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

Rachid Guerraoui Vassos Hadzilacos Petr Kuznetsov Sam Toueg

July 2, 2012

Abstract

We define *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 [23]. 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) [20]. One way to circumvent such impossibility results is through the use of *unreliable failure detectors* [10]. Intuitively, a failure detector provides each process with some (possibly incomplete and inaccurate) information about failures, e.g., a list of processes currently suspected to have crashed.

Failure detectors can be compared by "reduction": Intuitively, failure detector $\mathcal{D}$ is *weaker than* failure detector $\mathcal{D}'$ if there is an algorithm that transforms $\mathcal{D}'$ into $\mathcal{D}$. Note that if $\mathcal{D}$ is weaker than $\mathcal{D}'$, any problem that can be solved with $\mathcal{D}$ can also be solved with $\mathcal{D}'$. For any problem P, a natural question is to determine the *weakest* failure detector to solve P, i.e., to determine the failure detector $\mathcal{D}^*$ such that (a) there is an algorithm that uses $\mathcal{D}^*$ to solve P, and (b) $\mathcal{D}^*$ is weaker than any failure detector $\mathcal{D}$ that can be used to solve P. Finding the weakest failure detector to solve a problem P amounts to determining the minimum amount of information about failures that is necessary to solve P. It also provides important intuition about systems in which P is solvable: P is solvable in *any* system where the weakest failure detector for P can be implemented. Such a system may be defined in terms of partial synchrony assumptions, or in terms of other assumptions, e.g., the number and timing of failures.

Chandra *et al.* [9] determined the weakest failure detector to solve consensus in systems with a majority of correct processes, while Delporte *et al.* [15] generalized this result to all systems, regardless of the number of correct processes.

As with consensus, failure detectors can be used to solve NBAC [24, 22]. 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 define a natural variation of consensus, called *quittable consensus* (QC);

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

(c) we determine the weakest failure detector to solve QC; 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 occurred *or* someone voted No; in contrast, in QC the decision to quit is allowed only if a failure 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. If there is a correct process, then there is a time after which Ω outputs the id of the same correct process at all correct processes [9].

- 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 any correct process consists of only correct processes [15].

- The *failure signal* failure detector $\mathcal{FS}$ outputs **green** or **red** at each process. As long as there are no failures, $\mathcal{FS}$ must output **green** at every process; once a failure occurs, and only if it does, $\mathcal{FS}$ must eventually output **red** permanently at every correct process [12, 24].

We 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) [15],[1] and to $\mathcal{FS}$. Intuitively, 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 correct processes or like the failure detector $\mathcal{FS}$ at all correct 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. Furthermore, Ψ can switch from $\perp$ to $\mathcal{FS}$ only if a failure occurred. This result has an intuitively appealing interpretation: To solve QC, a failure detector must eventually either truthfully inform all the correct processes that a failure occurred, in which case they can decide Q, or it must be powerful enough to allow processes to solve consensus on their proposed values. This matches the behaviour of Ψ.

We also prove that NBAC is in some sense equivalent to QC modulo the failure detector $\mathcal{FS}$. Intuitively, (a) given $\mathcal{FS}$, any QC algorithm can be converted to an algorithm for NBAC, and (b) any algorithm for NBAC can be converted to an algorithm for QC, and can also be used to implement $\mathcal{FS}$.

Using this equivalence we 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.

[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 $\mathcal{D}'$.

Related work. The model of asynchronous systems augmented with failure detectors was introduced in [10] as one way to circumvent the impossibility result of [20]. Chandra *et. al* proved that Ω is the weakest failure detector to solve consensus in systems with a majority of correct processes [9]. Delporte *et. al* generalized this result to prove that (Ω, Σ) is the weakest failure detector to solve consensus in any system, regardless of the number of correct processes [15]. Failure detectors have been used to capture the minimum information about failures that is necessary to solve other basic problems in distributed computing, such as set agreement [18, 37], mutual exclusion [17], boosting obstruction-freedom to wait-freedom [25], implementing an atomic register in message-passing systems [15], and implementing uniform reliable broadcast [5, 29] in systems with lossy communication links. It is worth noting that the result that Ω is the weakest failure detector for solving consensus led to the discovery of several consensus algorithms for other important models, in particular, for several weak models of *partial synchrony* (e.g., see [2, 1, 33, 30, 6]). This was done by implementing Ω in such systems, and then combining this implementation with any algorithm that solves consensus using Ω, thus exploiting the modularity of the failure detector approach.

The NBAC problem has been studied extensively in the context of transaction processing [23, 36]. Its relation to consensus was first explored in [28]. Charron-Bost and Toueg [12] and Guerraoui [24] 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 NBAC was explored and settled in special settings. Fromentin *et al.* [22] determine that to solve NBAC between *every* pair of processes in the system, one needs a *perfect failure detector* [10]. Guerraoui and Kouznetsov [26] determine the weakest failure detector for NBAC for a restricted class of failure detectors. From results of [12] and [24] 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, remained open until our results appeared, in preliminary form, in [16].[3]

Quittable consensus is closely related to the *detectable broadcast* problem introduced and studied by Fitzi *et al.* in a different setting, namely, synchronous systems with arbitrary process failures [21]. Roughly speaking, in the detectable broadcast problem, correct processes either agree on the broadcast value or, if failures occur, they may agree to "reject" the broadcast; furthermore, if any correct process rejects the broadcast, then the "adversary gets no information about the sender's input" — a privacy requirement that is relevant in the case of arbitrary failures.

Quittable consensus is also related to the *abortable consensus* problem that Chen defined in the context of message-passing systems with probabilistic message delays and losses [13]. Roughly speaking, in abortable consensus some processes are allowed to abort when the behavior of the system degenerates (e.g., there are many process failures or message delays or losses). In contrast to quittable consensus, however, abortable consensus does not require agreement: some processes may decide the same value while others abort.

Other weakenings of the consensus problem were studied in the context of obstruction-free object implementations in shared-memory systems. For example, Attiya *et al.* defined objects that may reply with a special value "pause" or "fail" to some processes if there is step contention [7]. Similarly, Aguilera *et al.* defined abortable objects that may return "abort" in the event of interval contention [3]. In both works, when the object is consensus, agreement in not required: some processes may "pause" or "abort" while others agree on the same value. Furthermore, pausing, failing, or aborting is allowed when there is contention, not failures. In contrast, in quittable consensus the decision to quit must be agreed by all processes and is allowed only in the case of failures.

Roadmap. The rest of the paper is organized as follows: In Section 2 we review the model of computation. Sections 3 and 4 contain the precise specifications of the failure detectors used in this paper, and of QC and

[2] An exception is the case where at most one process may fail. In this case, any algorithm that solves NBAC can be converted into one that solves consensus, but the reverse does not hold.

[3] That paper contained additional results by Delporte, Fauconnier and Guerraoui, which have since appeared in full form in [15].

NBAC, the two problems we consider. In Section 5 we show that QC and NBAC are closely related. In that section we also identify the weakest failure detector Ψ to solve QC and prove that $(\Psi, \mathcal{FS})$ is the weakest failure detector to solve NBAC. In Section 6 we show that Ψ is sufficient to solve QC. Sections 7 and 8 contain the proof that Ψ is necessary to solve QC. We conclude with some final remarks in Section 9.

2 The model

Our model of asynchronous computation is the one described in [9], which augments the model of Fischer, Lynch, and Paterson [20] with failure detectors. Henceforth, we assume a discrete global clock to which the processes do not have access. The range of this clock's ticks is $\mathbb{N}$.

2.1 Systems

We consider distributed message-passing systems with a set of $n \geq 2$ processes $\Pi = \{1, 2, \ldots, n\}$. Processes execute steps of computation asynchronously, i.e., there is no bound on the delay between steps. (Section 2.4 describes what a process does in each step.) Each pair of processes are connected by a reliable link. The links transmit messages with finite but unbounded delay. They are modeled as a set M, called the *message buffer*, that contains triples of the form $(p, data, q)$ indicating that p has sent the message $data$ to q, and q has not yet received it. We assume that each message sent by a process p to a process q is unique; this can be guaranteed by having the sender include a counter with each message.

2.2 Failures, failure patterns and environments

We consider crash failures only: processes fail only by halting prematurely. A *failure pattern* is a function $F : \mathbb{N} \to 2^{\Pi}$, where $F(t)$ is the set of processes that have crashed through time t. Since processes never recover from crashes, $F(t) \subseteq F(t+1)$. Let $faulty(F) = \bigcup_{t \in \mathbb{N}} F(t)$ be the set of faulty processes in a failure pattern F; and $correct(F) = \Pi - faulty(F)$ be the set of correct processes in F. When the failure pattern F is clear from the context, we say that process p is *correct* if $p \in correct(F)$, and p is *faulty* if $p \in faulty(F)$.

An *environment*, denoted $\mathcal{E}$, is a set of failure patterns. Intuitively, an environment $\mathcal{E}$ describes the number and timing of failures that can occur in the system. Thus, a result that applies to all environments is one that holds regardless of the number and timing of failures. We denote by $\mathcal{E}^*$ the set of *all* failure patterns. Intuitively, in a system with environment $\mathcal{E}^*$ each process may crash, and it may do so any time.

2.3 Failure detectors

A *failure detector history H with range $\mathcal{R}$* describes the behavior of a failure detector during an execution. Formally, it is a function $H : \Pi \times \mathbb{N} \to \mathcal{R}$, where $H(p, t)$ is the value output by the failure detector module of process p at time t.

A *failure detector $\mathcal{D}$ with range $\mathcal{R}$* is a function that maps every failure pattern F to a nonempty set of failure detector histories with range $\mathcal{R}$. $\mathcal{D}(F)$ is the set of all possible failure detector histories that may be output by $\mathcal{D}$ in a failure pattern F. Typically we specify a failure detector by stating the properties that its histories satisfy.

Given two failure detectors $\mathcal{D}$ and $\mathcal{D}'$, we denote by $(\mathcal{D}, \mathcal{D}')$ the failure detector whose output is an ordered pair in which the first element corresponds to an output of $\mathcal{D}$, and the second element corresponds to an output of $\mathcal{D}'$. More precisely, if $\mathcal{R}$ and $\mathcal{R}'$ are the ranges of $\mathcal{D}$ and $\mathcal{D}'$, respectively, then the range of $(\mathcal{D}, \mathcal{D}')$ is $\mathcal{R} \times \mathcal{R}'$. For all failure patterns F,

$$(\mathcal{D}, \mathcal{D}')(F) = \big\{ H'' \mid \exists H \in \mathcal{D}(F),\ \exists H' \in \mathcal{D}'(F),\ \forall p \in \Pi,\ \forall t \in \mathbb{N} :\ H''(p,t) = \big(H(p,t), H'(p,t)\big) \big\}$$

2.4 Algorithms

An algorithm $\mathcal{A}$ is modeled as a collection of n deterministic automata. There is an automaton $\mathcal{A}(p)$ for each process p. Computation proceeds in steps of these automata. In each step, a process p atomically

- receives a single message m from the message buffer M, or the empty message λ;
- queries its local failure detector module and receives a value d;
- changes its state; and
- sends a message to every process.

The state transition and the messages that p sends are all uniquely determined by the automaton $\mathcal{A}(p)$, the state of p at the beginning of the step, the received message m, and the failure detector value d. Formally, a step is a tuple $e = (p, m, d, \mathcal{A})$, where p is the process taking step e, m is the message received by p during e, d is the failure detector value seen by p in e, and $\mathcal{A}$ is the algorithm being executed.

The message received in a step is nondeterministically selected from $M \cup \{\lambda\}$. This reflects the asynchrony of the communication channels: a process p may receive the empty message despite the existence of unreceived messages addressed to p.

We assume that each process p has a read-only *input variable*, denoted IN_p, and a write-once *output variable*, denoted OUT_p. Technically, these variables are components of the states of the automaton $\mathcal{A}(p)$. In each initial state of $\mathcal{A}(p)$, the input variable IN_p has some value in $\{0,1\}^*$, and the output variable OUT_p is initialized to the special value $\perp \notin \{0,1\}^*$ (to denote that it was not yet written by p).

2.5 Configurations

A *configuration* of an algorithm $\mathcal{A}$ is a pair (s, M), where s is a function that maps each process p to a state of $\mathcal{A}(p)$, and M is the message buffer. Recall that M is a set of triples $(p, data, q)$, where p sent $data$ to q, which has not yet received it. An *initial* configuration of algorithm $\mathcal{A}$ is a pair (s, M), where $M = \emptyset$ and $s(p)$ is an initial state of the automaton $\mathcal{A}(p)$.

A step $(p, m, d, \mathcal{A})$ is *applicable* to a configuration $C = (s, M)$ if and only if $m \in M \cup \{\lambda\}$. If e is a step applicable to configuration C, $e(C)$ denotes the configuration that results when we apply e to C. This is uniquely determined by the automaton $\mathcal{A}(p)$ of the process p that takes step e.

2.6 Schedules

A *schedule* S of an algorithm $\mathcal{A}$ is a finite or infinite sequence of steps of $\mathcal{A}$. We denote by *participants*(S) the set of processes that take at least one step in schedule S. The ith step in schedule S is denoted by $S[i]$. A schedule S is *applicable* to a configuration C if S is the empty schedule, or $S[1]$ is applicable to C, $S[2]$ is applicable to $S[1](C)$, etc. If S is finite and is applicable to C, $S(C)$ denotes the configuration that results when we apply schedule S to configuration C.

Let S be a schedule applicable to an initial configuration I of an algorithm $\mathcal{A}$, and let i, j be positive integers such that $i, j \leq |S|$. We say that step i *causally precedes* step j in S with respect to I if and only if one of the following holds [32]:

- $S[i]$ and $S[j]$ are steps of the same process and $i < j$;
- $S[i]$ is a step in which a message m is sent and $S[j]$ is a step in which m is received, i.e., step $S[i]$ applied to configuration $S[1] \cdots S[i-1](I)$ results in the sending of m and $S[j] = (-, m, -, \mathcal{A})$;[4] or

[4]The symbol "$-$" in a field of a tuple indicates an arbitrary permissible value for that field of the tuple. We use this convention throughout the paper.

- there is a positive integer $k \leq |S|$ such that step i causally precedes step k, and step k causally precedes step j in S with respect to I.

Note that if $S[i]$ and $S[j]$ are steps involving the sending and receipt of the same message m, then $i < j$ (because if $j < i$, then $S[j]$ would be receiving m before m is sent in $S[i]$, contradicting the fact that S is applicable to I). This implies:

Observation 1 *If step i causally precedes step j in S with respect to I then $i < j$.*

2.7 Runs

A *run of algorithm $\mathcal{A}$ using failure detector $\mathcal{D}$ in environment $\mathcal{E}$* is a tuple $R = (F, H, I, S, T)$ where F is a failure pattern in $\mathcal{E}$, H is a failure detector history in $\mathcal{D}(F)$, I is an initial configuration of $\mathcal{A}$, S is a schedule of $\mathcal{A}$, and T is a list of times in $\mathbb{N}$ (informally, $T[i]$ is the time when step $S[i]$ is taken) such that the following hold:

(1) S is applicable to I.

(2) S and T are both finite sequences of the same length, or are both infinite sequences.

(3) For all positive integers $i \leq |S|$, if $S[i] = (p, -, d, \mathcal{A})$, then $p \notin F(T[i])$ and $d = H(p, T[i])$.

(4) For all positive integers $i < j \leq |S|$, $T[i] \leq T[j]$.

(5) For all positive integers $i, j \leq |S|$, if step i causally precedes step j in S with respect to I then $T[i] < T[j]$.

Property (3) states that a process does not take steps after crashing, and that the failure detector value seen in a step is the one dictated by the failure detector history H. Property (4) states that the sequence of times when processes take steps in a schedule is nondecreasing, and property (5) states that these times respect causal precedence.

A run whose schedule is finite (respectively, infinite) is called a finite (respectively, infinite) run. An *admissible run* of algorithm $\mathcal{A}$ using failure detector $\mathcal{D}$ in environment $\mathcal{E}$ is an infinite run $R = (F, H, I, S, T)$ of $\mathcal{A}$ using $\mathcal{D}$ in $\mathcal{E}$ with two additional properties:

(6) Every correct process takes an infinite number of steps in S.

(7) Each message sent to a correct process is eventually received. More precisely, for every finite prefix S' of S, and every $q \in \mathit{correct}(F)$, if the message buffer in configuration $S'(I)$ contains a message $m = (-, -, q)$, then for some $i \in \mathbb{N}$, $S[i] = (q, m, -, \mathcal{A})$.

The input and output of a run $R = (F, H, I, S, T)$ of an algorithm $\mathcal{A}$ are defined as follows. The *input of R*, denoted $\mathcal{I}(R)$, is the vector $(\mathcal{I}_1, \ldots, \mathcal{I}_n)$ where $\mathcal{I}_p$ is the value of the input variable IN_p in the initial configuration I of R. The *output of R*, denoted $\mathcal{O}(R)$, is the vector $(\mathcal{O}_1, \ldots, \mathcal{O}_n)$ where $\mathcal{O}_p$ is the pair (v, t) such that p writes v in its output variable OUT_p at time t in run R ($\mathcal{O}_p = \perp$ if p never writes OUT_p in run R).

