detail our results, which involve the following three failure detectors.

- 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 [1].

- 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 consists of only correct processes [4].

- 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 process [3, 9].

We first show that there is a weakest failure detector to solve QC. This failure detector, which we denote Ψ, is closely related to the weakest failure detector to solve consensus, namely (Ω, Σ) [4],[1] and to $\mathcal{FS}$. Specifically, 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 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 bevariour 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. 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.

NBAC has been studied extensively in the context of transaction processing [8, 13]. Its relation to consensus was first explored in [11]. Charron-Bost and Toueg [3] and Guerraoui [9] 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.[2] The problem of determining the weakest failure detector to solve

[1]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 $\mathcal{D}'$.

[2]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.

NBAC was explored and settled in special settings. Fromentin *et al.* [7] determine that to solve NBAC between *every* pair of processes in the system, one needs a *perfect failure detector* [2]. Guerraoui and Kouznetsov [10] determine the weakest failure detector for NBAC in a restricted class of failure detectors; while from results of [3] and [9] 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.

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 give the exact specification of QC, and in Section 4 we prove that Ψ is the weakest failure detector to solve QC. In Section 5 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.

2 Model

We consider the asynchronous message-passing model in which processes have access to failure detectors [1, 2]. 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 behaviour 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.[3] 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.

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

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 [1].

Some algorithms meet their specification only under some assumptions 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,

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

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

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

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

$$\begin{aligned} H \in \mathcal{FS}(F) \quad \Leftrightarrow \quad & \forall p \in \Pi\, \forall t \in \mathbb{N}\; \big(H(p,t) = \textbf{red} \rightarrow F(t) \neq \emptyset\big) \wedge \\ & \big(\mathit{faulty}(F) \neq \emptyset \rightarrow \forall p \in \mathit{correct}(F)\, \exists t \in \mathbb{N}\, \forall t' \geq t\; H(p,t') = \textbf{red}\big). \end{aligned}$$

3 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 decision 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 decision.

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

Validity: A process may only decide 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.

```
    Code for each process $p$:
Procedure PROPOSE$(v)$: { $v$ is $1$ or $0$ }
    while $\Psi_p = \bot$ do nop
    if $\Psi_p \in \{$green, red$\}$
        then { henceforth $\Psi$ behaves like $\mathcal{FS}$ }
            return Q
        else { henceforth $\Psi$ behaves like $(\Omega, \Sigma)$ }
            $d :=$ CONSPROPOSE$(v)$ { use $\Psi$ to run $(\Omega, \Sigma)$-based consensus algorithm }
            return $d$
```

Figure 1: Using Ψ to solve QC.

4 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 $\mathcal{E}$.

4.1 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 $\bot$. 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 $\bot$ 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 $\bot$ to $\mathcal{FS}$ is allowable *only* if a failure previously occurred. Furthermore, if a failure does occur processes are not *required* to switch from $\bot$ to $\mathcal{FS}$; they may still switch to (Ω, Σ).

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

$$H \in \Psi(F) \quad \Leftrightarrow \quad \Big(\exists H' \in (\Omega, \Sigma)(F)\, \forall p \in \Pi \exists t \in \mathbb{N}\, \big(\forall t' < t\, H(p,t') = \bot \land \forall t' \geq t\, H(p,t') = H'(p,t')\big)\Big) \lor$$
$$\Big(\exists t^* \in \mathbb{N}\, F(t^*) \neq \emptyset \;\land\; \exists H' \in \mathcal{FS}(F)\, \forall p \in \Pi\, \exists t \geq t^* \big(\forall t' < t\, H(p,t') = \bot \land \forall t' \geq t\, H(p,t') = H'(p,t')\big)\Big)$$

