Non-Blocking Atomic Commit in Asynchronous Distributed Systems with Failure Detectors

Rachid Guerraoui

Communication Systems Department

Swiss Federal Institute of Technology

CH-1015 Lausanne

Abstract

This paper addresses the Non-Blocking Atomic Commit (NB-AC) problem in asynchronous distributed systems augmented with failure detectors. We first show that, in these systems, NB-AC and Consensus are incomparable. Roughly speaking, there is a failure detector that solves NB-AC but not Consensus and a failure detector that solves Consensus but not NB-AC. Then we introduce the Anonymously Perfect failure detector $?\mathcal{P}$. We show that, to solve NB-AC, $?\mathcal{P}$ is necessary (while $\mathcal{P}$ is not), whereas $?\mathcal{P} + \diamond\mathcal{S}$ is sufficient when a majority of the processes are correct. We draw from our results some observations on the practical solvability of NB-AC.

1 Introduction

To ensure the atomicity of a distributed transaction, an agreement problem must be solved among the set of the processes participating in the transaction (e.g., the transaction manager and the database servers). This problem, called the *Atomic*

Commit problem (AC) [10], requires the processes to agree on a common outcome for the transaction: *commit* or *abort*. When every correct process (i.e., processes that do not crash) should eventually reach an outcome despite the failure of other processes, the problem is called *Non-Blocking Atomic Commit (NB-AC)* [14]. Solving this problem enables correct processes to relinquish resources (e.g., locks) without waiting for crashed processes to recover.

More precisely, the NB-AC problem consists for a set of processes to reach a common decision, *commit* or *abort*, according to some initial votes, *yes* or *no*, such that the following properties are satisfied:

- *Agreement:* No two processes decide differently.
- *Termination:* Every correct process eventually decides.
- *Abort-Validity: Abort* is the only possible decision if some process votes *no*.
- *Commit-Validity: Commit* is the only possible decision if every process is correct and votes *yes*.

This paper addresses this problem in asynchronous message passing distributed systems where channels are reliable, processes can only fail by crashing, and process failures can be suspected using failure detectors [2].

1.1 Consensus vs NB-AC

NB-AC resembles the well-known (binary) *Consensus* problem [5]. Both problems require the processes to reach one common decision, out of two possible ones, despite the crash of some of the processes. In Consensus, the processes need to decide on one out of two values, 0 or 1, based on proposed values, 0 or 1, such that the following properties are satisfied:

- *Agreement*: No two processes decide differently; [1]

- *Termination*: Every correct process eventually decides.

- *Validity*: The value decided must be a value proposed.

Besides their first glance similarities, there are indeed some differences between NB-AC and Consensus. To point out these differences, assume we translate in Consensus, "propose 0" to "vote *no*", "propose 1" to "vote *yes*", "decide 1" to "decide *commit*" and "decide 0" to "decide *abort*".

- In NB-AC, a process can only decide *commit* if *all* processes have voted *yes*. In Consensus, *commit* decision does not require that all votes be *yes*. This intuitively means that Consensus is not a special form of NB-AC.

- In NB-AC, the processes can all propose *yes* and yet decide *abort*. In Consensus, the processes would in this case have to decide *commit*. This intuitively means that NB-AC is not a special form of Consensus.

Many interesting theoretical results have been stated for Consensus (e.g., [5, 2, 3]) and it is not clear whether those results apply to NB-AC, precisely because of the differences pointed out above. The motivation of this work is to explore the applicability of those results to NB-AC.

1.2 FLP

In [5], Fisher, Lynch and Paterson (FLP) proved that Consensus cannot be solved (deterministically) in an asynchronous system if one process may crash. This

[1] We consider in this paper the *Uniform* variant of Consensus [11]. Uniform Consensus is a stronger variant of Consensus where even faulty processes are supposed to satisfy agreement. Considering Uniform Consensus (instead of Consensus) simplifies our discussion but does not impact the nature of our results.

impossibility result has been stated in the context of the *Weak Consensus* problem, defined with the *Agreement* and *Termination* properties of Consensus, plus the following *Weak Validity* property: both 0 and 1 are possible decision values, i.e., there is at least one run where the decision is 1 and one run where the decision is 0. NB-AC is a special form of Weak Consensus (any solution to NB-AC is also a solution to Weak Consensus) [2]. Consequently, NB-AC cannot be solved in an asynchronous system if one process may crash [11].