2.8 Problems

We consider *input/output problems*, i.e., problems where each process has an input value and produces an output value. We can specify such a problem P as a set of triples of the form $(F, \mathcal{I}, \mathcal{O})$: intuitively, $(F, \mathcal{I}, \mathcal{O}) \in P$ if and only if, when the failure pattern is F and the processes' input is $\mathcal{I}$, the processes' output $\mathcal{O}$ is acceptable, i.e., it "satisfies" problem P. More precisely, a problem P is a set of triples $(F, \mathcal{I}, \mathcal{O})$ where F is a failure pattern, $\mathcal{I}$ is a vector $(\mathcal{I}_1, \ldots, \mathcal{I}_n)$ of input values (each one in $\{0,1\}^*$), and $\mathcal{O}$ is a vector $(\mathcal{O}_1, \ldots, \mathcal{O}_n)$ where each $\mathcal{O}_p$ is either $\perp$ or a pair (v, t) such that v is an output value in $\{0,1\}^*$ and t is a time in $\mathbb{N}$. We say that $\mathcal{I}$ *is an input vector of P* if $(F, \mathcal{I}, \mathcal{O}) \in P$ for some F and $\mathcal{O}$.

2.9 Solving a problem

Let P be a problem, $\mathcal{A}$ an algorithm, $\mathcal{D}$ a failure detector, and $\mathcal{E}$ an environment. We say that:

- A run $R = (F, H, I, S, T)$ of $\mathcal{A}$ using $\mathcal{D}$ in $\mathcal{E}$ *satisfies* P if and only if $(F, \mathcal{I}(R), \mathcal{O}(R)) \in P$, or there is no $\mathcal{O}$ such that $(F, \mathcal{I}(R), \mathcal{O}) \in P$.[5]
- *$\mathcal{A}$ solves P using $\mathcal{D}$ in $\mathcal{E}$* if and only if
  (a) every admissible run R of $\mathcal{A}$ using $\mathcal{D}$ in $\mathcal{E}$ satisfies P, and
  (b) for every input vector $\mathcal{I} = (\mathcal{I}_1, \ldots, \mathcal{I}_n)$ of P, there is an initial configuration I of $\mathcal{A}$ with this input (i.e., in configuration I we have $IN_p = \mathcal{I}_p$ for every process p).
- *$\mathcal{D}$ can be used to solve P in $\mathcal{E}$* (or simply *P can be solved with $\mathcal{D}$ in $\mathcal{E}$*) if and only if there is an algorithm that solves P using $\mathcal{D}$ in $\mathcal{E}$.

2.10 Comparing failure detectors

Intuitively, a failure detector $\mathcal{D}'$ is weaker than a failure detector $\mathcal{D}$ if processes can use $\mathcal{D}$ to emulate $\mathcal{D}'$; so if they can solve a problem with $\mathcal{D}'$, they can also solve it with $\mathcal{D}$. We say that processes can use $\mathcal{D}$ to emulate $\mathcal{D}'$ in an environment $\mathcal{E}$ if there is an algorithm that transforms $\mathcal{D}$ to $\mathcal{D}'$ in $\mathcal{E}$ as follows. The transformation algorithm, denoted $\mathcal{T}_{\mathcal{D}\to\mathcal{D}'}$, uses $\mathcal{D}$ to maintain a variable $\mathcal{D}'\text{-}output_p$ at every process p; $\mathcal{D}'\text{-}output_p$ functions as the output of the emulated failure detector $\mathcal{D}'$ at p. For each admissible run R of $\mathcal{T}_{\mathcal{D}\to\mathcal{D}'}$, let O_R be the history of all the $\mathcal{D}'\text{-}output$ variables in R; i.e., $O_R(p, t)$ is the value of $\mathcal{D}'\text{-}output_p$ at time t in R. Algorithm *$\mathcal{T}_{\mathcal{D}\to\mathcal{D}'}$ transforms $\mathcal{D}$ to $\mathcal{D}'$ in environment $\mathcal{E}$* if and only if for every admissible run $R = (F, H, I, S, T)$ of $\mathcal{T}_{\mathcal{D}\to\mathcal{D}'}$ using $\mathcal{D}$ in $\mathcal{E}$, $O_R \in \mathcal{D}'(F)$.

We say that *$\mathcal{D}'$ is weaker than $\mathcal{D}$ in $\mathcal{E}$* if there is an algorithm $\mathcal{T}_{\mathcal{D}\to\mathcal{D}'}$ that transforms $\mathcal{D}$ to $\mathcal{D}'$ in $\mathcal{E}$. It is easy to see that if $\mathcal{D}'$ is weaker than $\mathcal{D}$ in $\mathcal{E}$, then every problem that can be solved with $\mathcal{D}'$ in $\mathcal{E}$ can also be solved with $\mathcal{D}$ in $\mathcal{E}$. We say that two failure detectors are *equivalent in $\mathcal{E}$* if each is weaker than the other in $\mathcal{E}$.

2.11 Weakest failure detector

A failure detector $\mathcal{D}^*$ is the *weakest failure detector to solve problem P in environment $\mathcal{E}$* if and only if:

Sufficiency. $\mathcal{D}^*$ can be used to solve P in $\mathcal{E}$.

Necessity. For any failure detector $\mathcal{D}$, if $\mathcal{D}$ can be used to solve P in $\mathcal{E}$ then $\mathcal{D}^*$ is weaker than $\mathcal{D}$ in $\mathcal{E}$.

Note that there may be several distinct failure detectors that are the weakest to solve a problem P. It is easy to see, however, that they are all equivalent: If $\mathcal{D}$ and $\mathcal{D}'$ are two failures detectors that are weakest to solve the same problem P, $\mathcal{D}'$ can be used to solve P (by sufficiency of $\mathcal{D}'$) and so $\mathcal{D}$ is weaker than $\mathcal{D}'$ (by necessity of $\mathcal{D}$). Symmetrically, $\mathcal{D}'$ is weaker than $\mathcal{D}$, and so $\mathcal{D}$ and $\mathcal{D}'$ are equivalent. For this reason, we speak of *the* weakest, rather than *a* weakest failure detector to solve P.

3 The failure detectors used in this paper

We now define the failure detectors Ω, Σ, $\mathcal{FS}$, and Ψ that we informally described in Section 1.

[5] Intuitively, this means that when the failure pattern is F and the input is $\mathcal{I}(R)$ the problem P does not care what the output is.

- At each process, the *leader failure detector* Ω outputs the id of a process; furthermore, if a correct process exists, then there is a time after which Ω outputs the id of the same correct process at every correct process. Formally:

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

$$\Omega(F) = \big\{H \mid \mathit{correct}(F) \neq \emptyset \;\Rightarrow \\ \big(\exists q \in \mathit{correct}(F),\, \forall p \in \mathit{correct}(F),\, \exists t \in \mathbb{N},\, \forall t' \geq t :\; H(p,t') = q\big)\big\}$$

- 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 any correct process consists of only correct processes. Formally:

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

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

- 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; once a failure occurs, and only if it does, $\mathcal{FS}$ eventually outputs **red** permanently at every correct process. Formally:

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

$$\mathcal{FS}(F) = \big\{H \mid \forall p \in \Pi,\, \forall t \in \mathbb{N} :\; \big(H(p,t) = \textbf{red} \Rightarrow F(t) \neq \emptyset\big) \wedge \\ \big(\mathit{faulty}(F) \neq \emptyset \Rightarrow \forall p \in \mathit{correct}(F),\, \exists t \in \mathbb{N},\, \forall t' \geq t :\; H(p,t') = \textbf{red}\big)\big\}$$

- The failure detector Ψ initially outputs $\perp$ and may eventually switch to behaving permanently like (Ω, Σ) or like $\mathcal{FS}$. This switch has the following properties: (a) it must occur at all correct processes; (b) it must be consistent (it is not possible for Ψ to behave like (Ω, Σ) at a process p at time t and like $\mathcal{FS}$ at process p' at time t'); and (c) Ψ may start behaving like $\mathcal{FS}$ at a process only if a failure occurred. Formally:

  The range of Ψ is $\{\perp\} \cup \{\textbf{green}, \textbf{red}\} \cup \{(p, P) \mid p \in \Pi \wedge P \in 2^{\Pi}\}$. For every failure pattern F,

$$\Psi(F) = \Big\{H \;\Big|\; \exists H' \in (\Omega,\Sigma)(F) \cup \mathcal{FS}(F),\, \forall p \in \Pi : \big(p \in \mathit{faulty}(F) \;\wedge\; \forall t \in \mathbb{N} :\; H(p,t) = \perp\big) \vee \\ \exists t \in \mathbb{N} : \big(\forall t' < t :\; H(p,t') = \perp \;\wedge\; \forall t' \geq t :\; H(p,t') = H'(p,t') \;\wedge\; H' \in \mathcal{FS}(F) \Rightarrow F(t) \neq \emptyset\big)\Big\}$$

4 Specification of Consensus, QC and NBAC

In this section we define the three problems considered in this paper, namely, consensus, quittable consensus, and non-blocking atomic commit. Each one of these problems is an input/output problem that can be formally specified as explained Section 2.8; our definitions are more informal here.

4.1 Consensus and QC

In the consensus problem, each process p has some input value $v \in V = \{0,1\}^*$ (we say that p *proposes* v) and must write some output value $v \in V$ (we say that p *decides* v) such that the following properties hold:

Termination: Every correct process eventually decides some value.

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

Validity: If a process decides v then some process proposes v.

Quittable consensus is similar to consensus, except that processes are allowed to decide a special value Q $\notin V$ (which means "quit") if a failure occurred. More precisely, QC has the same requirements as consensus, except that the above validity property is replaced by the following one:

Validity: Each process may only decide some value in $V \cup \{\text{Q}\}$, where Q $\notin V$. Moreover,

(i) If a process decides $v \neq$ Q then some process proposes v.
(ii) If a process decides Q then a failure occurred.[6]

A straightforward proof by indistinguishable scenarios leads to:

Observation 2 *Let $\mathcal{A}$ be an algorithm that solves consensus, or quittable consensus, using a failure detector $\mathcal{D}$ in an environment $\mathcal{E}$. In every run R of $\mathcal{A}$ using $\mathcal{D}$ in $\mathcal{E}$, if a process decides some value $v \in V$ at some time t, then there is a process that proposes v and takes at least one step in R by time t.*

4.2 Non-blocking atomic commit

In the NBAC problem, each process p has some input value $v \in \{\text{Yes}, \text{No}\}$ (we say that p *votes* v) and must write some output value $v \in \{\text{Commit}, \text{Abort}\}$ (we say that p *decides* v) such that the following properties hold:

Termination: Every correct process eventually decides some value.

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

Validity: Each process may only decide Commit or Abort. Moreover,

(i) If a process decides Commit then all processes vote Yes.
(ii) If a process decides Abort then either some process vote No or a failure occurred.

As with Observation 2, an obvious proof by indistinguishable scenarios leads to:

Observation 3 *Let $\mathcal{A}$ be an algorithm that solves NBAC using a failure detector $\mathcal{D}$ in an environment $\mathcal{E}$. In every run R of $\mathcal{A}$ using $\mathcal{D}$ in $\mathcal{E}$, if a process decides Commit at some time t, then all processes vote Yes and take at least one step in R by time t.*

4.3 Using a consensus, QC, or NBAC algorithm inside another algorithm

An algorithm $\mathcal{A}$ can use an algorithm $\mathcal{A}_c$ that solves consensus by emulating $\mathcal{A}_c$ as follows. In the pseudocode of $\mathcal{A}$, a process p can execute a statement of the form "$d :=$ PROPOSE(v)" to start an emulated execution of $\mathcal{A}_c$ with input (i.e., proposal) value v. This statement first sets up the initial state of $\mathcal{A}_c(p)$ to correspond to the input value v, and then starts to execute the steps of $\mathcal{A}_c(p)$ with this initial state. If and when p decides in this emulated execution of $\mathcal{A}_c(p)$, the decision value is assigned to the variable d, and p resumes executing the steps of $\mathcal{A}$. Concurrently, p continues to execute the steps of $\mathcal{A}_c(p)$ until $\mathcal{A}_c(p)$ halts. Similar comments apply for an algorithm $\mathcal{A}$ that uses a QC or NBAC algorithm.

[6] Throughout this paper, when we say "if event x occurs then event y occurr*ed*" we mean, more precisely, "if event x occurs at time t then event y occurred by time $t' \leq t$".

CODE THAT PROCESS p EXECUTES TO VOTE v, WHERE v IS YES OR NO, FOR NBAC:

```
1  send v to all        /* send vote v to all processes */
2  wait until [(∀q ∈ Π, received q's vote) or FS_p = red]
3  if the votes of all processes are received and are Yes then
4     myproposal := 1
5  else    /* some vote was No or there was a failure */
6     myproposal := 0
7  mydecision := PROPOSE(myproposal)    /*execute A_qc using D to solve an instance of QC */
8  if mydecision = 1 then
9     decide Commit
10 else    /* mydecision = 0 or Q */
11    decide Abort
```

Figure 1: Algorithm $\mathcal{A}_{nbac}$ uses $(\mathcal{D}, \mathcal{FS})$ to solve NBAC

5 Relating NBAC and QC and their weakest failure detectors

NBAC is in some sense equivalent to the combination of QC and failure detector $\mathcal{FS}$. More precisely:

Theorem 4 *In every environment $\mathcal{E}$:*

(1) If a failure detector $\mathcal{D}$ can be used to solve QC in $\mathcal{E}$, then $(\mathcal{D}, \mathcal{FS})$ can be used to solve NBAC in $\mathcal{E}$.

(2) If a failure detector $\mathcal{D}'$ can be used to solve NBAC in $\mathcal{E}$, then

- *$\mathcal{D}'$ can be used to solve QC in $\mathcal{E}$, and*
- *$\mathcal{D}'$ can be transformed to $\mathcal{FS}$ in $\mathcal{E}$.*

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

(1) Suppose that failure detector $\mathcal{D}$ can be used to solve QC in $\mathcal{E}$, i.e., there is an algorithm $\mathcal{A}_{qc}$ that uses $\mathcal{D}$ to solve QC in $\mathcal{E}$. Figure 1 shows an algorithm $\mathcal{A}_{nbac}$ that uses $(\mathcal{D}, \mathcal{FS})$ to solve NBAC in $\mathcal{E}$. $\mathcal{A}_{nbac}$ works as follows. Each process p sends its vote to every process, and waits until it receives a vote from every process or the $\mathcal{FS}$ component of $(\mathcal{D}, \mathcal{FS})$ indicates that a failure occurred. If p receives a vote from every process and all the votes are Yes, it sets *myproposal* to 1; otherwise some vote was No or a failure occurred, and p sets *myproposal* to 0. Then, in line 7, p participates in an execution of the QC algorithm $\mathcal{A}_{qc}$ (which uses the $\mathcal{D}$ component of $(\mathcal{D}, \mathcal{FS})$) where p's initial value is set to *myproposal* (as explained in Section 4.3). If p decides 1 in this execution of $\mathcal{A}_{qc}$, then p decides Commit for NBAC; if p decides 0 or Q in this execution of $\mathcal{A}_{qc}$, then p decides Abort for NBAC.

We now prove that, in every admissible run, algorithm $\mathcal{A}_{nbac}$ satisfies all the properties of NBAC.

Termination. This property holds trivially if all processes are faulty, so assume that some process is correct. Let p be any correct process. Since every correct process executes the algorithm in Figure 1, if p never receives the vote of some process q, then q must have crashed. In that case, by the specification of the failure detector $\mathcal{FS}$, eventually $\mathcal{FS}_p =$ **red** forever. Thus, p eventually completes the wait statement on line 2. Therefore, eventually every correct process starts the execution of $\mathcal{A}_{qc}$ in line 7 (as explained in Section 4.3). By the termination property of QC, every correct process completes its execution of line 7, and eventually decides.

Uniform agreement. Follows from the uniform agreement property of QC.

CODE THAT PROCESS p EXECUTES TO PROPOSE v FOR QC:

```
1  send v to all          /* send QC proposal v to all processes */
2  d := VOTE(Yes)         /* execute B_nbac using D' to solve an instance of NBAC */
3  if d = Abort then
4    decide Q
5  else
6    wait until [∀q ∈ Π, received q's proposal]
7    decide smallest proposal received
```

Figure 2: Algorithm $\mathcal{B}_{qc}$ uses $\mathcal{D}'$ to solve QC

Validity. Let p be any process.

(a) Suppose p decides Commit (line 9). Then p decided 1 in its execution of $\mathcal{A}_{qc}$ on line 7. By part (i) of validity of QC, some process q proposes 1 on line 7 (i.e., q starts its emulation of $\mathcal{A}_{qc}$ with initial value 1 on that line). Before doing so, q must have received Yes votes from all processes (see lines 3–4). So, if a process decides Commit, all processes vote Yes.

(b) Suppose p decides Abort (line 11). Then p decided 0 or Q in its execution of $\mathcal{A}_{qc}$ on line 7. If p decided Q, then by part (ii) of validity of QC, a failure occurred. If p decided 0 then, by Observation 2, there is a process q that proposes 0 and took a step in the emulation of $\mathcal{A}_{qc}$ on line 7. Before doing so, q must have received a vote No from some process or found that $\mathcal{FS}_q =$ **red** (see lines 3, 5, 6). The latter can happen only if a failure occurred. We conclude that if a process decides Abort, some process votes No or a failure occurred.

(2) Suppose that failure detector $\mathcal{D}'$ can be used to solve NBAC in $\mathcal{E}$, i.e., there is an algorithm $\mathcal{B}_{nbac}$ that uses $\mathcal{D}'$ to solve NBAC in $\mathcal{E}$.