4.2 Using Ψ to solve QC

It is easy to use Ψ to solve QC in any environment $\mathcal{E}$ (see Figure 1). Each process p waits until the output of Ψ becomes different from $\bot$. 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 [4] that there is an algorithm that uses (Ω, Σ) to solve consensus in any environment. Therefore, in this case, processes propose their initial value to that consensus algorithm and return the value decided by that algorithm. In Figure 1, CONSPROPOSE() denotes p's invocation of the algorithm that solves consensus using (Ω, Σ). Hence the following result:

Theorem 1 *For all environments $\mathcal{E}$, Ψ can be used to solve QC in $\mathcal{E}$.*

```
    Code for each process $p$:
on initialization:
    $\Psi$-output$_p$ := $\perp$ { $\Psi$-output$_p$ is the output of $p$'s module of $\Psi$ }
task 1:
    do forever { This is done exactly as in [1] }
    cobegin
       $p$ builds an ever-increasing DAG $G_p$ of failure detectors samples
          by repeatedly sampling its failure detector and exchanging samples
          with other processes.
    ||
       $p$ uses $G_p$ and the $n+1$ initial configurations to construct a forest $\Upsilon_p$
          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 $H \in D(F)$.
    coend
task 2:
    wait until $p$ decides in some run of every tree of the forest $\Upsilon_p$
    if $p$ decides Q in some run
       then
          $p$ executes $\mathcal{A}$ by proposing $0$
       else { every tree of the forest $\Upsilon_p$ has a run where $p$ decides $0$ or $1$ }
          let $I$ and $I'$ be initial configurations that differ only in the proposal of one process
             and $S$ and $S'$ be schedules in $\Upsilon_p$ so that $p$ decides $0$ in $S(I)$ and $1$ in $S'(I')$
          $p$ executes $\mathcal{A}$ by proposing $(I, I', S, S')$
    wait until $p$ decides in this execution of $\mathcal{A}$
    if $p$ decides $0$ or Q
       then { extract $\mathcal{FS}$ }
          $\Psi$-output$_p$ := red
       else { $p$'s decision is of the form $(I_0, I_1, S_0, S_1)$; extract $(\Omega, \Sigma)$ }
          $\Omega$-output$_p$ := $p$ ; $\Sigma$-output$_p$ := $\Pi$
          cobegin
             { extract $\Omega$ }
             do forever $\Omega$-output$_p$ := id of the process that $p$ extracts using $\Upsilon_p$ and the procedure described in [1]
          ||
             { extract $\Sigma$ }
             let $(I_0, I_1, S_0, S_1)$ be the decision value of $p$
             let $\mathcal{C}$ be the set of configurations reached by applying all prefixes of $S_0, S_1$ to $I_0, I_1$, respectively
             do forever
                wait until $p$ adds a new failure detector sample $u$ to its DAG $G_p$
                repeat
                   let $G_p(u)$ be the subgraph induced by the descendants of $u$ in the current DAG of samples $G_p$.
                   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 $C$
                until for each $C \in \mathcal{C}$ there is a schedule $S \in \mathcal{S}_C$ such that $p$ decides in $S(C)$
                $\Sigma$-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 $S(C)$
          ||
             { combine $\Omega$ and $\Sigma$ to $\Psi$ }
             do forever $\Psi$-output$_p$ := $(\Omega$-output$_p, \Sigma$-output$_p)$
          coend
```

Figure 2: Extracting Ψ from $\mathcal{D}$ and QC algorithm $\mathcal{A}$

4.3 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 2 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 [1]. 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).[4] 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 $\mathcal{A}$ 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** then it proposes 0 to $\mathcal{A}$ (line 11), otherwise it proposes (I, I', S, S') (line 14) where 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 $S'(I')$.[5] The following lemma proves that these configurations and schedules exist.

Lemma 2 *If any process p reaches line 12 then there are initial configurations I and I', and schedules S and S' in Υ_p, such that (a) I and I' differ only in the proposal of one process, and (b) p decides 0 in $S(I)$ and 1 in $S'(I')$.*

PROOF SKETCH. If any process p reaches line 12, then 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). The result immediately follows. $\square$

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 [1] (see Section 4.3.1). Σ is extracted using novel techniques explained in Section 4.3.2.