1.3 Circumventing FLP

Chandra and Toueg have shown in [2] that the FLP impossibility can be circumvented using failure detectors. More precisely, they have shown that Consensus can be solved (deterministically) in an asynchronous system augmented with the failure detector $\diamond\mathcal{S}$ *(Eventually Strong)* and the assumption of a majority of correct processes. Failure detector $\diamond\mathcal{S}$ guarantees *Strong Completeness*, i.e., eventually, every process that crashes is permanently suspected by every process, and *Eventual Weak Accuracy*, i.e., eventually, some correct process is never suspected. Failure detector $\diamond\mathcal{S}$ can however make an arbitrary number of mistakes, i.e., false suspicions.

Can we also circumvent the impossibility of solving NB-AC using some failure detector? The answer is of course "yes". The Three-Phase Commit (3PC) algorithms of Skeen [14] solve NB-AC with the failure detector $\mathcal{P}$ (*Perfect*). This failure detector ensures *Strong Completeness* (recalled above) and *Strong Accuracy*, i.e., no process is suspected before it crashes [2]. Failure detector $\mathcal{P}$ does never make any mistake and obviously provides more knowledge about failures than $\diamond\mathcal{S}$. A natural question then comes to mind: Can we circumvent the impossibility of

[2]Which is obviously not the case for Consensus.

solving NB-AC using $\diamond\mathcal{S}$?

We show in this paper that the answer to this question is "no". Failure detector $\diamond\mathcal{S}$ cannot solve NB-AC, even if only one process may crash. Does this mean that NB-AC is strictly harder than Consensus? i.e., does NB-AC require more knowledge about failures than Consensus? We show that the answer to this question is also "no". We exhibit a failure detector $\mathcal{B}$ *("Stillborn" detector)* that solves NB-AC and we show that $\mathcal{B}$ cannot be transformed into $\diamond\mathcal{S}$ (if we assume that at least two processes can crash in a system with a majority of correct processes). Informally, failure detector $\mathcal{B}$ is defined as follows: in runs of the system where no process initially crashes (i.e., no process crashes at time 0), $\mathcal{B}$ guarantees *Strong Completeness* and *Strong Accuracy* (i.e., $\mathcal{B}$ behaves like $\mathcal{P}$); in runs where some process initially crashes (i.e., some process is *"stillborn"*), $\mathcal{B}$ detects the initial crash by permanently outputting, at every process p_i, the process p_i itself. Hence, processes accurately know if some process has initially crashed: they do not necessarily know which one, nor do they know how many of such processes have crashed. In these runs (with at least one "stillborn" process), the processes can safely decide *abort*: NB-AC becomes trivial and there is no need for $\diamond\mathcal{S}$. An interesting consequence of the existence of $\mathcal{B}$ is that Consensus and NB-AC are *incomparable* (if we assume that at least two processes can crash in a system with a majority of correct processes).

1.4 The Anonymously Perfect Failure Detector

Exhibiting failure detector $\mathcal{B}$ enables us in particular to show that there are circumstances under which we can solve NB-AC but not Consensus.

Failure detector $\mathcal{B}$ is however *bogus* in the following sense. It does not really encapsulate the synchrony of the underlying system, i.e., it cannot be implemented even in a perfectly synchronous system. However, $\mathcal{B}$ is a valid failure detector according to the original failure detector definition [2]. It is hence natural to ask the following question: if we exclude such *bogus* failure detectors, is $\mathcal{P}$ the weakest failure detector to solve NB-AC?

We show that the answer to this question is "no". We introduce the *Anonymously Perfect* failure detector $?\mathcal{P}$, which we show is necessary for NB-AC. This failure detector satisfies the two following properties: *Anonymous Accuracy*, i.e., no crash is detected unless *some* process crashes; and *Anonymous Completeness*, i.e., if a crash occurs then, eventually, every correct process permanently detects that *some* crash occurs.