(a) *$\mathcal{D}'$ can be used to solve QC in $\mathcal{E}$.* An algorithm $\mathcal{B}_{qc}$ that uses $\mathcal{D}'$ to solve QC in $\mathcal{E}$ is shown in Figure 2. Informally, it works as follows. Each process p first sends its QC proposal, some value $v \in V$, to all processes. Then, in line 2, p participates in an execution of the NBAC algorithm $\mathcal{B}_{nbac}$ (which uses $\mathcal{D}'$) with initial value Yes, i.e., an execution of NBAC where p votes Yes. If this execution returns Abort, p decides Q; if it returns Commit, p waits to receive a proposal from every process and decides the smallest of these proposals.

We now prove that, in every admissible run, algorithm $\mathcal{B}_{qc}$ satisfies all the properties of QC.

Termination. This property holds trivially if all processes are faulty, so assume that some process is correct. Every correct process executes the statement $d :=$ VOTE(Yes) on line 2 to participate in an execution of NBAC (as explained in Section 4.3). By the termination property of NBAC, all correct processes eventually decide, i.e., they complete the execution of this statement. If line 2 sets d to Abort, then p decides Q on line 4. Otherwise, it must set d to Commit. By Observation 3, every process q votes Yes and took a step in the emulation of $\mathcal{B}_{nbac}$ on line 2. Before doing so, q sent its QC proposal to all processes (on line 1). So, p eventually receives a proposal from every process, completes the wait statement on line 6, and decides some value for QC on line 7.

Uniform agreement. By the uniform agreement property of NBAC, all the processes that set their variable d in line 2, set it to the same value. Thus, all the processes that decide some value (for QC) do so on line 4, or they all decide on line 7. In the first case they all decide Q, and in the second case they all decide the smallest proposal of all processes in Π. So no two processes decide differently.

Validity. Let p be any process. If p decides $v \neq$ Q (on line 7), then v is the smallest proposal that p received, and thus some process proposes v. Now suppose p decides Q (on line 4). Thus, p's execution

CODE FOR EACH PROCESS p:

1 $\mathcal{FS}$-$output_p \leftarrow$ **green**
2 **repeat**
3 $d :=$ VOTE(Yes) /* *execute $\mathcal{B}_{nbac}$ using $\mathcal{D}'$ to solve an instance of NBAC* */
4 **until** $d =$ Abort
5 $\mathcal{FS}$-$output_p \leftarrow$ **red**

Figure 3: Transforming any $\mathcal{D}'$ that can be used to solve NBAC into $\mathcal{FS}$

of the statement $d :=$ VOTE(Yes) on line 2 sets d to Abort. By part (ii) of validity of NBAC, either some process votes No or a failure occurred. But no process votes No. Thus, a failure occurred.

(b) *$\mathcal{D}'$ can be transformed to $\mathcal{FS}$ in $\mathcal{E}$* (this result can be found in [12, 24]). The transformation algorithm is shown in Figure 3. At each process p, the variable $\mathcal{FS}$-$output_p$ (which emulates the output of $\mathcal{FS}$ at p) is initially **green**. Processes emulate consecutive and independent executions of $\mathcal{B}_{nbac}$ using $\mathcal{D}'$ to solve consecutive instances of NBAC while voting Yes in every instance. If and when a process p decides Abort in an instance of NBAC, then p sets $\mathcal{FS}$-$output_p$ to **red**, and never changes $\mathcal{FS}$-$output_p$ thereafter.

From the agreement and termination properties of NBAC, it is easy to show by induction that the following holds (the proof is omitted here):

Claim 4.1 *Either all correct processes execute the **repeat-until** loop of lines 2–4 infinitely many times, or they all exit this loop and execute line 5.*

Suppose no failures occur. Since (a) all the processes are correct, and (b) they all vote Yes in every instance of NBAC executed in line 2, then by part (ii) of validity of NBAC no process ever decides Abort on line 2, and so $\mathcal{FS}$-$output$ remains **green** at all processes, forever.

Suppose a failure occurs. Then there is a process p that crashes and a k such that p does not participate (i.e., does not take any step) in the k-th instance of NBAC. We claim that every correct process eventually sets $\mathcal{FS}$-$output$ to **red** on line 5. Suppose, for contradiction, that some correct process never sets $\mathcal{FS}$-$output$ to **red** on line 5. By Claim 4.1, it must be that all correct processes execute the **repeat-until** loop of lines 2–4 infinitely many times, and so they participate in the k-th instance of NBAC. Since p takes no steps in this instance, then, by Observation 3, correct processes cannot decide Commit in that instance. So, by Claim 4.1, they all decide Abort in the k-th instance of NBAC, and then they they exit the **repeat-until** loop — a contradiction. Thus, every correct process eventually sets $\mathcal{FS}$-$output$ to **red** on line 5.

Finally, suppose some process p sets $\mathcal{FS}$-$output$ to **red** on line 5 at some time t. Then p must have decided Abort in an instance of NBAC on line 2 by time t. By part (ii) of validity of NBAC, some process votes No in that instance of NBAC, or a failure occurred by time t. Since no process ever votes No, a failure occurred by time t.

□

The close relationship between NBAC and QC established in Theorem 4 allows us to relate the weakest failure detectors to solve these problems.

Theorem 5 *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) if a failure detector $\mathcal{D}'$ can be used to solve QC in $\mathcal{E}$ then $\mathcal{D}'$ can be transformed to $\mathcal{D}$ in $\mathcal{E}$.

To prove that $(\mathcal{D}, \mathcal{FS})$ is the weakest failure detector to solve NBAC in $\mathcal{E}$, we now show two facts:

(1) *$(\mathcal{D}, \mathcal{FS})$ can be used to solve NBAC in $\mathcal{E}$.* This follows directly from (i) and Theorem 4(1).

(2) *If a failure detector $\mathcal{D}'$ can be used to solve NBAC in $\mathcal{E}$, then $\mathcal{D}'$ can be transformed to $(\mathcal{D}, \mathcal{FS})$.*

To see this, let $\mathcal{D}'$ be a failure detector that can be used to solve NBAC in $\mathcal{E}$. By Theorem 4(2):

- $\mathcal{D}'$ can be used to solve QC in $\mathcal{E}$. So by (ii) above, $\mathcal{D}'$ can be transformed to $\mathcal{D}$ in $\mathcal{E}$.
- $\mathcal{D}'$ can be transformed to $\mathcal{FS}$ in $\mathcal{E}$.

Thus, $\mathcal{D}'$ can be transformed to $(\mathcal{D}, \mathcal{FS})$ in $\mathcal{E}$.

□

The weakest failure detectors to solve QC and NBAC. In Section 6 we show that Ψ can be used to solve QC in every environment (Theorem 8). In Section 8 we show that, in every environment $\mathcal{E}$, any failure detector that can be used to solve QC in $\mathcal{E}$ can be transformed to Ψ in $\mathcal{E}$ (Theorem 30). From these two facts, we have:

Corollary 6 *For every environment $\mathcal{E}$, Ψ is the weakest failure detector to solve QC in $\mathcal{E}$.*

Theorem 5 relates the weakest failure detector to solve QC to the weakest failure detector to solve NBAC. So by Corollary 6 and Theorem 5, we have:

Corollary 7 *For every environment $\mathcal{E}$, $(\Psi, \mathcal{FS})$ is the weakest failure detector to solve NBAC in $\mathcal{E}$.*

6 Ψ is sufficient to solve QC

Recall that, intuitively, Ψ behaves as follows (see Section 3 for a precise definition). 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 correct processes or like the failure detector $\mathcal{FS}$ at all correct processes. The switch from $\perp$ to (Ω, Σ) or $\mathcal{FS}$ is consistent at all processes, and a switch from $\perp$ to $\mathcal{FS}$ can happen only if a failure occurred.

In Figure 4 we show an algorithm that uses Ψ to solve QC in any environment. This algorithm uses an algorithm $\mathcal{A}_c$ that solves *consensus* using (Ω, Σ) in any environment. Delporte *et al.* have shown that such an algorithm exists [15].

Informally, the algorithm in Figure 4 works as follows. To propose some value $v \in V$ for QC, a process p waits until Ψ_p (p's module of failure detector Ψ) outputs a value different from $\perp$. At that time, either Ψ_p starts behaving like $\mathcal{FS}$ or it starts behaving like (Ω, Σ). If Ψ_p behaves like $\mathcal{FS}$ (which happens only if a failure occurred), then p decides Q. If, on the other hand, Ψ_p behaves like (Ω, Σ), then p participates in an execution of the consensus algorithm $\mathcal{A}_c$ where it proposes v (it does so by executing the $d :=$ PROPOSE(v) statement on line 5, as explained in Section 4.3). Process p adopts the decision value of this execution of $\mathcal{A}_c$, as its decision for QC.

Theorem 8 *For every environment $\mathcal{E}$, the algorithm in Figure 4 uses Ψ to solve QC in $\mathcal{E}$.*

CODE THAT PROCESS p EXECUTES TO PROPOSE v FOR QC:

```
1  wait until [Ψ_p ≠ ⊥]
2  if Ψ_p ∈ {green, red} then    /* Ψ behaves like FS and, thus, a failure occurred */
3     decide Q
4  else    /* henceforth Ψ behaves like (Ω, Σ) */
5     d := PROPOSE(v)    /* execute A_c using Ψ to solve an instance of consensus */
6     decide d
```

Figure 4: Using Ψ to solve QC.

PROOF. Consider any admissible run of the algorithm in Figure 4. We will prove that this run satisfies the properties of QC.

Termination. This property holds trivially if all processes are faulty, so assume that some process is correct. Let p be any correct process. By the specification of Ψ, there is a time after which Ψ_p has values in the range of either $\mathcal{FS}$ or (Ω, Σ); thus, p completes the wait statement on line 1. If eventually Ψ_p has values in the range of $\mathcal{FS}$ then p decides Q (see lines 2-3). Otherwise, Ψ never outputs values in the range of $\mathcal{FS}$ at any process, and there is a time after which Ψ outputs only values in the range of (Ω, Σ) at all correct processes. Thus, eventually every correct process executes the statement $d :=$ PROPOSE(v) for some v on line 5, i.e., every correct process participates in an execution of $\mathcal{A}_c$. By the termination property of consensus, this execution terminates, and so p decides d on line 6.

Uniform agreement. By the specification of Ψ, it is not possible that Ψ outputs a value in the range of $\mathcal{FS}$ at one process and a value in the range of (Ω, Σ) at another. From this observation and the fact that $\mathcal{A}_c$ satisfies uniform agreement (for consensus), it follows that the algorithm in Figure 4 satisfies uniform agreement (for QC).

Validity. Let p be any process.
(i) Suppose p decides some value $v \neq$ Q for QC (on line 6). Thus, p also decides v in its execution of the consensus algorithm $\mathcal{A}_c$ on line 5. From Observation 2, at least one process q starts to execute the statement $d :=$ PROPOSE(v) on line 5. Therefore, process q executes the algorithm in Figure 4 with QC proposal v. So, if a process decides $v \neq$ Q (for QC), some process proposes v (for QC).
(ii) Suppose p decides Q for QC at some time t (on line 3) . Thus, $\Psi_p \in \{\textbf{green}, \textbf{red}\}$ by time t. By the specification of Ψ, a failure occurred by time t. □

7 Some auxiliary results

In this section we present some technical lemmas used in our proof that in every environment Ψ is necessary to solve QC, presented in Section 8. The lemmas in Section 7.2 appeared in [9], sometimes in different form.

7.1 Mergeable runs

Several proofs in distributed computing employ a technique known as the "partition argument". At the heart of this technique is the ability to combine two different runs R_0 and R_1 of an algorithm $\mathcal{A}$ that involve *disjoint* sets of processes P_0 and P_1, respectively, into a single run of $\mathcal{A}$ in which the processes in P_0 behave as in R_0 and the processes in P_1 behave as in R_1. We now formalize this, and prove that in our model it is possible to combine such "mergeable" runs in this manner.

Let $R_0 = (F, H, I, \hat{S} \cdot S_0, \hat{T}_0 \cdot T_0)$ and $R_1 = (F, H, I, \hat{S} \cdot S_1, \hat{T}_1 \cdot T_1)$ be two finite runs of an algorithm $\mathcal{A}$ using failure detector $\mathcal{D}$ in some environment $\mathcal{E}$, such that $|\hat{T}_0| = |\hat{T}_1| = |\hat{S}|$ and *participants*$(S_0) \cap$ *participants*$(S_1) = \emptyset$. Note that the schedules of these two runs start with the same prefix $\hat{S}$, while their

continuations S_0 and S_1 involve disjoint sets of processes. A *merging* of two such runs is a tuple $R = (F, H, I, \hat{S} \cdot S, \hat{T} \cdot T)$ where (a) $\hat{T}$ is whichever of $\hat{T}_0$ or $\hat{T}_1$ has the smaller last element (either one, if both have the same last element); (b) T is the sequence consisting of the times in T_0 and T_1 in nondecreasing order, and (c) S is the sequence consisting of the steps in S_0 and S_1 merged in the same order as the elements of T_0 and T_1 were merged into T. For example, suppose that $S_0 = a_1, a_2, a_3$, $T_0 = 3, 5, 7$; and $S_1 = b_1, b_2, b_3, b_4$, $T_1 = 2, 4, 5, 6$. Then $T = 2, 3, 4, 5, 5, 6, 7$, and the two possibilities for S are $b_1, a_1, b_2, b_3, a_2, b_4, a_3$ or $b_1, a_1, b_2, a_2, b_3, b_4, a_3$. More formally, the requirements for $R = (F, H, I, \hat{S} \cdot S, \hat{T} \cdot T)$ to be a merging of R_0 and R_1 are:

- $|S| = |S_0| + |S_1|$ and $|T| = |T_0| + |T_1|$;
- $\hat{T} = \hat{T}_b$ for some $b \in \{0, 1\}$ such that the last element of T_b is less than or equal to the last element of $T_{\bar{b}}$;
- T is nondecreasing;
- for each $b \in \{0, 1\}$ and each $i \in \{1, 2, \ldots, |S_b|\}$ there is a $j \in \{1, 2, \ldots, |S|\}$ such that $S_b[i] = S[j]$ and $T_b[i] = T[j]$; and
- for each $j \in \{1, 2, \ldots, |S|\}$ there is a $b \in \{0, 1\}$ and an $i \in \{1, 2, \ldots, |S_b|\}$ such that $S[j] = S_b[i]$ and $T[j] = T_b[j]$.

Lemma 9 *Let $R_0 = (F, H, I, \hat{S} \cdot S_0, \hat{T}_0 \cdot T_0)$ and $R_1 = (F, H, I, \hat{S} \cdot S_1, \hat{T}_1 \cdot T_1)$ be two finite runs of an algorithm $\mathcal{A}$ using failure detector $\mathcal{D}$ in some environment $\mathcal{E}$, such that $|\hat{T}_0| = |\hat{T}_1| = |\hat{S}|$ and participants$(S_0) \cap$ participants$(S_1) = \emptyset$. Let $R = (F, H, I, \hat{S} \cdot S, \hat{T} \cdot T)$ be a merging of R_0 and R_1. Then*

(a) R is also a run of $\mathcal{A}$ using $\mathcal{D}$ in $\mathcal{E}$.

(b) For each $b \in \{0, 1\}$ and each process $p \in$ participants$(\hat{S} \cdot S_b)$, the state of p is the same in $\hat{S} \cdot S(I)$ as in $\hat{S} \cdot S_b(I)$.

The proof of Lemma 9 is straightforward though somewhat tedious; it is given in Appendix A.

7.2 DAGs and simulations

To complete the proof that, for any environment $\mathcal{E}$, Ψ is the weakest failure detector to solve QC in $\mathcal{E}$, it remains to show that any failure detector that can be used to solve QC in $\mathcal{E}$ can be transformed to Ψ in $\mathcal{E}$. In this section we review a technique for proving statements of this type. The technique was introduced by Chandra et al., who used it to prove that any failure detector that can be used to solve consensus can be transformed to Ω [9]. We will use it in this paper to prove that any failure detector that can be used to solve QC can be transformed to Ψ (see Section 8).

Suppose we want to prove that $\mathcal{D}^*$ is the weakest failure detector to solve some problem P in some environment $\mathcal{E}$. Let $\mathcal{D}$ be any failure detector that can be used to solve P in $\mathcal{E}$, i.e., there is an algorithm $\mathcal{A}$ that uses $\mathcal{D}$ to solve P in $\mathcal{E}$. We need to show that $\mathcal{D}$ can be transformed to $\mathcal{D}^*$. The proof technique of [9] shows how to use $\mathcal{D}$ and $\mathcal{A}$ to emulate $\mathcal{D}^*$ in $\mathcal{E}$. This emulation consists of two interacting components: the communication component and the computation component. In the communication component, each process continuously "samples" its local module of $\mathcal{D}$ and exchanges messages with other processes to construct an ever-growing directed acyclic graph (DAG) of failure detector samples of $\mathcal{D}$. In the computation component, p uses this DAG to simulate schedules of the algorithm $\mathcal{A}$ (which uses $\mathcal{D}$ to solve P). Based on these simulated schedules, p simulates the output of the failure detector $\mathcal{D}^*$ that we want to emulate.

We now explain in more detail how each process builds its DAG of failure detector samples and how it uses this DAG to simulate schedules of $\mathcal{A}$.