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.

[4]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 [1].

[5]We assume here that $\mathcal{A}$ can solve multivalued QC. This causes no loss of generality: by using the technique of [12] one can transform any binary QC algorithm into a multivalued one.

4.3.1 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 [1], with some minor differences explained below.

As in [1], 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 [1] 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 [1] (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*).

In [1] 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 [1] to extract the id of a correct process! This is why, in our transformation algorithm, processes do not always attempt to extract Ω from $\mathcal{D}$. However, 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 3 *If any process reaches line 22 then the limit forest Υ has a critical index.*

PROOF SKETCH. If a process reaches line 22, then it previously decided a tuple of the form (I_0, I_1, S_0, S_1) in line 15. Thus, by the Validity property of $\mathcal{A}$, some process q (not necessarily correct) proposed some tuple in line 14. We first show that the limit forest Υ has some run where some *correct* process decides a value other than Q.

Since q proposed a tuple in line 14, it must have decided some value $v \neq \mathrm{Q}$ in some partial run R of $\mathcal{A}$ in Υ_q. Before q proposed its tuple in line 14, it sent to all processes the finite path of q's DAG (of failure detector samples) that gave rise to the partial run R. Thus, after receiving this path and integrating it in its own DAG, every correct process p also constructs partial run R (which is compatible with this path), and includes it in its own forest Υ_p. So the partial run R where q decides v is also embedded in the limit forest Υ. Note that Υ includes an infinite run R^* that extends R such that all the correct processes take an infinite number of steps in R^*. By the Termination and Uniform Agreement properties of $\mathcal{A}$, all the correct processes decide v (the same as q) in run R^*. So Υ has a run where all correct processes decide $v \neq \mathrm{Q}$.

From the above, the root of some tree Υ^j of the limit forest Υ has tag $v \neq \mathrm{Q}$. Without loss of generality, assume $v = 0$. Note that the root of tree Υ^n, where all processes propose 1, must have a tag $u \neq 0$ (it can be 1 or Q). Therefore, some index i between j and n must be critical. □

4.3.2 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).

4.3.3 Sketch of the proof

Lemma 4 *Let $R_1 = \langle F, H, I, S \cdot S_1, T \cdot T_1 \rangle$ and $R_2 = \langle F, H, I, S \cdot S_2, T \cdot T_2 \rangle$ be partial runs of $\mathcal{A}$, such that the sets of processes that take steps in S_1 and in S_2 are disjoint and T_1 and T_2 contain distinct times. Let $\hat{T}$ be the merging of T_1 and T_2 (in increasing order) and $\hat{S}$ be the corresponding merging of S_1 and S_2 (i.e., the steps of $\hat{S}$ are the steps of S_1 and S_2 in the order indicated by $\hat{T}$). Then $\hat{R} = \langle F, H, I, S \cdot \hat{S}, T \cdot \hat{T} \rangle$ is also a partial run of $\mathcal{A}$.*

Corollary 5 *Let $R_1 = \langle F, H, I, S \cdot S_1, T \cdot T_1 \rangle$ and $R_2 = \langle F, H, I, S \cdot S_2, T \cdot T_2 \rangle$ be partial runs of $\mathcal{A}$, such that the sets of processes that take steps in S_1 and in S_2 are disjoint and T_1 and T_2 contain distinct times. If some process decides x_1 in R_1 and some process decides x_2 in R_2 then $x_1 = x_2$.*

Lemma 6 *For each correct process p, there is a time after which Σ-output$_p$ contains only correct processes.*

PROOF SKETCH. Let p be a correct process. Note that: (a) p takes a new failure detector sample u infinitely often (in line 27), and (b) Σ-*output*$_p$ contains only processes that take steps after the most recent sample u taken by p. Since faulty processes eventually stop taking steps, there is a time after which Σ-*output*$_p$ does not contain any faulty process. □