We show that failure detector $\diamond\mathcal{S}+?\mathcal{P}$, while not *bogus*, is strictly weaker than $\mathcal{P}$ ($\diamond\mathcal{S}+?\mathcal{P}$ provides less knowledge about failures than $\mathcal{P}$) and yet it solves NB-AC. Interestingly, the structure of failure detector $\diamond\mathcal{S}+?\mathcal{P}$ provides some insight about the practical solvability of NB-AC. Intuitively, the *Commit-Validity* property of NB-AC is related to $?\mathcal{P}$ and the *Termination* property is related to $\diamond\mathcal{S}$. This observation suggests the use of two time-out values for a practical failure detection implementation. The first time-out value (underlying $?\mathcal{P}$) should be large enough to reduce the risk of aborting transactions because of false suspicions, whereas the second time-out value (underlying $\diamond\mathcal{S}$) should be small enough to decrease the fail-over time.

Having shown however that $\diamond\mathcal{S}$ is not necessary for NB-AC means that $\diamond\mathcal{S}+?\mathcal{P}$ is not the weakest failure detector to solve NB-AC. So what is indeed that failure detector? We leave this question open.

1.5 Roadmap

Section 2 defines the system model we consider. Section 3 shows that Consensus and NB-AC are incomparable if at least two processes can crash but a majority are correct. Section 4 introduces failure detector $?\mathcal{P}$. We show that, to solve NB-AC, $?\mathcal{P}$ is necessary and $?\mathcal{P} + \diamond\mathcal{S}$ is sufficient with a majority of correct processes. We also show that $\mathcal{P}$ is strictly stronger than $?\mathcal{P} + \diamond\mathcal{S}$ if at least two processes can crash in a system of at least three processes. Section 5 draws, from the very structure of $?\mathcal{P} + \diamond\mathcal{S}$, some practical observations on the solvability of NB-AC. Section 6 discusses some related work and summarizes the main contributions of the paper.

2 System Model

Our model of asynchronous computation with failure detection is the FLP model [5] augmented with the failure detector abstraction [2, 3]. [3] A discrete global clock is assumed, and Φ, the range of the clock's ticks, is the set of natural numbers. The global clock is used for presentation simplicity and is not accessible to the processes. We sketch here the fundamentals of the model that are needed for our results. The reader interested in specific details about the model should consult [3].

2.1 Failure patterns and environments

We consider a distributed system composed of a finite set of n processes $\Omega = \{p_1, p_2, \ldots, p_n\}$. A process p_i is said to *crash at time* t if p_i does not perform any *action* after time t (the notion of *action* is recalled below). Failures are permanent, i.e., no process *recovers* after a crash. A *correct* process is a process that does not

[3] For presentation simplicity, we do not distinguish in this paper failure detectors and failure detector classes [3].

crash. A *failure pattern* is a function F from Φ to 2^{Ω}, where $F(t)$ denotes the set of processes that have crashed through time t. The set of correct processes in a failure pattern F is noted $correct(F)$. For instance, the *failure-free* pattern is the failure pattern where no process crashes, i.e., $correct(F) = \Omega$. An *environment* E is a set of failure patterns. Environments describe the crashes that can occur in a system. When stating a result, we shall explicitely specify the environment for which the result applies, unless it applies to all environments.

2.2 Failure detectors

Roughly speaking, a failure detector $\mathcal{D}$ is a distributed oracle which gives hints about failure patterns. Each process p_i has a local failure detector module of $\mathcal{D}$, denoted by $\mathcal{D}_i$. Associated with each failure detector $\mathcal{D}$ is a range $R_{\mathcal{D}}$ of values output by the failure detector. A *failure detector history* H with range R is a function H from $\Omega \times \Phi$ to R. For every process $p_i \in \Omega$, for every time $t \in \Phi$, $H(p_i, t)$ denotes the value of the failure detector module of process p_i at time t, i.e., $H(p_i, t)$ denotes the value output by $\mathcal{D}_i$ at time t. A *failure detector* $\mathcal{D}$ is a function that maps each failure pattern F to a set of failure detector histories with range $R_{\mathcal{D}}$. $\mathcal{D}(F)$ denotes the set of possible failure detector histories permitted for the failure pattern F, i.e., each history represents a possible behaviour of $\mathcal{D}$ for the failure pattern F. The failure detectors introduced in [2] all have a range $R = 2^{\Omega}$. For any such failure detector $\mathcal{D}$, any failure pattern F and any history H in $\mathcal{D}(F)$, $H(p_i, t)$ is the set of processes *suspected* by process p_i at time t.