CODE FOR EACH PROCESS p:

1 **initialize**
2 $k_p \leftarrow 0$
3 $G_p \leftarrow$ empty graph

4 **loop**
5 receive a message m
6 $d_p \leftarrow \mathcal{D}_p$
7 **if** $m \neq \lambda$ **then** $G_p \leftarrow G_p \cup m$
8 $k_p \leftarrow k_p + 1$
9 $v_p \leftarrow (p, d_p, k_p)$
10 add node v_p to G_p and an edge from every other node in G_p to v_p
11 send G_p to every process

Figure 5: Algorithm $\mathcal{A}_{\text{DAG}}$ builds DAGs of failure detector samples of $\mathcal{D}$

7.2.1 Building DAGs of failure detector samples

The DAG-building algorithm, denoted $\mathcal{A}_{\text{DAG}}$, is shown in Figure 5. In our algorithm descriptions, which we give in pseudocode, we use the following conventions. Variables of process p are subscripted with p. If $\mathcal{D}$ is a failure detector, then $\mathcal{D}_p$ denotes the function call by which p can access its local module of $\mathcal{D}$; this call returns the current value of p's local module of $\mathcal{D}$. The pseudocode of each process begins with an **initialize** clause, which defines the process' state in the initial configuration. (Variables whose values are not explicitly set in this clause, can be assigned arbitrary values in the initial configuration.)

In $\mathcal{A}_{\text{DAG}}$, each process p maintains a DAG of failure detector samples of $\mathcal{D}$ in the variable G_p. Each node of this DAG is of the form (q, d, k); such a triple indicates that process q obtained value d when it queried its failure detector module $\mathcal{D}_q$ for the kth time. (The third component is included to ensure that distinct samplings of the failure detector result in distinct nodes.) We call such triples *samples*; a sample $(q, -, -)$ is said to be *of* or *taken by* process q. We use the terms "node (of the DAG)" and "sample" interchangeably.

Process p periodically performs the following actions:

(a) it receives a message, which is either a DAG previously sent to p by another process, or the empty message (line 5);

(b) it queries its local failure detector module $\mathcal{D}_p$, receiving a value that it stores in variable d_p (line 6);

(c) it updates its DAG G_p by first adding to it the DAG that it received in (a), and then adding to it a new node with the failure detector value it got in (b), as well as edges from all other nodes to the new node (lines 7-10); and

(d) it sends the updated G_p to all processes (line 11).

Note that this sequence of actions (receiving a message, querying the local failure detector module, changing local state, and sending messages to other processes) corresponds exactly to the sequence of actions taken in a single step in our model. Thus, each iteration of the loop in Figure 5 is executed as a single step.

We now present some properties of the DAGs of samples computed by algorithm $\mathcal{A}_{\text{DAG}}$. In the following, we consider an arbitrary admissible run $R = (F, H, I, S, T)$ of $\mathcal{A}_{\text{DAG}}$ using failure detector $\mathcal{D}$ in some arbitrary environment $\mathcal{E}$. We use the following notation throughout this section: In the context of a given run of an

algorithm, the value of variable x_p at time t is denoted x_p^t; if p takes a step at time t, then x_p^t is the value of x_p *after* that step.

We start with some simple observations, in each of which p is an arbitrary process. Since p never removes any nodes or edges from G_p, the DAG contained in this variable is monotonically nondecreasing. That is,

Observation 10 *For all $t, t' \in \mathbb{N}$, if $t \leq t'$ then G_p^t is a subgraph of $G_p^{t'}$.*

We define the *limit DAG of a process* p to be $G_p^\infty = \cup_{t \in \mathbb{N}} G_p^t$.

In the same step that a process updates its DAG (line 10), it also sends the new DAG to all processes (line 11); thus each correct process will eventually receive that DAG and will incorporate it into its own. Thus,

Observation 11 *For every correct process p, every process q, and every time $t \in \mathbb{N}$, G_q^t is a subgraph of G_p^∞.*

From this it follows immediately that

Observation 12 *If p and q are correct processes then $G_p^\infty = G_q^\infty$.*

Since k_p is incremented in each iteration of p's loop, when p takes sample $(p, -, k)$, it has already taken samples $(p, -, k')$ for all $k' < k$; and, at that time, it adds edges from all such nodes to $(p, -, k)$. Thus,

Observation 13 *If $v = (p, -, k)$ and $v' = (p, -, k')$ are nodes of G_p^∞ and $k \geq k'$, then v is a descendant of v' in G_p^∞.*

Let $v = (q, d, k)$ be any node of G_p^∞. It is obvious from the code of $\mathcal{A}_{\text{DAG}}$ that process q received d from its failure detector module in its kth step. Let $\tau(v)$ to be the time when q takes this step. More precisely, if $S[i]$ is the kth step of q in S, then $\tau(v) = T[i]$. (Recall that S is the schedule and T is the sequence of times of the run R of $\mathcal{A}_{\text{DAG}}$ that we are considering.) From property (3) of runs, we have:

Observation 14 *If $v = (q, d, k)$ is a node of G_p^∞, then $q \notin F(\tau(v))$ and $d = H(q, \tau(v))$.*

From the algorithm $\mathcal{A}_{\text{DAG}}$, it is clear that if (u, v) is an edge of the limit DAG G_p^∞, then the step in which sample u was taken causally precedes the step in which sample v was taken in schedule S with respect to I (the initial configuration of run R). From property (5) of the runs of $\mathcal{A}_{\text{DAG}}$ (see Section 2.7), it follows that $\tau(u) < \tau(v)$. By induction we can generalize this observation from single edges to finite or infinite paths of G_p^∞:

Observation 15 *If $g = v_0, v_1, \ldots$ is a finite or infinite path in G_p^∞, then the sequence of times $\tau(v_0), \tau(v_1), \ldots$ is strictly increasing.*

Let G be any DAG; if v is a node of G, then $G|v$ is the subgraph of G induced by the descendants of v in G; otherwise, $G|v$ is the empty graph. Informally, the next lemma states that any finite path in process p's limit DAG eventually appears permanently in p's DAG.

Lemma 16 *Let p be a process and v be a node of G_p^∞. For each finite path g in $G_p^\infty|v$, there is a time t such that, for all $t' \geq t$, $g \in G_p^{t'}|v$.*

PROOF. In $G_p^\infty|v$, let g be any finite path, g' be a finite path from v to the first node of g, and h be the path consisting of g' followed by g. Since $G_p^\infty = \cup_{t \in \mathbb{N}} G_p^t$, it is clear that for each edge e of h there is a time $t(e)$ such that e is in $G_p^{t(e)}$. Let $t = \max\{t(e) : e \text{ is an edge of } h\}$. By Observation 10, every edge e of h (and hence v and the entire path g) is in $G_p^{t'}$ for all $t' \geq t$. Since g is in $G_p^\infty|v$, every node in g is a descendant of v. Thus, g is in $G_p^{t'}|v$, for all $t' \geq t$. □

Since faulty processes eventually crash and cease to take steps, from a certain point on only correct processes take samples. This is the basic intuition underlying the next lemma.

Lemma 17 *For every correct process p, there is a sample v^* of p in G_p^∞ such that $G_p^\infty|v^*$ contains only samples of correct processes. Furthermore,*

(a) There is a time after which any node v in variable v_p (line 9) is a descendant of v^* in G_p^∞.

(b) For any descendant v of v^* in G_p^∞ and any $t \in \mathbb{N}$, $G_p^t|v$ contains only samples of correct processes.

PROOF. Since p is correct, it takes infinitely many steps. Let t^* be the first time that p takes a step after all faulty processes have crashed, and let v^* be the sample that p takes in that step. Consider any node v of $G_p^\infty|v^*$. Since v is a descendant of v^* in G_p^∞, by Observation 15, $\tau(v) \geq \tau(v^*) = t^*$. Since all faulty processes have crashed by time t^*, the process that takes sample v (at time $\tau(v) \geq t^*$) must be correct. So, $G_p^\infty|v^*$ contains only samples of correct processes.

(a) Let $v^* = (p, -, k^*)$. Since k_p increases in each iteration of p's loop, eventually k_p has values that are more than k^*. Therefore, eventually only nodes whose third entry is more than k^* are assigned to v_p. By Observation 13 all these nodes are descendants of v^* in G_p^∞.

(b) Consider any descendant v of v^* in G_p^∞ and any time $t \in \mathbb{N}$. Clearly, $G_p^t|v$ is a subgraph of $G_p^\infty|v$, and $G_p^\infty|v$ is a subgraph of $G_p^\infty|v^*$. Since $G_p^\infty|v^*$ contains only samples of correct processes, so does its subgraph $G_p^t|v$. □

Since correct processes keep taking samples and exchanging their DAGs forever, every correct process' limit DAG has an infinite path with infinitely many samples of each correct process. This observation is formalized by Lemma 19. To prove it, it is convenient to prove the following lemma first.

Lemma 18 *Suppose p is a correct process and let G be a subgraph of G_p^t for some time t. For every correct process q, there is a time t' such that $G_p^{t'}$ contains a sample w of q and an edge from every node of G to w.*

PROOF. Let s be the first step that p takes after time t. By Observation 10, G is still in p's DAG just before this step. There are two cases:

$p = q$. In step s, p adds to its DAG a new sample $w = (p, -, -)$, and edges from every other node in its DAG (in particular, from every node in G) to w. Thus, when this step is completed, say at time t', $G_p^{t'}$ has the desired properties.

$p \neq q$. In step s, p sends to all processes a DAG that contains G. Now consider the step in which q receives that DAG. In that step, q first incorporates the DAG it receives, which contains G, into its own DAG. Then q adds to its DAG a new sample $w = (q, -, -)$, and edges from every other node in its DAG (in particular, from every node in G) to w. Finally, q sends the resulting DAG to all processes. Consider the step in which p receives that DAG. When it does so, p incorporates the DAG it receives into its own DAG. Thus, when this step is completed, say at time t', $G_p^{t'}$ has the desired properties. □

Lemma 19 *If p is a correct process and v is a node of G_p^∞, then G_p^∞ has an infinite path that starts with v and contains infinitely many samples of each correct process.*

PROOF. Since v is a node of G_p^∞, there is a time t_0 such that v is in $G_p^{t_0}$. By repeated application of Lemma 18, there is an infinite sequence of times $t_0, t_1, \ldots$ and an infinite sequence of paths $g^0, g^1, \ldots$ such that for all $i \in \mathbb{N}$, (a) g^i is in $G_p^{t_i}$ and starts with v, (b) g^i is a prefix of g^{i+1}, and (c) each correct process has at least i samples in g^i.

Let g^∞ be the "limit" of sequence $g^0, g^1, \ldots$ That is, g^∞ is the infinite path which, up to length $|g^i|$, is identical to g^i. (This is well-defined because of (b).) It is now easy to see that g^∞ is a path in G_p^∞ that starts with v and contains infinitely many samples of each correct process. □

7.2.2 Simulating schedules of an algorithm $\mathcal{A}$

In the previous section, we saw how each process p can execute algorithm $\mathcal{A}_{\text{DAG}}$ using a failure detector $\mathcal{D}$ to build an ever-increasing DAG of samples of $\mathcal{D}$ (under the "current" failure pattern F and failure detector history $H \in \mathcal{D}(F)$). We now explain how each process p can use its DAG of samples of $\mathcal{D}$ to simulate schedules of runs of *any* algorithm $\mathcal{A}$ using $\mathcal{D}$ (with failure pattern F and failure detector history $H \in \mathcal{D}(F)$). These are called *simulated schedules* of $\mathcal{A}$. Another way of thinking about these simulated schedules is that they are schedules of runs that could have occurred if processes were running algorithm $\mathcal{A}$ using $\mathcal{D}$, instead of running $\mathcal{A}_{\text{DAG}}$ using $\mathcal{D}$.

Fix an initial configuration I of algorithm $\mathcal{A}$, and a path $g = (p_1, d_1, k_1), (p_2, d_2, k_2), \ldots$ of the DAG contained in G_p at some time t, or of the limit DAG G_p^∞. Our goal is to define the set of simulated schedules determined by path g and initial configuration I. Path g tells us that the following could have happened in an execution of algorithm $\mathcal{A}$ under the current failure pattern F and failure detector history $H \in \mathcal{D}(F)$: process p_1 takes the first step and sees value d_1 from its failure detector module; then process p_2 takes the second step and sees value d_2 from its failure detector module; and so on. This sequence of process ids and failure detector values, along with the initial configuration I, define a *set* of schedules of $\mathcal{A}$, each schedule in this set corresponding to different delays that the messages sent might experience.

More precisely, we say that a schedule S is *compatible* with the path $g = (p_1, d_1, k_1), (p_2, d_2, k_2), \ldots$ if and only if it has the same length as g, and $S = (p_1, m_1, d_1, \mathcal{A}), (p_2, m_2, d_2, \mathcal{A}), \ldots$ for some (possibly null) messages $m_1, m_2, \ldots$ The set of simulated schedules determined by g and initial configuration I is the set of all schedules that are compatible with g and applicable to I.

Let G be any DAG of samples and I be any initial configuration of $\mathcal{A}$. $\mathbf{Sch}(G, I)$ denotes the set of schedules of $\mathcal{A}$ that are compatible with some path in G and are applicable to I. Note that if G is finite then $\mathbf{Sch}(G, I)$ contains a finite number of finite schedules.

We now present some properties of simulated schedules. In the following, we consider an arbitrary admissible run R of $\mathcal{A}_{\text{DAG}}$ using failure detector $\mathcal{D}$ in some arbitrary environment $\mathcal{E}$. Let $F \in \mathcal{E}$ be the failure pattern of this run and $H \in \mathcal{D}(F)$ its failure detector history.

The first lemma justifies the name "simulated schedules"; it states that these schedules really are schedules of runs of algorithm $\mathcal{A}$ using $\mathcal{D}$, with failure pattern F and failure detector history H.

Lemma 20 *Let p be a process, $t \in \mathbb{N} \cup \{\infty\}$, G be a subgraph of G_p^t, and I be an initial configuration of algorithm $\mathcal{A}$. For each schedule $S \in \mathbf{Sch}(G, I)$, there is a list of times T, all at most t, such that $R_\mathcal{A} = (F, H, I, S, T)$ is a run of $\mathcal{A}$ using $\mathcal{D}$ in $\mathcal{E}$.*

PROOF. Let S be any schedule in $\mathbf{Sch}(G, I)$. Thus, S is a schedule of $\mathcal{A}$ that is applicable to I and compatible with some path $g = v_1, v_2, \ldots$ in G. Let $T = \tau(v_1), \tau(v_2), \ldots$ Recall that for each positive integer $i \leq |S|$, $\tau(v_i)$ is the time when sample v_i was taken. A sample can't appear in any DAG until the time it is taken. Since v_i is a node in a subgraph of G_p^t, $\tau(v_i) \leq t$.

We claim that $R_\mathcal{A} = (F, H, I, S, T)$ is a run of $\mathcal{A}$ using $\mathcal{D}$ in $\mathcal{E}$. Since $F \in \mathcal{E}$, $H \in \mathcal{D}(F)$ and I is an initial configuration of $\mathcal{A}$, it suffices to verify that $R_\mathcal{A}$ satisfies properties (1)–(5) of runs. S is applicable to I (property (1)) by definition of $\mathbf{Sch}(G, I)$. S and T have the same length (property (2)) because each of them has the same length as g. The fact that in R no process takes a step after it has crashed, and that the failure detector value in each step is consistent with the history H (property (3)) follows from Observation 14, since S is compatible with path $g = v_1, v_2, \ldots$ and $T = \tau(v_1), \tau(v_2), \ldots$ Observation 15 implies that T is strictly

increasing, and so property (4) is also satisfied. To show property (5), we must prove that if step i causally precedes step j in S with respect to I then $T[i] < T[j]$. This follows from Observation 1 and the fact T is strictly increasing. □

By Lemma 20, every infinite schedule $S^\infty \in \mathbf{Sch}(G_p^\infty, I)$ is a schedule of an infinite run of $\mathcal{A}$ using $\mathcal{D}$ in $\mathcal{E}$. However, S^∞ is not necessarily a schedule of an *admissible* run, i.e., a run where each correct process takes an infinite number of steps (property (6)) and eventually receives every message sent to it (property (7)). The next lemma, however, states that every finite schedule $S \in \mathbf{Sch}(G_p^\infty, I)$ can be extended to *some* infinite schedule $S^\infty \in \mathbf{Sch}(G_p^\infty, I)$ of an *admissible* run of $\mathcal{A}$.

Lemma 21 *Suppose p is a correct process and let I be an initial configuration of $\mathcal{A}$. For any finite schedule $S \in \mathbf{Sch}(G_p^\infty, I)$ there is a schedule $S^\infty \in \mathbf{Sch}(G_p^\infty, I)$ that extends S and a list of times T^∞ such that $R_\mathcal{A} = (F, H, I, S^\infty, T^\infty)$ is an* admissible *run of $\mathcal{A}$ using $\mathcal{D}$ in $\mathcal{E}$. Furthermore, for any node u in G_p^∞, S^∞ can be chosen so that $S^\infty = S \cdot \sigma^\infty$, for a schedule σ^∞ that is compatible with a path in $G_p^\infty|u$.*

PROOF. Let S be any finite schedule in $\mathbf{Sch}(G_p^\infty, I)$ and u be any node in G_p^∞. Thus S is applicable to I and compatible with a finite path g of G_p^∞. By Lemma 18 (applied with $q = p$ and G consisting of the path g and the node u) and the monotonicity of the DAGs (Observation 10), G_p^∞ contains a sample v of p and an edge from every node of g and from u to v. By Lemma 19, G_p^∞ has an infinite path γ that starts with v and contains infinitely many samples of each correct process. Note that $g \cdot \gamma$ is a path in G_p^∞ (because there are edges from every node in g to the first node of γ); and γ is a path in $G_p^\infty|u$ (because there is an edge from u to the first node of γ).