Lemma 7 *Let p, q be any processes, Σ-output$_p$ and Σ-output$_q$ always intersect.*

PROOF SKETCH. Recall that p and q agreed on a value of the form (I_0, I_1, S_0, S_1) in a (real) execution of $\mathcal{A}$ (line 15). I_0 and I_1 are initial configurations that differ only in the proposal of one process, and S_0 and S_1 are (simulated) schedules of $\mathcal{A}$ in which some process decides 0 in $S_0(I_0)$ and 1 in $S_1(I_1)$. Thus, p and q also agree on the set of configurations $\mathcal{C}$ that is obtained by applying all prefixes of S_0 and S_1 to I_0 and I_1, respectively.

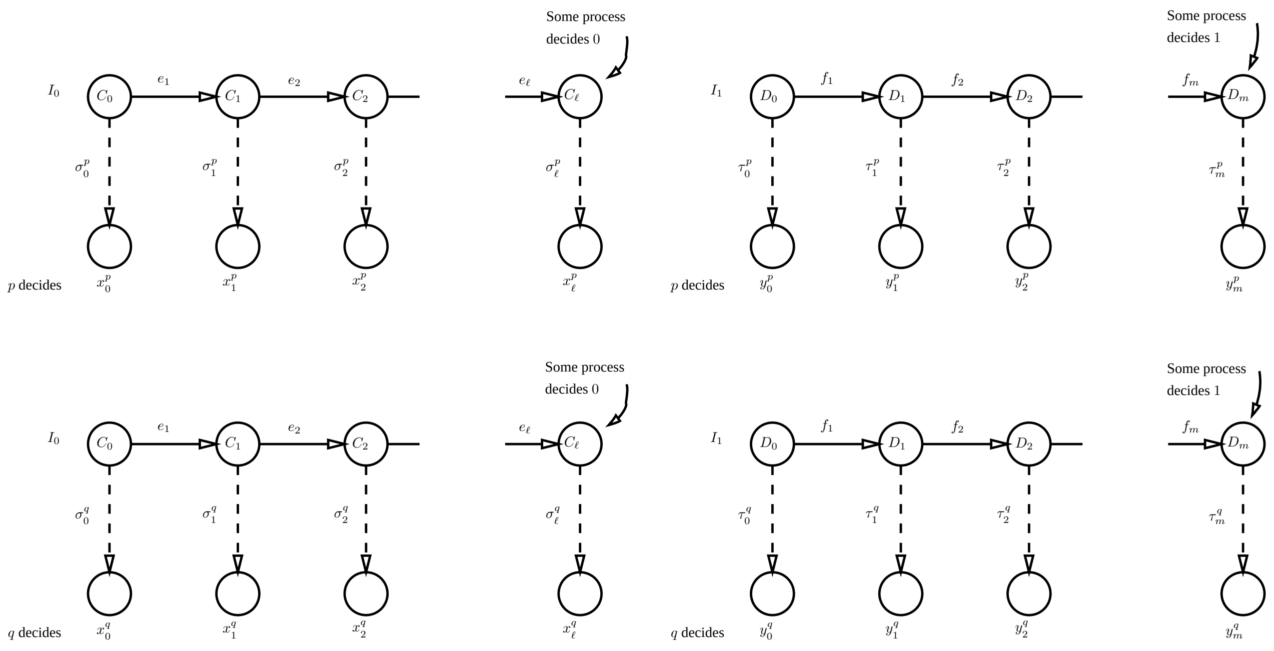
More precisely, let $S_0 = e_1 e_2 \dots e_\ell$ and $S_1 = f_1 f_2 \dots f_m$ (where the e_is and f_js are steps). Let $C_0 = I_0$ and $C_i = e_i(C_{i-1})$ for $1 \le i \le \ell$; similarly, $D_0 = I_1$ and $D_j = f_j(D_{j-1})$ for $1 \le j \le m$. Thus, $\mathcal{C} = \{C_0, \dots, C_\ell, D_0, \dots, D_m\}$.