It is important to notice that the asynchronous system model augmented with the abstraction of failure detector defines a general system model. Every failure detector actually defines a specific system model.

2.3 Algorithms

An *algorithm* using a failure detector $\mathcal{D}$ is a collection A of n deterministic automata A_i (one per process p_i). Computation proceeds in steps of the algorithm. In each step of an algorithm A, a process p_i atomically performs the following three actions: (1) p_i receives a message from some process p_j, or a "null" message λ; (2) p_i queries and receives a value d from its failure detector module $\mathcal{D}_i$ ($d \in R_{\mathcal{D}}$ is said to be *seen* by p_i); (3) p_i changes its state and sends a message (possibly null) to some process. This third action is performed according to (a) the automaton A_i, (b) the state of p_i at the beginning of the step, (c) the message received in action 1, and (d) the value d seen by p_i in action 2. The message received by a process is chosen non-deterministically among the messages in the message buffer destined to p_i, and the null message λ. A *configuration* is a pair (I, M) where I is a function mapping each process p_i to its local state, and M is a set of messages currently in the message buffer. A configuration (I, M) is an initial configuration if $M = \emptyset$ (no message is initially in the buffer): in this case, the states to which I maps the processes are called *initial states*. A *step* of an algorithm A is a tuple $e = (p_i, m, d, A)$, uniquely defined by the algorithm A, the identity of the process p_i that takes the step, the message m received by p_i, and the failure detector value d seen by p_i during the step. A step $e = (p_i, m, d, A)$ is *applicable to a configuration* (I, M) if and only if $m \in M \cup \{\lambda\}$. The *unique* configuration that results from applying e to configuration $C = (I, M)$ is noted $e(C)$.

2.4 Schedules and runs

A *schedule* of an algorithm A is a (possibly infinite) sequence $S = S[1]; S[2]; \ldots S[k]; \ldots$ of steps of A. A schedule S is applicable to a configuration C if (1) S is the empty schedule, or (2) $S[1]$ is applicable to C, $S[2]$ is applicable to $S[1](C)$

(the configuration obtained from applying $S[1]$ to C), etc.

Let A be any algorithm, $\mathcal{D}$ any failure detector and E any environment. A *partial run* of A *using* $\mathcal{D}$ for E is a tuple $R = \langle F, H, C, S, T \rangle$ where F is a failure pattern in E, H is a failure detector history in $\mathcal{D}(F)$, C is an initial configuration of A, T is a finite sequence of increasing time values, and S is a finite schedule of A such that, (1) $|S| = |T|$, (2) S is applicable to C, and (3) for all $k \leq |S|$ where $S[k] = (p_i, m, d, A)$, we have $p_i \notin F(T[k])$ and $d = H(p_i, T[k])$.

A *run* of A *using* $\mathcal{D}$ for E is a tuple $R = \langle F, H, C, S, T \rangle$ where F is a failure pattern in E, H is a failure detector history in $\mathcal{D}(F)$, C is an initial configuration of A, S is an infinite schedule of A, T is an infinite sequence of increasing time values, and in addition to the conditions above of a partial run ((1), (2) and (3)), the two following conditions are satisfied: (4) every correct process takes an infinite number of steps, and (5) every message sent to a correct process p_j is eventually received by p_j.

Let $R_1 = \langle F_1, H_1, C_1, S_1, T_1 \rangle$ be any partial run of some algorithm A, and $R_2 = \langle F_2, H_2, C_2, S_2, T_2 \rangle$ any run of A. We say that R_2 is an *extension* of R_1 if $C_1 = C_2$, $\forall t \leq T_1[|T_1|]$, $\forall p_i \in \Omega$: $F_2(t) \subseteq F_1(t)$ and $H_1(p_i, t) = H_2(p_i, t)$, and $\forall i, 1 \leq i \leq |T_1|$: $S_1[i] = S_2[i]$ and $T_1[i] = T_2[i]$. We also say that R_1 is a *partial run* of R_2.