We define an infinite sequence of schedules $\sigma^0, \sigma^1, \ldots$ such that for each $i \in \mathbb{N}$, (a) σ^i has length i, (b) σ^i is compatible with the path consisting of the first i nodes of γ, (c) σ^i is applicable to $S(I)$, and (d) if $i > 0$, σ^{i-1} is a prefix of σ^i. The definition is by induction:

Basis. σ^0 is the empty schedule. It is obvious that this has the required properties.

Induction step. Let i be an arbitrary positive integer, and assume that σ^{i-1} with the required properties has been defined. Let the ith node of γ be $(p, d, -)$. Then $\sigma^i = \sigma^{i-1} \cdot (p, m, d, \mathcal{A})$, where m is the message defined as follows: If the message buffer of configuration $S \cdot \sigma^{i-1}(I)$ has no message to p (i.e., no message of the form $(-, -, p)$), then $m = \lambda$; otherwise, m is the *oldest* message to p in the message buffer of $S \cdot \sigma^{i-1}(I)$ (i.e., there is no message m' to p in the message buffer of $S \cdot \sigma^{i-1}(I)$ and prefix S' of $S \cdot \sigma^{i-1}$ such that the message buffer of $S'(I)$ contains m' but not m). It is straightforward to verify that σ^i has the required properties: length i, compatible with the first i nodes of γ, applicable to $S(I)$, and an extension of σ^{i-1}.

Now define σ^∞ to be the "limit" of the sequence $\sigma^0, \sigma^1, \ldots$ — i.e., the infinite schedule whose prefix of length i is σ^i. (This is well-defined because, for all $i \in \mathbb{N}$, σ^i has length i and is a prefix of σ^{i+1}.) Clearly σ^∞ is compatible with γ and applicable to $S(I)$. Let $S^\infty = S \cdot \sigma^\infty$. We have:

- $S^\infty \in \mathbf{Sch}(G_p^\infty, I)$. This follows from the fact that S^∞ is compatible with path $g \cdot \gamma$ in G_p^∞, and S^∞ is applicable to I (because S is applicable to I, and σ^∞ is applicable to $S(I)$).
- σ^∞ is compatible with path γ in $G_p^\infty|u$.

By Lemma 20, there is a time list T^∞ such that $R_\mathcal{A} = (F, H, I, S^\infty, T^\infty)$ is a run of $\mathcal{A}$ using $\mathcal{D}$ in $\mathcal{E}$. It remains to prove that $R_\mathcal{A}$ is admissible. We first note that each correct process takes infinitely many steps in $R_\mathcal{A}$; this is because S^∞ is compatible with $g \cdot \gamma$ and γ contains infinitely many samples of each correct process. Furthermore, from the way we choose the message received in each step of σ^∞, every message sent to a correct process is eventually received in $R_\mathcal{A}$. So, $R_\mathcal{A}$ has the required properties (6) and (7) of admissible runs. □

The following lemma is an immediate consequence of Observation 11 and the definition of $\mathbf{Sch}(-, -)$:

Lemma 22 *For every correct process p, every process q, every time $t \in \mathbb{N}$, and every initial configuration I of $\mathcal{A}$, **Sch**$(G_q^t, I) \subseteq$ **Sch**(G_p^∞, I).*

The following lemma is an immediate consequence of Lemma 16 and the definition of **Sch**$(-,-)$:

Lemma 23 *Let p be a process and I be an initial configuration of $\mathcal{A}$. For each finite schedule $S \in$ **Sch**(G_p^∞, I), there is a time t such that, for all $t' \geq t$, $S \in$ **Sch**$(G_p^{t'}, I)$.*

8 Ψ is necessary to solve QC

In this section, we show that Ψ is necessary to solve QC. Let $\mathcal{D}$ be any 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 $\mathcal{E}$. We must show that there is an algorithm that transforms $\mathcal{D}$ into Ψ in $\mathcal{E}$. We do so by giving a transformation algorithm that uses $\mathcal{A}$ and $\mathcal{D}$ to emulate the output of Ψ — a failure detector that initially outputs $\perp$ and later behaves either like (Ω, Σ) or like $\mathcal{FS}$. This transformation algorithm, denoted $\mathcal{T}_{\mathcal{D}\to\Psi}$, is shown in Figures 6–7, and is explained below.

8.1 Overview of the transformation

To make the presentation clearer, the code of each process p in algorithm $\mathcal{T}_{\mathcal{D}\to\Psi}$ is given by three concurrent threads.[7] In Thread 1, p builds a DAG of samples of failure detector $\mathcal{D}$ using the algorithm discussed in Section 7.2.1. In Thread 2, p uses its current DAG to determine whether (a) it is legitimate for Ψ to behave like $\mathcal{FS}$ and output **red** permanently (because a failure occurred in the current run) or (b) it is possible to "extract" (Ω, Σ) in the current run. Then p participates in an instance of QC to reach agreement with the other processes on (a) or (b). In Thread 3, p produces the output of failure detector Ψ according to the agreement reached in Thread 2. We now explain $\mathcal{T}_{\mathcal{D}\to\Psi}$ in more detail.

First recall that in the algorithm $\mathcal{A}$ that solves QC, the value that a process p proposes (i.e., its input value) is encoded in the initial state of $\mathcal{A}(p)$. For each $j \in [0..n]$, let I^j be the initial configuration of $\mathcal{A}$ in which every process $q \in [1..j]$ proposes 1 and every process $q \in [j+1..n]$ proposes 0. Thus in any run starting from I^0, all processes propose 0, and starting from I^n they all propose 1.

In $\mathcal{T}_{\mathcal{D}\to\Psi}$, each process p starts by outputting $\perp$ (line 4), and then it executes three concurrent threads:

- In Thread 1, p builds G_p, a DAG of failure detector values seen by processes in the current run.
- In Thread 2, p repeatedly examines $n+1$ sets of simulated schedules of algorithm $\mathcal{A}$, namely **Sch**(G_p, I^0), $\ldots$, **Sch**(G_p, I^n) (line 21), until it finds that for every $j \in [0..n]$, **Sch**(G_p, I^j) contains a schedule S_p^j such that p decides some value x_p^j in $S_p^j(I^j)$ (line 22). If for some $j \in [0..n]$, $x_p^j =$ Q, then a failure occurred (in the failure pattern of the current run), and so p knows that it is legitimate to extract $\mathcal{FS}$ and output **red** in this run. Otherwise for every $j \in [0..n]$, x_p^j is either 0 or 1, and in this case p determines that it is possible to extract (Ω, Σ) in the current run.

  At this point p participates in an instance of QC (by using the algorithm $\mathcal{A}$ and failure detector $\mathcal{D}$) to *agree* with the other processes on whether to extract $\mathcal{FS}$ and output **red**, or to extract (Ω, Σ). Specifically, if p has determined that it is legitimate to output **red** then it proposes 0 in this execution of QC (lines 25–26). Otherwise, p proposes a tuple $(I^i, I^{i+1}, S_p^i, S_p^{i+1})$, where $i \in [0..n-1]$ is such that $x_p^i = 0$ and $x_p^{i+1} = 1$ (lines 27–29). Note that such an index i must exist: Since p reaches line 27, by the condition on line 25 every x_p^j is either 0 or 1; by the validity property of QC, $x_p^0 = 0$ and $x_p^n = 1$; thus, for some $i \in [0..n-1]$, $x_p^i = 0$ and $x_p^{i+1} = 1$.

[7] It is straightforward to write the code of p as the code of a sequential process that can be directly expressed in our model (e.g., p can execute the three concurrent threads in round-robin fashion).

CODE FOR EACH PROCESS p:

1 **initialize**
2 Ω-$output_p \leftarrow p$
3 Σ-$output_p \leftarrow \Pi$
4 Ψ-$output_p \leftarrow \perp$
5 $k_p \leftarrow 0$
6 $G_p \leftarrow$ empty graph
7 *decision*$_p \leftarrow \perp$

8 **cobegin**
9 /* THREAD 1 — BUILD DAG OF $\mathcal{D}$-SAMPLES */
10 **loop**
11 receive a message m
12 $d_p \leftarrow \mathcal{D}_p$
13 **if** $m \neq \lambda$ **then** $G_p \leftarrow G_p \cup m$
14 $k_p \leftarrow k_p + 1$
15 $v_p \leftarrow (p, d_p, k_p)$
16 add node v_p to G_p and an edge from every other node in G_p to v_p
17 send G_p to every process

18 || /* THREAD 2 — CHOOSE BEHAVIOUR OF Ψ */
19 $\forall j \in [0..n]$, $I^j \leftarrow$ initial configuration of $\mathcal{A}$ where the initial state of each $q \in [1..j]$ corresponds to proposal 1 and the initial state of each $q \in [j+1..n]$ corresponds to proposal 0
20 **repeat**
21 $\forall j \in [0..n]$, **Sch**$(G_p, I^j) \leftarrow$ set of schedules of $\mathcal{A}$ compatible with G_p and applicable to I^j
22 **until** $\forall j \in [0..n]$, $\exists S_p^j \in$ **Sch**(G_p, I^j) : p decides in $S_p^j(I^j)$
23 $\forall j \in [0..n-1]$, let $S_p^j \in$ **Sch**(G_p, I^j) and x_p^j be such that p decides x_p^j in $S_p^j(I^j)$
24 /* execute $\mathcal{A}$ using $\mathcal{D}$ to solve an instance of QC */
25 **if** $\exists j \in [0..n]$ such that $x_p^j =$ Q **then**
26 *decision*$_p :=$ PROPOSE(0) /* *propose* 0 *in this instance of* QC */
27 **else**
28 let $i \in [0..n-1]$ be such that $x_p^i = 0$ and $x_p^{i+1} = 1$
29 *decision*$_p :=$ PROPOSE$(I^i, I^{i+1}, S_p^i, S_p^{i+1})$ /* *propose* $(I^i, I^{i+1}, S_p^i, S_p^{i+1})$ *in this instance of* QC */

30 || /* THREAD 3 — OUTPUT VALUE OF Ψ */
31 **wait until** *decision*$_p \neq \perp$
32 **if** *decision*$_p = 0$ or *decision*$_p =$ Q **then** /* *a failure occurred* */
33 Ψ-$output_p \leftarrow$ **red**
34 **else** /* p*'s decision is a tuple with two initial configurations and two schedules* */
35 $(I_0, I_1, S_0, S_1) :=$ *decision*$_p$
36 $u_p \leftarrow v_p$
37 **loop**
38 Ω-$output_p \leftarrow$ *extract-leader*() /* *see Figure* 7 */
39 Σ-$output_p \leftarrow$ *extract-quorum*() /* *see Figure* 7 */
40 Ψ-$output_p \leftarrow (\Omega$-$output_p, \Sigma$-$output_p)$
41 **coend**

Figure 6: Algorithm $\mathcal{T}_{\mathcal{D} \to \Psi}$

CODE FOR EACH PROCESS p:

42 **function** *extract-leader*():
43 determine the current leader ℓ_p of p using G_p as in [9]
44 **return** ℓ_p

45 **function** *extract-quorum*():
46 $\forall b \in \{0,1\}$, $\forall S$ prefix of S_b, **Sch**$(G_p|u_p, S(I_b)) \leftarrow$ set of schedules of $\mathcal{A}$ compatible with $G_p|u_p$ and applicable to $S(I_b)$
47 **if** $\forall b \in \{0,1\}$, $\forall S$ prefix of S_b, $\exists \sigma_S \in$ **Sch**$(G_p|u_p, S(I_b))$: p decides in $S \cdot \sigma_S(I_b)$ **then**
48 /* *return new quorum* */
49 *new-quorum*$_p \leftarrow \bigcup \{$*participants*$(\sigma) | \exists b \in \{0,1\}, \exists S$ prefix of S_b : $(\sigma \in$ **Sch**$(G_p|u_p, S(I_b)) \wedge p$ decides in $S \cdot \sigma(I_b))\}$
50 $u_p \leftarrow v_p$
51 **return** *new-quorum*$_p$
52 **else** /* *return old quorum* */
53 **return** Σ-$output_p$

Figure 7: Functions used by algorithm $\mathcal{T}_{\mathcal{D}\to\Psi}$

- In Thread 3, p computes the output values of Ψ according to the decision of the QC executed in Thread 2. If this decision is 0 or Q, then p stops outputting $\perp$ and outputs **red** from that time on (lines 32–33). If the decision is some tuple (I_0, I_1, S_0, S_1) (line 35), then p stops outputting $\perp$ and starts extracting Ω (line 38) and Σ (line 39), combining these two outputs into the output of Ψ (line 40). Ω is extracted as in [9] (see Section 8.2) and Σ is extracted using novel techniques (see Section 8.3).

Note that in $\mathcal{T}_{\mathcal{D}\to\Psi}$, processes use the algorithm $\mathcal{A}$ for QC in two different ways and for different purposes. First, each process uses its DAG of failure detector samples to *simulate* many schedules 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 participate in a real execution of $\mathcal{A}$ 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 simulating schedules of $\mathcal{A}$ to effect this extraction.

For the remainder of Section 8 we consider an arbitrary admissible run of algorithm $\mathcal{T}_{\mathcal{D}\to\Psi}$ using $\mathcal{D}$ in some arbitrary environment $\mathcal{E}$. Let $F \in \mathcal{E}$ be the failure pattern of this run and $H \in \mathcal{D}(F)$ be its failure detector history.

8.2 Extracting Ω

The specification of Ω requires the following: at each process, Ω outputs the id of a process; furthermore, if a correct process exists, then there is a time after which Ω outputs the id of the same correct process p^* at every correct process. Note that this specification is trivially satisfied in runs where all processes are faulty. So in the rest of Section 8.2, we assume that there is at least one correct process in the run under consideration, i.e, $correct(F) \neq \emptyset$.

If in $\mathcal{T}_{\mathcal{D}\to\Psi}$ processes decide to extract Ω, they do so by executing the algorithm that extracts Ω described in [9], as we now explain. As in [9], processes build a DAG of samples of failure detector $\mathcal{D}$ (Thread 1 in Figure 6). More precisely, processes build a DAG of the failure detector values that they see in $H \in \mathcal{D}(F)$. By Observation 12, the limit DAG of all correct processes is the same. Let G^∞ denote that DAG. (G^∞ is well-defined since a correct process exists.)

Recall that I^j is the initial configuration of $\mathcal{A}$ in which the initial state of each process $q \in [1..j]$ corresponds to proposal 1 and the initial state of each process $q \in [j+1..n]$ corresponds to proposal 0. Furthermore $\mathbf{Sch}(G^\infty, I^i)$ is the set of schedules of $\mathcal{A}$ that is compatible with G^∞ and applicable to I^i. For each $i \in [0..n]$, we organize the set of schedules in $\mathbf{Sch}(G^\infty, I^i)$ into a tree Υ^i, called the *limit tree* (for initial configuration I^i). The nodes of this tree are the schedules in $\mathbf{Sch}(G^\infty, I^i)$, and there is an edge from node S to node S' if and only if there is a step e such that $S' = S \cdot e$. We also define the *limit forest* Υ to be the set of limit trees $\{\Upsilon^0, \Upsilon^1, \ldots, \Upsilon^n\}$.

In [9], algorithm $\mathcal{A}$ solves the binary version of consensus where processes propose only 0 or 1. So, by the validity property of consensus, the only possible decisions are 0 or 1. In [9], it is shown that the root of each tree Υ^i of the limit forest Υ has a descendant S such that some correct process decides in $S(I^i)$. The root of Υ^i is v-*valent* for $v \in \{0,1\}$ if it has no descendant S such that some correct process decides $u \neq v$ in $S(I^i)$; the root of Υ_i is *multivalent* if it is not v-valent for any $v \in \{0,1\}$. It is clear that the root of Υ^i is either v-valent for exactly one value v, or it is multivalent. The limit forest Υ has a *critical index* $i \in [1..n]$ if and only if the root of Υ^{i-1} is v-valent and the root of Υ^i is u-valent for $u \neq v$ (in which case index i is *univalent critical*) or the root of Υ^i is multivalent (in which case index i is *multivalent critical*).

At a high level, the extraction of Ω in [9] works as follows:

(a) First, it is shown that the limit forest Υ has a critical index i. This part of the proof uses the validity property of consensus.

(b) Then it is shown that for each critical index i of Υ, one can identify a corresponding process j that is necessarily correct in the failure pattern F of the current run.[8] This part of the proof uses only the termination and uniform agreement properties of consensus; in particular, it does not rely on the validity property.

(c) Finally, it is shown how all correct processes can eventually converge on the smallest critical index i of Υ, and on the correct process j that corresponds to i. This part of the proof also does not use the validity property of consensus.

The above three steps outline the extraction of Ω when the given algorithm $\mathcal{A}$ solves binary consensus. Here we want to extract Ω when $\mathcal{A}$ solves QC, and therefore $\mathcal{A}$ also solves binary QC where proposals are only 0 or 1. Note that binary consensus and binary QC share the same uniform agreement and termination properties, and *they differ only in their validity property*.

By the validity property of binary QC, there are now three possible decisions 0, 1, or Q (instead of only 0 or 1). The definitions of v-valent or multivalent nodes remain the same, except that now $v \in \{0, 1, \mathrm{Q}\}$. The definitions of univalent and multivalent critical index i also remain the same.

To extract Ω here, one may try to apply steps (a), (b) and (c) exactly as in [9]. Unfortunately, this does not quite work: with binary QC it is not always the case that the limit forest Υ has a critical index. This is because, in contrast to consensus, the validity property of QC allows processes to decide Q if failures occur. To see why Υ may not have a critical index, suppose some process crashes (in the failure pattern F of the current run). With QC, all the processes that decide "in the limit forest Υ" may decide Q. In this case, the roots of all the trees in Υ are Q-valent, Υ has no critical index, and we cannot apply steps (b) and (c) above to extract the id of a correct process.