Let Q_p and Q_q be the values of Σ-*output*$_p$ and Σ-*output*$_q$ at any two times. We must prove that $Q_p \cap Q_q \neq \emptyset$. Suppose, for contradiction, that this is not the case.

Consider the iteration of the loop in lines 26-32 at the end of which p set Σ-*output*$_p$ to Q_p. In that iteration, for each C_i, $0 \le i \le \ell$, p determined a schedule σ_i^p such that p decides some value, denoted x_i^p, in $\sigma_i^p(C_i)$; and, for each D_j, $0 \le j \le m$, p determined a schedule τ_j^p such that p decides some value, denoted y_j^p, in $\tau_j^p(C_j)$. Note that Q_p is the set of processes that take steps in the schedules σ_i^p, $0 \le i \le \ell$, and τ_j^p, $0 \le j \le m$.

Consider now the iteration of the loop in lines 26-32 at the end of which q set Σ-*output*$_q$ to Q_q. We define σ_i^q, x_i^q, τ_j^q, and y_j^q, in an analogous manner. (See Figure 3.)

Claim 7.1 *For all i, $0 \le i \le \ell$, $x_i^p = x_i^q$; and for all j, $0 \le j \le m$, $y_j^p = y_j^q$.*



Some process decides 0
Some process decides 1
I_0
C_0
C_1
C_2
C_ℓ
e_1
e_2
e_ℓ
I_1
D_0
D_1
D_2
D_m
f_1
f_2
f_m
σ_0^p σ_1^p σ_2^p σ_ℓ^p
τ_0^p τ_1^p τ_2^p τ_m^p
p decides x_0^p x_1^p x_2^p x_ℓ^p
p decides y_0^p y_1^p y_2^p y_m^p
σ_0^q σ_1^q σ_2^q σ_ℓ^q
τ_0^q τ_1^q τ_2^q τ_m^q
q decides x_0^q x_1^q x_2^q x_ℓ^q
q decides y_0^q y_1^q y_2^q y_m^q

Figure 3: Illustration of proof of Lemma 7

PROOF SKETCH. Since Q_p and Q_q are disjoint, for each i, $0 \le i \le \ell$, the sets of processes that take steps in σ_i^p and σ_i^q are disjoint. Thus, by Corollary 5 (applied with $I = I_0$, $S = e_1 e_2 \dots e_i$, $S_1 = \sigma_i^p$ and $S_2 = \sigma_i^q$), $x_i^p = x_i^q$. The proof that $y_j^p = y_j^q$ is analogous. □

By Claim 7.1, we can now defi ne$x_i = x_i^p = x_i^q$ and $y_j = y_j^p = y_j^q$.

Claim 7.2 *For all* i, $0 \le i < \ell$, $x_{i+1} = x_i$*; and for all* j, $0 \le j < m$, $y_{j+1} = y_j$.

PROOF SKETCH. Recall that $C_{i+1} = e_{i+1}(C_i)$. Since Q_p and Q_q are disjoint, the sets of processes that take steps in σ_i^p and σ_i^q are disjoint. Thus, the process that takes step e_{i+1} does not take a step in at least one of σ_i^p or σ_i^q. Without loss of generality, assume that it does not take a step in σ_i^p. Let $\sigma = e_{i+1} \cdot \sigma_{i+1}^q$. Note that σ is applicable to C_i. Moreover, the sets of processes that take steps in σ_i^p and σ are disjoint. Thus, by Corollary 5 (applied with $I = I_0$, $S = e_1 e_2 \dots e_i$, $S_1 = \sigma_i^p$ and $S_2 = \sigma$), $x_i = x_{i+1}$. The proof that $y_j = y_{j+1}$ is analogous. □

Claim 7.3 $x_0 = 0$ *and* $y_0 = 1$.