2.5 Solvability

An algorithm A *solves* a problem B using a failure detector $\mathcal{D}$ for environment E if every run of A using $\mathcal{D}$ for E satisfies the specification of B. We say that $\mathcal{D}$ solves B for E if there is an algorithm that solves B using $\mathcal{D}$ for E. We say that a failure detector $\mathcal{D}1$ is *stronger* than a failure detector $\mathcal{D}2$ ($\mathcal{D}2 \preceq \mathcal{D}1$) for environment E if there is an algorithm that *transforms* $\mathcal{D}1$ into $\mathcal{D}2$ for E [2], i.e., that *emulates*

$\mathcal{D}2$ with the outputs of $\mathcal{D}1$ for E [2]. We also say that $\mathcal{D}2$ is *weaker* than $\mathcal{D}1$ for E. We say that $\mathcal{D}1$ is *strictly stronger* than $\mathcal{D}2$ ($\mathcal{D}2 \prec \mathcal{D}1$) for E if $\mathcal{D}2 \preceq \mathcal{D}1$ and $\mathcal{D}1 \not\preceq \mathcal{D}2$ for E. We also say that $\mathcal{D}2$ is *strictly weaker* than $\mathcal{D}1$ for E. We say that a failure detector $\mathcal{D}$ is the weakest to solve a problem B for environment E if (a) $\mathcal{D}$ solves B for E and (b) any failure detector that solves B is stronger than $\mathcal{D}$ for E.

Finally, we say that a problem B_1 is *harder* than a problem B_2 for environment E, if any failure detector that solves B_1 for E also solves B_2 for E. If B_1 is not harder than B_2 for E and B_2 is not harder than B_1 for E, then we say that B_1 and B_2 are *incomparable* for E.

3 NB-AC vs Consensus

We show below that there is an environment where NB-AC and Consensus are *incomparable*. That is, we show that there is an environment where NB-AC is not harder than Consensus and Consensus is not harder than NB-AC.

We first show that NB-AC cannot be solved with failure detector $\diamond\mathcal{S}$ if at least one process can crash in a system of at least two processes (Lemma 3.1). As $\diamond\mathcal{S}$ solves Consensus if a majority of the processes are correct [2], then Consensus is not harder than NB-AC if at least one process can crash but a majority of the processes are correct.

Then we exhibit a failure detector $\mathcal{B}$ that solves NB-AC (Lemma 3.3), and we show that $\mathcal{B}$ is not stronger than $\diamond\mathcal{S}$ if two processes can crash but a majority are correct (Lemma 3.4). Since $\diamond\mathcal{S}$ was shown to be the weakest failure detector to solve Consensus [3] if a majority of the processes are correct [3], then NB-AC is not harder than Consensus for any environment where two processes can crash but a majority are correct. For any such environment, NB-AC and Consensus are hence incomparable.

3.1 Consensus is not harder than NB-AC

Lemma 3.1 *$\diamond\mathcal{S}$ does not solve NB-AC for any environment where at least one process can crash in a system of at least two processes.*

PROOF. (By contradiction).[4] We assume that there is an algorithm that solves NB-AC using $\diamond\mathcal{S}$ for an environment where at least one process can crash in a system of at least two processes. We then exploit the fact that $\diamond\mathcal{S}$ does not provide accurate information about the crash of every process (i.e., it can make an arbitrary number of false failure detections) to exhibit a contradiction. Basically, we first consider a run where all processes vote *yes* but one specific process p_1 initially crashes. We show that in this run, a correct process $p_2 \neq p_1$ has no choice then eventually deciding *abort* (because p_1 could have voted *no*). Second, we show that p_2 cannot, using $\diamond\mathcal{S}$, distinguish this run from one where p_1 is correct but just slow (and where p_2 should have decided *commit*). The actual detailed proof is given below.

Consider any algorithm A that solves NB-AC using failure detector $\diamond\mathcal{S}$ for an environment where at least one process can crash in a system of at least two processes. Consider any run $R = \langle F, H, C, S, T\rangle$ of A where all processes vote *yes*, one process p_1 crashes immediately at time 0 (without sending any message), and all other processes are correct. Consider correct process p_2. By the *Termination* property of NB-AC is violated in R, p_2 decides in R. Let $R_1 = \langle F, H, C, S_1, T_1\rangle$ be any partial run of R where p_2 decides. There are two cases to consider: p_2 decides (1) *commit* or (2) *abort*.