This is why, in our transformation algorithm of Figure 6, processes do not always attempt to extract Ω. As Lemma 24 below shows, however, if processes actually attempt to extract Ω (on line 38) then a critical index does exist in the limit forest Υ. It is important to note that if Υ has a critical index, then processes can converge on the identity of a correct process by applying steps (b) and (c) above, exactly as in [9]: This is because, the correctness of steps (b) and (c) does *not* rely on the validity property of consensus (which is the only difference between consensus and QC).

[8] If i is univalent critical, process i is necessarily correct; if i is bivalent critical, the limit tree Υ^i contains a subgraph that reveals the identity of a process j that is necessarily correct.

Lemma 24 *If any process reaches line 34 of algorithm $\mathcal{T}_{\mathcal{D}\to\Psi}$, then the limit forest Υ has a critical index.*

PROOF. If a process reaches line 34, then it decided some value $v \notin \{0, \mathrm{Q}\}$ in the instance of QC that it executed in Thread 2 (line 29). In this instance of QC, each process p can only propose 0 (line 26) or a tuple of the form $(I^j, I^{j+1}, S_p^j, S_p^{j+1})$ (line 29). Thus, by part (i) of the validity property of QC, it must be that some process q proposed $(I^i, I^{i+1}, S_q^i, S_q^{i+1})$ for some index $i \in [0..n]$ (line 29).

Claim 24.1 *There are finite schedules $S_0 \in \textbf{\textit{Sch}}(G^\infty, I^i)$ and $S_1 \in \textbf{\textit{Sch}}(G^\infty, I^{i+1})$ such that some correct process decides 0 in $S_0(I^i)$ and some correct process decides 1 in $S_1(I^{i+1})$.*

PROOF OF CLAIM 24.1. We prove the existence of S_0; the proof for S_1 is symmetric.

Since q proposed $(I^i, I^{i+1}, S_q^i, S_q^{i+1})$ on line 29, it must be that when q executed line 23, say at time t, there is a schedule $S_q^i \in \mathbf{Sch}(G_q^t, I^i)$ such that q decides 0 in $S_q^i(I^i)$. Let p be any correct process. By Lemma 22, $S_q^i \in \mathbf{Sch}(G_p^\infty, I^i)$. By Lemma 21, there is a schedule $S^\infty \in \mathbf{Sch}(G_p^\infty, I^i)$ that extends S_q^i such that $R_\mathcal{A} = (F, H, I^i, S^\infty, -)$ is an *admissible* run of $\mathcal{A}$ (which solves QC) using $\mathcal{D}$ in $\mathcal{E}$). By the termination property of QC, there is a finite prefix S_0 of S^∞ such that p decides in $S_0(I^i)$. Since $S^\infty \in \mathbf{Sch}(G_p^\infty, I^i)$, it follows that $S_0 \in \mathbf{Sch}(G_p^\infty, I^i)$, and, since $G_p^\infty = G^\infty$, $S_0 \in \mathbf{Sch}(G^\infty, I^i)$. Since both S_0 and S_q^j are prefixes of S^∞, one of them is a prefix of the other. Since q decides 0 in $S_q^i(I^i)$, by the uniform agreement property of QC, p also decides 0 in $S_0(I^i)$. $\square_{24.1}$

By Claim 24.1, the root of Υ^i is either 0-valent or multivalent, and the root of Υ^{i+1} is either 1-valent or multivalent. Thus, either the root of Υ^i or Υ^{i+1} is multivalent, or the root of Υ^i is 0-valent and the root of Υ^{i+1} is 1-valent. So, in all cases, there is a critical index in the limit forest Υ. $\square$

8.3 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 the quorums of correct processes contain only correct processes. This is done by the function *extract-quorum*() (lines 45-53) as follows.

Function *extract-quorum*() is called only on line 39, at which point p has agreed with other processes on a tuple (I_0, I_1, S_0, S_1) (line 35). Process p maintains in variable u_p a "recent" failure detector sample of its own. This is initialized to p's most recent sample when p executes line 36, and is updated to p's most recent sample each time p outputs a new quorum (lines 49–50).

To determine the quorum to output, p examines every prefix S of S_0 and S_1, looking for a schedule σ_S that (a) uses only failure detector samples that are "fresher" than u_p, (b) can be appended to S so that $S \cdot \sigma_S$ is a simulated schedule of $\mathcal{A}$, and (c) p decides at the end of that schedule. More precisely, if S is a prefix of S_b, where $b \in \{0, 1\}$, σ_S is required to be a schedule in $\mathbf{Sch}(G_p|u_p, S(I_b))$ (i.e., compatible with a path of samples at least as recent as u_p and applicable to $S(I_b)$), and p must decide in $S \cdot \sigma_S(I_b)$ (see the condition on line 47). If such a schedule σ_S can be found for *every* prefix S of S_0 and S_1, p computes a new quorum consisting of all processes that take steps in these σ_S's (line 49). Otherwise, p's quorum remains unchanged (lines 52–53).

Note how the sample in u_p acts as a "freshness barrier": p's new quorum contains only processes that have taken samples at least as recent as u_p. As we will see in the proof of Lemma 26 below, this (together with the fact that u_p contains ever more recent samples) ensures the completeness property of Σ: the quorum output by a correct process p eventually contains only correct processes.

We will also see in the proof of Lemma 28 that this way of choosing quorums ensures the intersection property of Σ: every two quorums output by any two processes at any times intersect. Intuitively, this follows from the uniform agreement property of QC: if two quorums do not intersect, we would be able to construct

an admissible run of the algorithm $\mathcal{A}$ in which two different values in $\{0, 1, \mathrm{Q}\}$ are decided, establishing a contradiction.

To prove that the completeness property holds, we first prove that the "freshness barrier" u_p is updated infinitely often (line 50), and consequently a new quorum is also computed infinitely often (line 49).

Lemma 25 *Every correct process p that reaches line 34 of algorithm $\mathcal{T}_{\mathcal{D}\to\Psi}$ assigns a quorum to new-quorum$_p$ (line 49) and a node to u_p (line 50) infinitely often.*

PROOF. Suppose some correct process p reaches line 34. It is clear from the algorithm that p either assigns both *new-quorum*$_p$ and u_p infinitely often, or assigns both of them only a finite number of times (see lines 50 and 51). Suppose, for contradiction, that p assigns u_p only a finite number of times. Since p reaches line 34, it also reaches line 36, and so it assigns u_p at least once. Let u be the last node of G_p that p assigns to u_p.

In the next two paragraphs we show that there is a time after which the condition of the if statement on line 47 is true forever. That is, for each prefix S of S_b, where $b \in \{0, 1\}$, there is a finite schedule σ_S, compatible with a path in $G_p|u$, such that $S \cdot \sigma_S \in \mathbf{Sch}(G_p, I_b)$ and p decides in $S \cdot \sigma_S(I_b)$.

Since p reaches line 34, it decided a value different from 0 or Q in the instance of QC that it executed in Thread 2 (line 26 or line 29). By part (i) of the validity property of QC, this decision value must be some tuple (I_0, I_1, S_0, S_1) that some process q proposed in Thread 2 (line 29). Thus, at some time t, for each $b \in \{0, 1\}$, $S_b \in \mathbf{Sch}(G_q^t, I_b)$ (see line 23). By Lemma 22, for each $b \in \{0, 1\}$, $S_b \in \mathbf{Sch}(G_p^\infty, I_b)$.

Consider any prefix S of S_b, where $b \in \{0, 1\}$. Since $S_b \in \mathbf{Sch}(G_p^\infty, I_b)$, it follows that $S \in \mathbf{Sch}(G_p^\infty, I_b)$. By Lemma 21 there is a schedule $S^\infty \in \mathbf{Sch}(G_p^\infty, I_b)$ that extends S such that $R_\mathcal{A} = (F, H, I_b, S^\infty, -)$ is an admissible run of $\mathcal{A}$ (which solves QC) using $\mathcal{D}$ in $\mathcal{E}$. Furthermore, S^∞ can be chosen so that $S^\infty = S \cdot \sigma_S^\infty$, for a schedule σ_S^∞ that is compatible with a path in $G_p^\infty|u$. By the termination property of QC, there is a finite prefix σ_S of σ_S^∞, such that p decides in $S \cdot \sigma_S(I_b)$. Since $S \cdot \sigma_S$ is a finite prefix of $S^\infty = S \cdot \sigma_S^\infty$, and $S^\infty \in \mathbf{Sch}(G_p^\infty, I_b)$, by Lemma 23 it follows that there is some time t_S such that, for all $t \geq t_S$, $S \cdot \sigma_S \in \mathbf{Sch}(G_p^t, I_b)$. Also, since σ_S^∞ is compatible with a path in $G_p^\infty|u$, and σ_S is a finite prefix of σ_S^∞, by Lemma 16, there is a time $\hat{t}_S$ such that, for all $t \geq \hat{t}_S$, σ_S is compatible with a path in $G_p^t|u$. Let $t_1 = \max\{t_S, \hat{t}_S : S \text{ is a prefix of } S_0 \text{ or } S_1\}$. Let t_2 be the time of the last assignment to u_p, and $t^* = \max(t_1, t_2)$. Thus, for all $t \geq t^*$, it is true that $u_p^t = u$ and, for every prefix S of S_b, where $b \in \{0, 1\}$, there is a finite schedule σ_S, compatible with a path in $G_p^t|u_p^t$, such that $S \cdot \sigma_S \in \mathbf{Sch}(G_p^t, I_b)$ and p decides in $S \cdot \sigma_S(I_b)$. In other words, after t^*, the condition of the if statement on line 47 is always satisfied.

Since p is correct and reaches line 37, it executes line 47 infinitely often. The first time after t^* that p executes that line, it finds that the condition of the if statement is satisfied, and assigns a node to u_p on line 50. This occurs after time t_2, contradicting the definition of t_2. □

Lemma 26 *For every correct process p that reaches line 34 of algorithm $\mathcal{T}_{\mathcal{D}\to\Psi}$, there is a time after which Σ-output$_p$ contains only correct processes.*

PROOF. Suppose some correct process p reaches line 34. By Lemma 17, there is a sample v^* of p in G_p^∞ such that $G_p^\infty|v^*$ contains only samples of correct processes. By Lemma 17(a), there is a time after which any node v contained in variable v_p is a descendant of v^* in G_p^∞. By Lemma 25, there are infinitely many assignments to u_p; in all of these u_p is assigned the node in v_p (see lines 36 and 50). Thus, there is a time t^* such that for all $t \geq t^*$, u_p^t is a descendant of v^* in G_p^∞. By Lemma 17(b), for all $t \geq t^*$, $G_p^t|u_p^t$ contains only samples of correct processes.

By Lemma 25, p computes a new quorum on line 49 infinitely often after time t^*. Every quorum assigned to Σ-*output*$_p$ (other than the initialization) is computed on line 49. Thus, it suffices to prove that any quorum assigned to *new-quorum*$_p$ on line 49 after time t^* contains only correct processes.

Consider any such assignment, say at time $t \geq t^*$ (see lines 47–49). The quorum assigned to *new-quorum*$_p$ at time t is the union of certain sets of the form *participants*(σ), where σ is a schedule compatible with $G_p^t|u_p^t$. Since $t \geq t^*$, $G_p^t|u_p^t$ contains only samples of correct processes. This implies that all processes in each such set *participants*(σ) are correct. Therefore, the quorum assigned to *new-quorum*$_p$ at time t contains only correct processes. □

We now prove that the quorums output by $\mathcal{T}_{\mathcal{D}\to\Psi}$ satisfy the intersection property of Σ. Intuitively, we do so by showing that if two quorums do *not* intersect then there are two runs of algorithm $\mathcal{A}$ such that: (a) processes decide differently in these two runs and (b) these two runs can be merged into a single run of $\mathcal{A}$ — a contradiction to the uniform agreement property of QC. To carry out this proof, we need the lemma that allows us to merge certain runs (Lemma 9 of Section 7.1). More precisely, we use the following corollary of this lemma:

Corollary 27 *Let $R_0 = (F, H, I, \hat{S} \cdot S_0, -)$ and $R_1 = (F, H, I, \hat{S} \cdot S_1, -)$ be two finite runs of $\mathcal{A}$ using $\mathcal{D}$ in $\mathcal{E}$ such that participants$(S_0) \cap$ participants$(S_1) = \emptyset$. If some process decides v_0 in R_0 and some process decides v_1 in R_1, then $v_0 = v_1$.*

PROOF. Immediate from Lemma 9, the fact that $\mathcal{A}$ uses $\mathcal{D}$ to solve QC in $\mathcal{E}$, and the uniform agreement property of QC. □

Lemma 28 *For all processes p and q, any two quorums assigned to Σ-output$_p$ and Σ-output$_q$ in algorithm $\mathcal{T}_{\mathcal{D}\to\Psi}$ intersect.*

PROOF. Suppose, for contradiction, that for some processes p and q, there is a time when Σ-*output*$_p = P$ and a time when Σ-*output*$_q = Q$, but $P \cap Q = \emptyset$.

First, observe that any set that p assigns to *new-quorum*$_p$ on line 49 cannot be empty. This is because this set must include the participants of a schedule σ that is applicable to some initial configuration I_0 such that p decides in $\sigma(I_0)$. It is easy to see that no process decides in any initial configuration, and so *participants*$(\sigma) \neq \emptyset$. Thus, any set that p assigns to *new-quorum*$_p$ on line 49 is nonempty. Similarly, any set that q assigns to *new-quorum*$_q$ on line 49 is also non-empty.

Note that at any time, Σ-*output*$_p = \Pi$ (at initialization) or Σ-*output*$_p =$ *new-quorum*$_p$. Similarly, Σ-*output*$_q = \Pi$ or Σ-*output*$_q =$ *new-quorum*$_q$. Since *new-quorum*$_p$ and *new-quorum*$_q$ are not empty, it must be that the non-intersecting quorums P and Q are assigned to *new-quorum*$_p$ and *new-quorum*$_q$, respectively, on line 49.

Since p and q reach line 49, they also reach line 34, and so they decide a value different from 0 or Q in the instance of QC they execute in Thread 2 (line 26 or line 29). By the validity and uniform agreement properties of QC, it must be that $decision_p = decision_q = (I_0, I_1, S_0, S_1)$ such that some process proposed (I_0, I_1, S_0, S_1) in Thread 2 (line 29). Note that I_0 and I_1 are initial configurations of algorithm $\mathcal{A}$ that differ only in the initial state of a single process, and S_0 and S_1 are schedules of $\mathcal{A}$ such that some process decides 0 in $S_0(I_0)$ and 1 in $S_1(I_1)$ (see lines 28–29).

For the notation defined in this and the next paragraph see Figure 8. Let $S_0 = e_1 \dots e_\ell$ and $S_1 = f_1 \dots f_m$, where the e_i's and f_j's are steps. Let $C_0 = I_0$ and $C_i = e_i(C_{i-1})$ for $i \in [1..\ell]$; similarly, $D_0 = I_1$ and $D_j = f_j(D_{j-1})$ for $j \in [1..m]$.

Let t be the time when p first assigns P to *new-quorum*$_p$ on line 49. By the condition on line 47, for each $i \in [0..\ell]$, there is a schedule σ_i^p such that $e_1 \dots e_i \cdot \sigma_i^p \in \mathbf{Sch}(G_p^t, I_0)$ and p decides some value x_i^p in $e_1 \dots e_i \cdot \sigma_i^p(I_0)$.[9] Similarly, for each $j \in [0..m]$, there is a schedule τ_j^p such that $f_1 \dots f_j \cdot \tau_j^p \in \mathbf{Sch}(G_p^t, I_1)$ and p decides some value y_i^p in $f_1 \dots f_j \cdot \tau_j^p(I_1)$. The quorum P is the union of the participants in the σ_i^p's

[9] We adopt the convention that, for $i = 0$, $e_1 \dots e_i$ is the empty schedule.