PROOF SKETCH. Recall that some process decides 0 in $C_\ell = S_0(I_0)$. Therefore, p decides 0 in $\sigma_\ell^p(C_\ell)$, and so $x_\ell = 0$. By Claim 7.2 and a trivial induction, $x_i = 0$ for all i, $0 \le i \le \ell$. In particular, $x_0 = 0$. The proof that $y_0 = 1$ is analogous. □

Let r be the process such that I_0 and I_1 differ only in the initial value of r. Since Q_p and Q_q are disjoint, the sets of processes that take steps in σ_0^p and τ_0^q are disjoint. So r does not take a step in at least one of σ_0^p and τ_0^q. Without loss of generality, assume that r does not take a step in σ_0^p. Thus, σ_0^p is also applicable to I_1. By Corollary 5 (applied with $I = I_1$, S being the empty schedule, $S_1 = \sigma_0^p$ and $S_2 = \tau_0^q$), $x_0 = y_0$. This contradicts Claim 7.3, and completes the proof of Lemma 7. □

Theorem 8 *For all environments $\mathcal{E}$, if failure detector $\mathcal{D}$ can be used to solve QC in $\mathcal{E}$, then the algorithm in Figure 2 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 2 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 2, 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 [1] (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 [1]. Since processes reach line 22, by Lemma 3, a critical index exists in the limit forest Υ. By following the proof of [1], 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 [1] refers to a *bi*valent node, we now refer to a *multi*valent one, and whenever [1] refers to 0-valent versus 1-valent nodes, we refer here to u-valent and v-valent nodes where $u, v \in \{0, 1, \text{Q}\}$ 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 6 and 7.

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 1 and 8, we have:

```
Code for each process $p$:

Procedure VOTE$(v)$: { $v$ is Yes or No }
  send $v$ to all
  wait until [(for each process $q$ in $\Pi$, received $q$'s vote) or $\mathcal{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

Corollary 9 *For all environments $\mathcal{E}$, Ψ is the weakest failure detector to solve QC in $\mathcal{E}$.*

5 The weakest failure detector to solve NBAC

5.1 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:

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.

5.2 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 10 *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. 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 [3, 9]. 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. □

```
Code for each process $p$:

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

Figure 5: Transforming NBAC into QC

5.3 The weakest failure detector to solve NBAC

Theorem 11 *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. 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) $\mathcal{D}$ 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 10(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 10(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 $\mathcal{E}$. $\square$

From Corollary 9 and Theorem 11, we immediately have:

Corollary 12 *For all environments $\mathcal{E}$, $(\Psi, \mathcal{FS})$ is the weakest failure detector to solve NBAC in $\mathcal{E}$.*

6 Final remarks

In environments where a majority of processes are correct it is easy to implement the quorum failure detector Σ: Each process periodically sends "join-quorum" messages, and takes as its present quorum any majority of processes that respond to that message [4]. Therefore, in such environments Ψ is equivalent to a simpler failure detector, one which outputs just Ω where Ψ outputs (Ω, Σ).

Our definitions of QC and NBAC do not allow a process to quit or abort because of a future failure. We could have defined these problems in a way that allows such behaviour, as in fact is the case in some specifications of NBAC in the literature. Our results also hold with these definitions, provided we make a corresponding change to the definitions of the failure detectors $\mathcal{FS}$ and Ψ: they are now allowed to output **red** in executions with failures even before a failure has occurred.

References

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

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

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

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

[5] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, V. Hadzilacos, P. Kouznetsov, and S. Toueg. The weakest failure detectors to solve certain fundamental problems in distributed computing. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing (PODC'04)*, pages 338–346, July 2004.

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

[7] E. Fromentin, M. Raynal, and F. Tronel. On classes of problems in asynchronous distributed systems with process crashes. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems (ICDCS'99)*, pages 470–477, 1999.

[8] 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.

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

[10] 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.

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

[12] A. Mostefaoui, M. Raynal, and F. Tronel. From Binary Consensus to Multivalued Consensus in asynchronous message-passing systems. *Information Processing Letters*, 73(5–6):207–212, March 2000.

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