[4]This proof is from [7], and the underlying intuitive idea actually originates from [4].

Consider case (1): process p_2 decides *commit* in R_1. We construct a partial run $R_2 = \langle F, H, C', S_1, T_1 \rangle$, similar to R_1, except that in R_2, process p_1 votes *no*. Since R_1 is a partial run of A and p_1 crashes at time 0, we have: (1) $|S_1| = |T_1|$, (2) S_1 is applicable to C' (because S_1 is applicable to C and p_1 crashes at time 0), and (3) for all $i \leq |S_1|$ where $S_1[i] = (p_j, m, d, A)$, $p_j \notin F(T_1[i])$ and $d = H(j, T_1[i])$. R_2 is hence also a partial run of A. Consider R_2' any run that is an extension of R_2. In R_2', process p_1 votes *no* and process p_2 decides *commit*, violating the *Abort-Validity* property of NB-AC: a contradiction.

Consider now case (2): process p_2 decides *abort* in R_1. Let F' be the failure-free pattern: all processes are correct in F'. Let H' be the failure detector history that is similar to H until time $T_1[|T_1|]$, and after time $T_1[|T_1|]$, no process is ever suspected. Obviously, H' belongs to $\diamond\mathcal{S}(F')$. Consider now partial run $R_3 = \langle F', H', C, S_1, T_1 \rangle$. We have: (1) $|S_1| = |T_1|$, (2) S_1 is applicable to C, and (3) for all $i \leq |S_1|$ where $S_1[i] = (p_j, m, d, A)$, $p_j \notin F'(T_1[i])$ and $d = H'(j, T_1[i])$ (all processes are correct in F', and until time $T_1[|T_1|]$, H and H' give the same outputs). R_3 is thus also a partial run of A. Consider R_3' any run that is an extension of R_3. In R_3', process p_2 decides *abort*. Since, in R_3', all processes are correct, they all vote *yes*, and p_2 decides *abort*, then the *Commit-Validity* of NB-AC is violated: a contradiction. [5] □

Proposition 3.2 *Consensus is not harder than NB-AC for any environment where at least one process can crash in a system where a majority are correct.*

PROOF. Follows from Lemma 3.1 and the fact that $\diamond\mathcal{S}$ solves Consensus for any environment where a majority of the processes are correct. □

[5]Note that this proof applies to a stronger lemma with failure detector $\diamond\mathcal{P}$ [2] (instead of $\diamond\mathcal{S}$). One could also construct a similar proof with failure detector $\mathcal{S}$ [2].

3.2 NB-AC is not harder than Consensus

We introduce here the *Stillborn* failure detector, denoted by $\mathcal{B}$. Each module of $\mathcal{B}$ outputs a subset of the processes in Ω, i.e., $R_{\mathcal{B}} = 2^{\Omega}$. When $\mathcal{B}$ outputs a process p_j at a process p_i and $i \neq j$, we say that p_i *suspects* p_j. When $\mathcal{B}$ outputs p_i at p_i, we say that p_i *detects a crash*.

For every failure pattern F where at least one process crashes at time 0 (i.e., $F(0) \neq \emptyset$), $\mathcal{B}(F)$ contains exactly one history H, and H satisfies the following property: every process permanently detects the crash. More precisely:

- $\forall t \in \Phi, \forall p_i \in \Omega : H(p_i, t) = \{p_i\}$.

For every failure pattern F where no process crashes at time 0 (i.e., $F(0) = \emptyset$), $\mathcal{B}(F)$ is the set of histories H that satisfy *Strong Completeness* and *Strong Accuracy* [2], i.e., in such failure patterns, $\mathcal{B}$ behaves like the *Perfect* failure detector $\mathcal{P}$ [2].