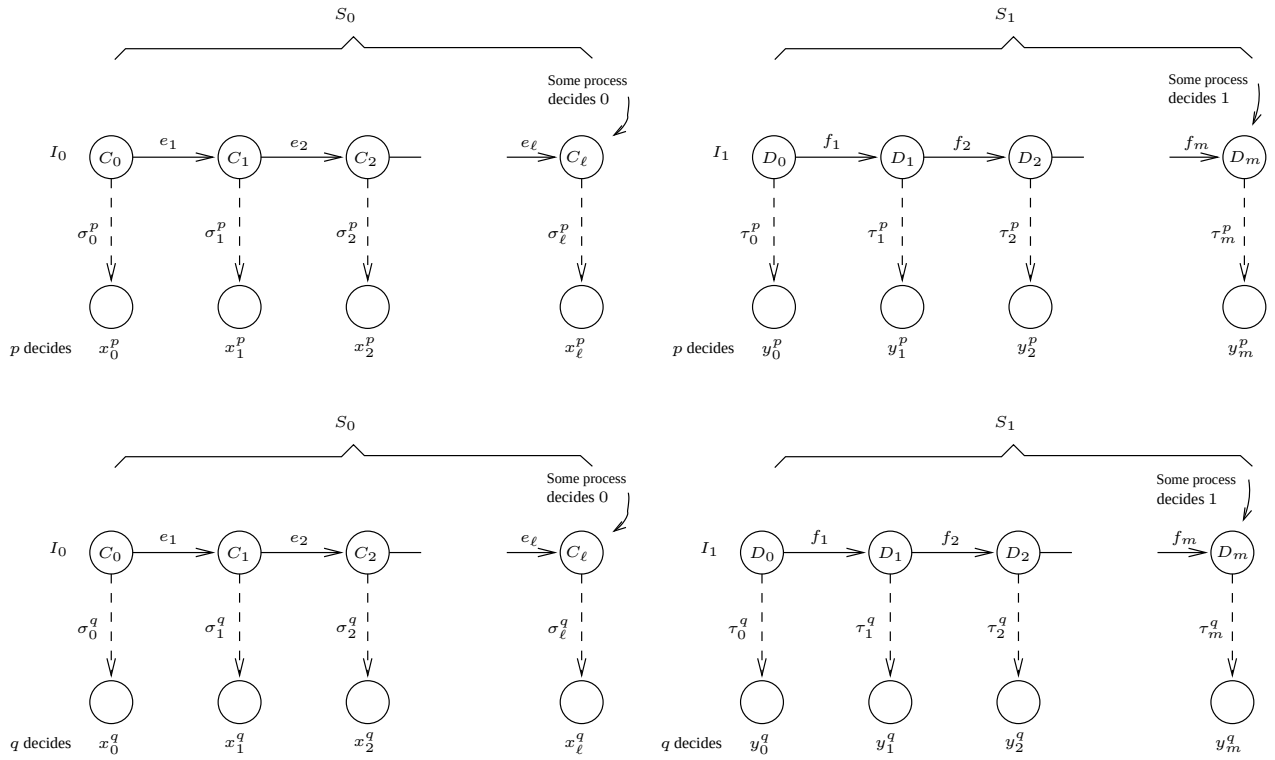

Figure 8: Illustration of the proof of Lemma 28

and τ_j^p's. Similarly, let t' be the time when q first assigns Q to *new-quorum*$_q$. We define σ_i^q, x_i^q, τ_j^q, and y_j^q, in an analogous manner. The quorum Q is the union of the participants of the σ_i^q's and τ_j^q's.

Claim 28.1 *For all $i \in [0..\ell]$, $x_i^p = x_i^q$; and for all $j \in [0..m]$, $y_j^p = y_j^q$.*

PROOF OF CLAIM 28.1. Since $e_1 \dots e_i \cdot \sigma_i^p \in \mathbf{Sch}(G_p^t, I_0)$, by Lemma 20, there is a run $R_0 = (F, H, I_0, e_1 \dots e_i \cdot \sigma_i^p, -)$ of $\mathcal{A}$ using $\mathcal{D}$ in $\mathcal{E}$. Process p decides x_i^p in R_0. Similarly, there is a run $R_1 = (F, H, I_0, e_1 \dots e_i \cdot \sigma_i^q, -)$ of $\mathcal{A}$ using $\mathcal{D}$ in $\mathcal{E}$, in which q decides x_i^q. Since P and Q are disjoint, so are their subsets *participants*(σ_i^p) and *participants*(σ_i^q). Thus, by Corollary 27 (applied with $I = I_0$, $\hat{S} = e_1 \dots e_i$, $S_1 = \sigma_i^p$, $S_2 = \sigma_i^q$, $v_0 = x_i^p$ and $v_1 = x_i^q$), we have that $x_i^p = x_i^q$. The proof that $y_j^p = y_j^q$ is analogous. $\square_{28.1}$

By Claim 28.1, we can now define $x_i = x_i^p = x_i^q$ and $y_j = y_j^p = y_j^q$.

Claim 28.2 *For all $i \in [0..\ell - 1]$, $x_{i+1} = x_i$; and for all $j \in [0..m - 1]$, $y_{j+1} = y_j$.*

PROOF OF CLAIM 28.2. Consider any $i \in [0..\ell - 1]$, and let r be the process that takes step e_{i+1}. Since P and Q are disjoint, $r \notin P$ or $r \notin Q$. Without loss of generality, suppose that $r \notin P$. In particular, $r \notin$ *participants*$(\sigma_i^p) \subseteq P$. Also, again because P and Q are disjoint, so are their subsets *participants*(σ_i^p) and *participants*(σ_{i+1}^q). Therefore, *participants*(σ_i^p) and *participants*$(e_{i+1} \cdot \sigma_{i+1}^q)$ are disjoint.

Since $e_1 \dots e_i \cdot \sigma_i^p \in \mathbf{Sch}(G_p^t, I_0)$, by Lemma 20, there is a run $R_0 = (F, H, I_0, e_1 \dots e_i \cdot \sigma_i^p, -)$ of $\mathcal{A}$ using $\mathcal{D}$ in $\mathcal{E}$. Process p decides x_i in R_0. Also, since $e_1 \dots e_{i+1} \cdot \sigma_{i+1}^q \in \mathbf{Sch}(G_q^{t'}, I_0)$ there is a run $R_1 = (F, H, I_0, e_1 \dots e_i e_{i+1} \cdot \sigma_{i+1}^q, -)$ of $\mathcal{A}$ using $\mathcal{D}$ in $\mathcal{E}$, in which q decides x_{i+1}. Thus, by Corollary 27 (applied with $I = I_0$, $\hat{S} = e_1 e_2 \dots e_i$, $S_1 = \sigma_i^p$, $S_2 = e_{i+1} \cdot \sigma_{i+1}^q$, $v_0 = x_i$ and $v_1 = x_{i+1}$), we have that $x_i = x_{i+1}$. The proof that $y_j = y_{j+1}$ is analogous. $\square_{28.2}$

Claim 28.3 *$x_0 = 0$ and $y_0 = 1$.*

PROOF OF CLAIM 28.3. Recall that some process decides 0 in $S_0(I_0)$, and p decides x_ℓ in $S_0 \cdot \sigma_\ell^p(I_0)$. Since $S_0 \cdot \sigma_\ell^q \in \mathbf{Sch}(G_q^t, I_0)$, by Lemma 20, there is a run $(F, H, I_0, S_0 \cdot \sigma_\ell^p, -)$ of $\mathcal{A}$ using $\mathcal{D}$ in $\mathcal{E}$. In this run, some process decides 0 and p decides x_ℓ. Since $\mathcal{A}$ solves QC using $\mathcal{D}$ in $\mathcal{E}$, by the uniform agreement property of QC, we have $x_\ell = 0$. By Claim 28.2 and a trivial induction, $x_i = 0$ for all $i \in [0..\ell]$. In particular, $x_0 = 0$. The proof that $y_0 = 1$ is analogous. $\square_{28.3}$

Since P and Q are disjoint, so are their subsets *participants*(σ_0^p) and *participants*(τ_0^q). Let r be the process such that I_0 and I_1 differ only in the initial state of r. Process 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, and p decides the same value, x_0, in $\sigma_0^p(I_1)$ as in $\sigma_0^p(I_0)$. Since $\sigma_0^p \in \mathbf{Sch}(G_p^t, I_0)$, we also have that $\sigma_0^p \in \mathbf{Sch}(G_p^t, I_1)$. By Lemma 20, there is a run $R_0 = (F, H, I_1, \sigma_0^p, -)$ of $\mathcal{A}$ using $\mathcal{D}$ in $\mathcal{E}$. Process p decides x_0 in R_0. Since $\tau_0^q \in \mathbf{Sch}(G_q^{t'}, I_1)$, again by Lemma 20, there is a run $R_1 = (F, H, I_1, \tau_0^q, -)$ of $\mathcal{A}$ using $\mathcal{D}$ in $\mathcal{E}$. Process q decides y_0 in R_1.

By Corollary 27 (applied with $I = I_1$, $\hat{S}$ being the empty schedule, $S_1 = \sigma_0^p$ and $S_2 = \tau_0^q$, $v_0 = x_0$ and $v_1 = y_0$), we have that $x_0 = y_0$. This contradicts Claim 28.3, and completes the proof of Lemma 28. $\square$

8.4 Correctness of the transformation

We are now ready to show that:

Theorem 29 *Algorithm $\mathcal{T}_{\mathcal{D}\to\Psi}$ transforms $\mathcal{D}$ to Ψ.*

PROOF. Recall that algorithm $\mathcal{A}$ uses $\mathcal{D}$ to solve QC in $\mathcal{E}$. As before, we consider an arbitrary admissible run of $\mathcal{T}_{\mathcal{D}\to\Psi}$ in $\mathcal{E}$, where $F \in \mathcal{E}$ is the failure pattern and $H \in \mathcal{D}(F)$ is the failure history of this run.

To show that $\mathcal{T}_{\mathcal{D}\to\Psi}$ transforms $\mathcal{D}$ to Ψ, we must prove that the values of the variables Ψ-*output*$_p$ conform to the specification of Ψ. By inspection of $\mathcal{T}_{\mathcal{D}\to\Psi}$, it is clear that Ψ-*output*$_p$ is either $\perp$, or **red** (in which case we say that it is of type $\mathcal{FS}$), or a pair (q, Q) where $q \in \Pi$ and $Q \subseteq \Pi$ (in which case we say that it is of type (Ω, Σ)).

(1) *For each process p, Ψ-output$_p$ is initially $\perp$ (line 4). If Ψ-output$_p$ ever changes value, it becomes of type $\mathcal{FS}$ forever (line 33) or of type (Ω, Σ) forever (line 40).*

This follows by inspection of $\mathcal{T}_{\mathcal{D}\to\Psi}$.

(2) *For all distinct 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 QC, p and q cannot decide different values on lines 26 and 29; thus, they cannot execute in different branches of the if-then-else statement of lines 32-34.

(3) *For each correct process p, eventually Ψ-output$_p \neq \perp$.*

To prove this we first show that every correct process p eventually completes the loop on lines 20–22. By Lemma 21 (taking S to be the empty schedule), for each $j \in [0..n]$ there is a schedule $S^\infty \in \mathbf{Sch}(G_p^\infty, I^j)$ such that $(F, H, I^j, S^\infty, -)$ is an admissible run of algorithm $\mathcal{A}$ (which solves QC) using $\mathcal{D}$ in $\mathcal{E}$. By the termination property of QC, there is a finite prefix S^j of S^∞ such that p decides in $S^j(I^j)$. By Lemma 23, there is a time t^j such that for all $t \geq t^j$, $S^j \in \mathbf{Sch}(G_p^t, I^j)$. Thus, after time $\max\{t^j : j \in [0..n]\}$, the exit condition on line 22 is true forever, and so eventually p completes the loop.

We claim that, after completing the loop on lines 20–22, every correct process p executes line 26 or line 29.

To show this claim, first note that since p completes this loop, then for every $j \in [0..n]$, there is a time t_j and a schedule $S_p^j \in \mathbf{Sch}(G_p^{t_j}, I^j)$ such that p decides some value x_p^j in $S_p^j(I^j)$. By Lemma 20, for all $j \in [0..n]$, there is a run $R_p^j = (F, H, I^j, S_p^j, -)$ of $\mathcal{A}$ using $\mathcal{D}$ in $\mathcal{E}$. Since (a) $\mathcal{A}$ solves QC using $\mathcal{D}$ in $\mathcal{E}$, (b) processes can only propose 0 or 1 in R_p^j, and (c) p decides x_p^j in R_p^j, then by the validity property of QC, $x_p^j \in \{0, 1, \mathrm{Q}\}$. Furthermore, since no process proposes 1 in the run R_p^0 whose initial configuration is I^0, we have $x_p^0 \in \{0, \mathrm{Q}\}$. Similarly, $x_p^n \in \{1, \mathrm{Q}\}$.

There are two possible cases:

- There is a $j \in [0..n]$ such that $x_p^j = \mathrm{Q}$. In this case, p executes line 26.
- For all $j \in [0..n]$, $x_p^j \neq \mathrm{Q}$. In this case, for all $j \in [0..n]$, $x_p^j \in \{0, 1\}$; moreover, $x_p^0 = 0$ and $x_p^n = 1$. So there must be some $i \in [0..n-1]$ such that $x_p^i = 0$ and $x_p^{i+1} = 1$. Thus, p executes line 29.

Thus, p executes line 26 or line 29, which shows the claim.

From this claim, all correct processes propose some value (on line 26 or 29) in an instance of QC executed in Thread 2. By the termination property of QC, all correct processes eventually decide in that instance, and so they all complete line 31. Thus, eventually every correct process p sets Ψ-*output*$_p$ to a non-$\perp$ value on line 33 or 40.

(4) *For each process p and time t, if Ψ-output$_p^t$ = **red** then a failure occurred by time t.*

To see this, let p be a process and t a time such that Ψ-*output*$_p^t$ = **red** (line 33). By lines 31-32, p decided 0 or Q on line 26 or 29 at some time $t' \leq t$. There are two possible cases:

(a) p decided Q at time $t' \leq t$. Then, by part (ii) of the validity property of QC, a failure occurred by time $t' \leq t$.

(b) p decided 0 at time $t' \leq t$. Then, by Observation 2, there must be at least one process q that proposes 0 and executes a step of the QC algorithm on line 26 by time $t' \leq t$. This implies that there is a time $t'' \leq t'$, an index $j \in [0..n]$, and a schedule $S_q^j \in \mathbf{Sch}(G_q^{t''}, I^j)$, such that q decides Q in $S_q^j(I^j)$ (see lines 23–25). By Lemma 20, there is a list of times T, *all at most* t'', such that (F, H, I^j, S_q^j, T) is a run of $\mathcal{A}$ using $\mathcal{D}$ in $\mathcal{E}$. By part (ii) of the validity property of QC, this implies that a failure occurred by time $t'' \leq t' \leq t$.

(5) *If the Ψ-output variable of some process p becomes of type $\mathcal{FS}$ at time t, then:*

- *A failure occurred by time t.*
  If Ψ-*output*$_p^t$ is of type $\mathcal{FS}$, then, by inspection of $\mathcal{T}_{\mathcal{D}\to\Psi}$, Ψ-*output*$_p^t$ = **red**. By (4), a failure occurred by time t.
- *For every correct process q, there is a time after which Ψ-output$_q$ = **red**.*
  By (2) and (3), for every correct process q, there is a time after which Ψ-*output*$_q$ is of type $\mathcal{FS}$. In $\mathcal{T}_{\mathcal{D}\to\Psi}$, the variable Ψ-*output*$_q$ can become of type $\mathcal{FS}$ only by being set to **red**.

(6) *If the Ψ-output variable of some process becomes of type (Ω, Σ), then:*

- *(i) For every process p and every time $t \in \mathbb{N}$, Ω-output$_p^t \in \Pi$; furthermore, (ii) if a correct process exists, then there is a correct process p^* and a time t^* such that, for every correct process p and every time $t \geq t^*$, Ω-output$_p^t = p^*$.*

Part (i) is immediate by inspection of $\mathcal{T}_{\mathcal{D}\to\Psi}$. Part (ii) is trivial if all processes are faulty, so suppose that some correct process exists. By assumption, the Ψ-*output* variable of some process becomes of type (Ω, Σ). Then, by (2) and (3) above, eventually the Ψ-*output* variable of every correct process also becomes of type (Ω, Σ). So every correct process sets its Ω-*output* variable repeatedly on line 38 using the extraction procedure described in [9]. Since some process reaches line 34, by Lemma 24, the limit forest Υ has a critical index. Thus, as we explained in Section 8.2, we can now apply steps (b) and (c) of the proof of [9] to show that there is some correct process p^* and a time t^* such that, for every correct process p and time $t \geq t^*$, Ω-*output*$_p^t = p^*$. The only difference is that whenever the proof in [9] refers to a *bi*valent node, we now refer to a *multi*valent one; and whenever [9] refers to u-valent versus v-valent nodes for some distinct u and v in $\{0, 1\}$, here u and v are in $\{0, 1, \mathrm{Q}\}$.

- *(i) For every correct process p, there is a time after which Σ-output$_p$ contains only correct processes, and (ii) for all processes p and q, any two quorums assigned to Σ-output$_p$ and Σ-output$_q$ intersect.*
  Part (i) was shown in Lemma 26 and part (ii) in Lemma 28.

From (1)–(6) above, it follows that the values of the variables Ψ-*output* conform to the specification of Ψ, as defined in Section 3: Initially, Ψ-*output* $= \perp$ at each process; eventually, however, Ψ-*output* behaves either like the failure detector (Ω, Σ) at all correct processes or like the failure detector $\mathcal{FS}$ at all correct processes. The switch from $\perp$ to (Ω, Σ) or $\mathcal{FS}$ is consistent at all processes, and a switch from $\perp$ to $\mathcal{FS}$ can happen only if a failure occurred.

□

Since Theorem 29 holds for any environment $\mathcal{E}$ and any failure detector $\mathcal{D}$ that can be used to solve QC in $\mathcal{E}$, we conclude that:

Theorem 30 *For every environment $\mathcal{E}$, if failure detector $\mathcal{D}$ can be used to solve QC in $\mathcal{E}$, then $\mathcal{D}$ can be transformed to Ψ in $\mathcal{E}$.*

8.5 Binary versus multivalued QC

Our proof that Ψ is the weakest failure detector to solve QC uses the fact that, in QC, each process can propose any value in the infinite set $\{0, 1\}^*$; i.e., the proof used the fact that QC is *multivalued*.[10] So one may ask whether Ψ is also the weakest failure detector to solve the *binary* version of QC where processes can only propose 0 or 1. The answer is affirmative.

To prove this, we use an algorithm by Mostéfaoui *et al.* that converts any algorithm that solves *binary* consensus into an algorithm that solves *multivalued* consensus [34]. With a straightforward modification, this conversion algorithm also works with quittable consensus: it converts any algorithm that solves binary QC (using some failure detector $\mathcal{D}$) into one that solves multivalued QC (using the *same* failure detector $\mathcal{D}$). This gives us the following:

Theorem 31 *For every environment $\mathcal{E}$, if failure detector $\mathcal{D}$ can be used to solve binary QC in $\mathcal{E}$, then $\mathcal{D}$ can be used to solve QC in $\mathcal{E}$.*

Therefore:

Corollary 32 *For every environment $\mathcal{E}$, Ψ is the weakest failure detector to solve binary QC in $\mathcal{E}$.*

[10] Specifically, in Thread 2 of $\mathcal{T}_{\mathcal{D}\to\Psi}$, processes may propose tuples of the form (I, I', S, S'), for some initial configurations and finite schedules of algorithm $\mathcal{A}$.

PROOF. Let $\mathcal{E}$ be any environment.

(a) Ψ can be used to solve binary QC in $\mathcal{E}$. This is obvious since, by Theorem 8, Ψ can be used to solve QC in $\mathcal{E}$.

(b) Suppose $\mathcal{D}$ can be used to solve binary QC in $\mathcal{E}$. By Theorem 31, $\mathcal{D}$ can be used to solve QC in $\mathcal{E}$. So, by Theorem 30, $\mathcal{D}$ can be transformed to Ψ in $\mathcal{E}$.

□

9 Final remarks

Failure detector emulations. Intuitively, a failure detector $\mathcal{D}$ is weaker than a failure detector $\mathcal{D}'$ if processes can use $\mathcal{D}'$ to emulate $\mathcal{D}$. Two technical definitions of failure detector emulation have been proposed in the literature [9, 31]. In this paper we adopted the original definition of emulation given in [9] since we used parts of the proof given in that paper. As we explain below, however, our results also hold with the definition of emulation given in [31].

With the original definition of emulation [9], an implementation of $\mathcal{D}$ must maintain local variables that mirror the output of $\mathcal{D}$ *at all times*. The definition of emulation given in [31] is weaker: with this definition, an implementation of $\mathcal{D}$ is required to behave like $\mathcal{D}$ *only when it is actually queried*.[11] The failure detectors Ψ and $(\Psi, \mathcal{FS})$, which we proved here to be the weakest for QC and NBAC under the original definition of failure detector emulation, are also weakest for these problems under the definition of emulation given in [31]. In a nutshell, this is because (a) all the algorithms that we give here also work under the model of [31], and (b) if processes can emulate a failure detector $\mathcal{D}$ according to the strong definition of emulation of [9] (i.e., p is able to maintain a variable $\mathcal{D}$-$output_p$ that *always* mirrors the output of $\mathcal{D}$) then processes can also emulate $\mathcal{D}$ according to the weaker definition of emulation of [31]: whenever it is queried, p can just return the value of $\mathcal{D}$-$output_p$. For the same reasons, all the failure detectors that we are aware of to be weakest for a problem under the definition of emulation of [9], are also weakest for these problems under the definition of emulation of [31]; this includes the weakest failure detectors for consensus [9] and non-uniform consensus [19], set agreement [18, 37], implementing an atomic register [15], and boosting obstruction-freedom to wait-freedom [25].

The newer definition of emulation given in [31] has two advantages over the original one of [9]. First, the original definition of emulation is more stringent than necessary: when using an emulated failure detector $\mathcal{D}$, it is sufficient that the emulated $\mathcal{D}$ behaves correctly only when it is queried — which is exactly what the newer definition stipulates. Second, the definition of emulation given in [31] is reflexive, i.e., for every failure detector $\mathcal{D}$, processes can use $\mathcal{D}$ to emulate $\mathcal{D}$. In contrast, as remarked by [31] and later in [11], the original definition of emulation is not reflexive: if the output of a failure detector $\mathcal{D}$ is sensitive to time, the processes, because they are asynchronous, may not be able to maintain variables that mirror the output of $\mathcal{D}$ *at all times* as the original definition of emulation requires. The non-reflexivity of the failure detector emulation under the original definition of emulation of [9] has no bearing on the results of this paper or on the other weakest failure detector results cited above: as we explained above, the same results also hold with the newer definition of emulation given in [31] which does satisfy reflexivity.

Granularity of steps. As in the models of Fischer *et al.* [20] and Chandra *et al.* [9], in our model a process can send a message m to every process in an atomic step. Since a sender cannot fail in "the middle" of a step that sends m to all, our model has the following property: if any process receives m, then every correct process

[11] More precisely, if the implementation of $\mathcal{D}$ is queried at time t_1 and it replies with a value d at time t_2, then d must be a valid value of $\mathcal{D}$ at some time $t \in [t_1, t_2]$; so, it is as if the query/reply occurred *atomically* at some time t within the interval of time that the query/reply actually took. In other words, the behaviour of the implementation of $\mathcal{D}$ is *linearizable* with respect to the specification of $\mathcal{D}$.

eventually receives m (*). One may ask whether our results also hold in another model, let's call it model B, where a process can send a message m to only one process in an atomic step. To answer this question, note that in model B (where processes may crash but links are reliable) one can implement Uniform Reliable Broadcast (URB) [27], a communication primitive that provides the property (*) of our model. Since model B can emulate the atomic "send m to all" of our model, it is easy to see that our results also hold in model B.

Systems that are not asynchronous. It is worth noting that since our model is that of an asynchronous system augmented with failure detectors, the algorithms that emulate failure detectors are also asynchronous [11]. So the weakest failure detectors that result from such emulations are also asynchronous in the sense that their output values could be delayed for any finite time. Some previous works explored failure detectors in systems that are not purely asynchronous. For example Aguilera *et al.* investigated the use of "fast " failure detectors to speed up agreement algorithms in some synchronous systems [4]. In another body of work, researchers considered the definition and implementation of failure detectors for systems where message delays and losses follow some probability distribution [14, 8, 35]. It may be interesting to investigate QC and NBAC in systems that are not asynchronous, and to determine whether these problems have weakest failure detectors in these systems. This, however, is beyond the scope of this paper.

Systems with a majority of correct processes. 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. Therefore, in such environments Ψ is equivalent to a simpler failure detector, one which outputs just Ω instead of (Ω, Σ).

Future failures. 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 occurs.

References

[1] Marcos Kawazoe Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. Communication-efficient leader election and consensus with limited link synchrony. In *PODC '04: Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing*, pages 328–337, 2004.

[2] Marcos Kawazoe Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. On implementing Omega in systems with weak reliability and synchrony assumptions. *Distributed Computing*, 21(4):285–314, 2008.

[3] Marcos Kawazoe Aguilera, Sven Frolund, Vassos Hadzilacos, Stephanie Horn, and Sam Toueg. Abortable and query-abortable objects and their efficient implementation. In *PODC '07: Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 23–32, 2007.

[4] Marcos Kawazoe Aguilera, Gérard Le Lann, and Sam Toueg. On the impact of fast failure detectors on real-time fault-tolerant systems. In *DISC '02: Proceedings of the Sixteenth International Symposium on Distributed Computing*, pages 354–370, 2002.

[5] Marcos Kawazoe Aguilera, Sam Toueg, and Boris Deianov. Revisiting the weakest failure detector for uniform reliable broadcast. In *DISC '99: Proceedings of the Thirteenth International Symposium on Distributed Computing*, pages 13–33, 1999.

[6] Antonio Fernández Anta, Ernesto Jiménez, and Michel Raynal. Eventual leader election with weak assumptions on initial knowledge, communication reliability, and synchrony. *J. Comput. Sci. Technol.*, 25(6):1267–1281, November 2010.

[7] Hagit Attiya, Rachid Guerraoui, Danny Hendler, and Petr Kuznetsov. The complexity of obstruction-free implementations. *Journal of the ACM*, 56(4):24:1–24:33, June 2009.

[8] Marin Bertier, Olivier Marin, and Pierre Sens. Implementation and performance evaluation of an adaptable failure detector. In *DSN '02: Proceedings of the Thirty-second annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 354–363, 2002.

[9] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.

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

[11] Bernadette Charron-Bost, Martin Hutle, and Josef Widder. In search of lost time. *Information Processing Letters*, 110:928–933, October 2010.

[12] Bernadette Charron-Bost and Sam Toueg. Unpublished notes, 2001.

[13] Wei Chen. Abortable consensus and its application to probabilistic atomic broadcast. Technical Report MSR-TR-2006-135, Microsoft Research, 2006.

[14] Wei Chen, Sam Toueg, and Marcos Kawazoe Aguilera. On the quality of service of failure detectors. *IEEE Trans. Comput.*, 51(5):561–580, May 2002.

[15] Carole Delporte-Gallet, Hugues Fauconnier, and Rachid Guerraoui. Tight failure detection bounds on atomic object implementations. *Journal of the ACM*, 57(4):22:1–22:32, April 2010.

[16] Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, Vassos Hadzilacos, Petr Kouznetsov, and Sam Toueg. The weakest failure detectors to solve certain fundamental problems in distributed computing. In *PODC '04: Proceedings of the Twenty-third ACM Symposium on Principles of Distributed Computing*, pages 338–346, 2004.

[17] Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, and Petr Kouznetsov. Mutual exclusion in asynchronous systems with failure detectors. *Journal of Parallel and Distributed Computing*, 65(4):492–505, April 2005.

[18] Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, and Andreas Tielmann. The weakest failure detector for message passing set-agreement. In *DISC '08: Proceedings of the Twenty-second International Symposium on Distributed Computing*, pages 109–120, 2008.

[19] Jonathan Eisler, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector to solve nonuniform consensus. *Distributed Computing*, 19(4):335–359, 2007.

[20] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[21] Matthias Fitzi, Daniel Gottesman, Martin Hirt, Thomas Holenstein, and Adam Smith. Detectable byzantine agreement secure against faulty majorities. In *PODC '02: Proceedings of the Twenty-first ACM Symposium on Principles of Distributed Computing*, pages 118–126, 2002.

[22] Eddy Fromentin, Michel Raynal, and Frederic Tronel. On classes of problems in asynchronous distributed systems with process crashes. In *ICDCS '99: Proceedings of the Nineteenth International Conference on Distributed Computing Systems*, pages 470–477, 1999.

[23] James 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. Also appears as Technical Report RJ2188, IBM Research Laboratory.

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

[25] Rachid Guerraoui, Michal Kapalka, and Petr Kouznetsov. The weakest failure detectors to boost obstruction-freedom. *Distributed Computing*, 20(6):415–433, 2008.

[26] Rachid Guerraoui and Petr Kouznetsov. On the weakest failure detector for non-blocking atomic commit. In *TCS '02: Proceedings of the Second International Conference on Theoretical Computer Science*, pages 461–473, 2002.

[27] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR 94-1425, Department of Computer Science, Cornelpublisl University, Dept. of Computer Science, Cornell University, Ithaca, NY 14853, May 1994.

[28] Vassos Hadzilacos. On the relationship between the atomic commitment and consensus problems. In Barbara B. Simons and Alfred Z. Spector, editors, *Fault-Tolerant Distributed Computing*, volume 448 of *LNCS*, pages 201–208. Springer-Verlag, 1986.

[29] Joseph Y. Halpern and Aleta Ricciardi. A knowledge-theoretic analysis of uniform distributed coordination and failure detectors. *Distributed Computing*, 17(3):223–236, 2005.

[30] Martin Hutle, Dahlia Malkhi, Ulrich Schmid, and Lidong Zhou. Chasing the weakest system model for implementing Ω and consensus. *IEEE Trans. Dependable Sec. Comput.*, 6(4):269–281, October 2009.

[31] Prasad Jayanti and Sam Toueg. Every problem has a weakest failure detector. In *PODC '08: Proceedings of the Twenty-seventh ACM Symposium on Principles of Distributed Computing*, pages 75–84, 2008.

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

[33] Dahlia Malkhi, Florian Oprea, and Lidong Zhou. *Omega* meets Paxos: leader election and stability without eventual timely links. In *DISC '05: Nineteenth International Symposium on Distributed Computing*, pages 199–213, 2005.

[34] Achour Mostéfaoui, Michel Raynal, and Frederic Tronel. From binary consensus to multivalued consensus in asynchronous message-passing systems. *Information Processing Letters*, 73(5–6):207–212, March 2000.

[35] Nicolas Schiper and Sam Toueg. A robust and lightweight stable leader election service for dynamic systems. In *DSN '08: Proceedings of the Thirty-eighth IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 207–216, 2008.

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

[37] Piotr Zielinski. Anti-Ω: the weakest failure detector for set agreement. *Distributed Computing*, 22(5-6):335–348, August 2010.

A Proof of Lemma 9

Lemma 9 *Let $R_0 = (F, H, I, \hat{S} \cdot S_0, \hat{T}_0 \cdot T_0)$ and $R_1 = (F, H, I, \hat{S} \cdot S_1, \hat{T}_1 \cdot T_1)$ be two finite runs of an algorithm $\mathcal{A}$ using failure detector $\mathcal{D}$ in some environment $\mathcal{E}$, such that $|\hat{T}_0| = |\hat{T}_1| = |\hat{S}|$ and participants$(S_0) \cap$ participants$(S_1) = \emptyset$. Let $R = (F, H, I, \hat{S} \cdot S, \hat{T} \cdot T)$ be a merging of R_0 and R_1. Then*

(a) R is also a run of $\mathcal{A}$ using $\mathcal{D}$ in $\mathcal{E}$.

(b) For each $b \in \{0, 1\}$ and each process $p \in$ participants$(\hat{S} \cdot S_b)$, the state of p is the same in $\hat{S} \cdot S(I)$ as in $\hat{S} \cdot S_b(I)$.

PROOF. To show that R is run of $\mathcal{A}$ using $\mathcal{D}$ in $\mathcal{E}$, we first note that $F \in \mathcal{E}$, $H \in \mathcal{D}(F)$, and I is indeed an initial configuration of $\mathcal{A}$. It now suffices to show that R satisfies properties (1)–(5) of runs. The fact that $\hat{S} \cdot S$ and $\hat{T} \cdot T$ have the same length (property (2)) is obvious from the definition of R. The fact that in R no process takes a step after it has crashed, and that the failure detector value in each step is consistent with the history H (property (3)), follows from the way R is constructed from R_0 and R_1, and the fact that R_0 and R_1 have this property. $\hat{T} \cdot T$ is nondecreasing (property (4)) because each of $\hat{T}_0 \cdot T_0$ and $\hat{T}_1 \cdot T_1$ is nondecreasing, $\hat{T}$ is chosen to be whichever of $\hat{T}_0$ and $\hat{T}_1$ has the smallest maximum element, and T is obtained by merging T_0 and T_1 in nondecreasing order. The times of the steps in R respect the causal precedence relation (property (5)) because R_0 and R_1 have this property, and no process takes a step in both S_0 and S_1. It remains to prove that $\hat{S} \cdot S$ is applicable to I (property (1)).

For the purposes of this proof, if σ is a schedule and $i \in \{0, 1, \ldots, |\sigma|\}$, we denote by σ^i the prefix of σ that has length i (σ^0 is the empty schedule). Also, for the suffix S of the schedule of the merged run R (i.e., the portion of the schedule of R produced by merging S_0 and S_1), and $b \in \{0, 1\}$, let $f_b(i)$ be the number of steps of S^i that come from S_b. Using a straightforward induction, we can show that for all $i \in \{0, 1, \ldots, |S|\}$:

(i) For all $b \in \{0, 1\}$, the set of messages between processes in *participants*$(\hat{S} \cdot S_b)$ (i.e., messages of the form $(p, -, q)$ where $p, q \in$ *participants*$(\hat{S} \cdot S_b)$) in the message buffer of configuration $\hat{S} \cdot S^i(I)$ is equal to the set of messages between processes in *participants*$(\hat{S} \cdot S_b)$ in the message buffer of configuration $\hat{S} \cdot S_b^{f_b(i)}(I)$.

(ii) For all $b \in \{0, 1\}$, the state of any process $p \in$ *participants*$(\hat{S} \cdot S_b)$ is the same in $\hat{S} \cdot S^i(I)$ as in $\hat{S} \cdot S_b^{f_b(i)}(I)$.

Below we use (i) to show that, for each $i \in \{1, 2, \ldots, |S|\}$, $S[i]$ is applicable to $\hat{S} \cdot S^{i-1}(I)$. This proves that $\hat{S} \cdot S$ is applicable to I.

Let $S[i] = (p, m, d, \mathcal{A})$. Let $b \in \{0, 1\}$ be such that $p \in$ *participants*(S_b) (such a b exists because every step of S is in either S_0 or S_1). Thus, $(p, m, d, \mathcal{A})$ is step $f_b(i)$ of S_b. Since R_b is a run, $\hat{S} \cdot S_b$ is applicable to I. In particular, step $(p, m, d, \mathcal{A})$ of S_b is applicable to $\hat{S} \cdot S_b^{f_b(i)-1}(I)$. Note that $f_b(i-1) = f_b(i) - 1$. So, $(p, m, d, \mathcal{A})$ is applicable to $\hat{S} \cdot S_b^{f_b(i-1)}(I)$. Thus, m is in the message buffer of $\hat{S} \cdot S_b^{f_b(i-1)}(I)$. Furthermore, it is a message between processes in *participants*$(\hat{S} \cdot S_b)$. This is because, (1) being in the message buffer of $\hat{S} \cdot S_b^{f_b(i-1)}(I)$ it was sent by a process in *participants*$(\hat{S} \cdot S_b^{f_b(i-1)})$; and (2) p, the recipient of m, is the process that takes the $f_b(i)$-th step of S_b. By (i), m is in the message buffer of $\hat{S} \cdot S^{i-1}(I)$. So, $(p, m, d, \mathcal{A})$ is applicable to $\hat{S} \cdot S^{i-1}(I)$, as wanted.

Part (b) of the lemma follows directly from (ii), taking $i = |S|$. □