Lemma 3.3 *$\mathcal{B}$ solves NB-AC.*

PROOF (SKETCH). We give here a brief description of an algorithm that solves NB-AC using $\mathcal{B}$. The algorithm uses the 3PC algorithm of Skeen [14] as an underlying component. (3PC solves NB-AC with $\mathcal{P}$.) The basic idea of our algorithm is the following. Every process p_i first consults its failure detector module $\mathcal{B}_i$. If $\mathcal{B}_i$ outputs p_i (i.e., some process has initially crashed), then process p_i immediately decides *abort*. Process p_i can safely do so thanks to the definition of failure detector $\mathcal{B}$. Otherwise, if $\mathcal{B}_i$ does not output $\{p_i\}$, then p_i executes a 3PC. It is easy to see that the properties of NB-AC are satisfied in every run of our algorithm. $\square$

Lemma 3.4 *$\mathcal{B} \not\succeq \Diamond\mathcal{S}$ for any environment where at least two processes can crash but a majority are correct.*

Proof: The proof is by contradiction. We consider an environment E where at least two processes can crash but a majority are correct. We assume that there is an algorithm $T_{\mathcal{B}\rightarrow\diamond\mathcal{S}}$ that transforms $\mathcal{B}$ into $\diamond\mathcal{S}$ and then we exhibit the following contradiction: we show that we can build, out of $T_{\mathcal{B}\rightarrow\diamond\mathcal{S}}$, an algorithm that solves Consensus in an asynchronous system for an environment where at least one process can crash but a majority are correct: contradicting the FLP impossibility [5]. We proceed in four steps. We first introduce a failure detector denoted by $\mathcal{HK}$: a characteristic of $\mathcal{HK}$ is that it can be implemented in a pure asynchronous system. In the second step, we show that $T_{\mathcal{B}\rightarrow\diamond\mathcal{S}}$ transforms $\mathcal{HK}$ into $\diamond\mathcal{S}$ in the environment E_1 defined as the subset of failure patterns in E where process p_1 crashes at time 0. In the third step, we derive from $T_{\mathcal{B}\rightarrow\diamond\mathcal{S}}$ an algorithm A that implements $\diamond\mathcal{S}$ within $\Omega \setminus \{p_1\}$ for an environment E_2 where one process can crash but a majority are correct. In the fourth step, we point out the fact that A can be used to solve Consensus in $\Omega \setminus \{p_1\}$ for an environment where one process can crash but a majority are correct - contradicting the FLP impossibility [5]. We detail these steps in the following.

1. Failure detector $\mathcal{HK}$ is defined as follows. For every failure pattern and at every process p_i, the local module $\mathcal{HK}_i$ permanently outputs p_i.[6] Obviously, failure detector $\mathcal{HK}$ can be implemented in a pure asynchronous system model.

2. Consider algorithm $T_{\mathcal{B}\rightarrow\diamond\mathcal{S}}$. Consider environment E_1, included in E, and defined as the subset of failure patterns in E where process p_1 initially crashes, i.e., p_1 crashes at time 0. Since E_1 is included in E, then $T_{\mathcal{B}\rightarrow\diamond\mathcal{S}}$ transforms $\mathcal{B}$ into $\diamond\mathcal{S}$ for E_1. For every failure pattern in E_1, module $\mathcal{B}_i$ permanently outputs p_i (by the definition of failure detector $\mathcal{B}$). Hence, for E_1,

[6] $\mathcal{HK}$ stands for *Hara-Kiri*.

$\mathcal{HK}$ outputs exactly the same values as $\mathcal{B}$. Consequently, $T_{\mathcal{B}\to\Diamond\mathcal{S}}$ transforms $\mathcal{HK}$ into $\Diamond\mathcal{S}$ for E_1.

3. Consider the subset of Ω, $\Omega \setminus \{p_1\}$. Let E_2 be the set of failure patterns F_2 of $\Omega \setminus \{p_1\}$ such that there is a failure pattern F_1 in E_1 and $F_2(t) = F_1(t) \setminus \{p_1\}$ for any time t in Φ. Since E_1 is the set of failure patterns of Ω where (a) p_1 initially crashes, (b) another process can crash (at any time), and (c) a majority are correct, then E_2 is the set of failure patterns of $\Omega \setminus \{p_1\}$ where one process can crash and a majority are correct.

   We denote by *output*$(\Diamond\mathcal{S})$ the variable used by $T_{\mathcal{B}\to\Diamond\mathcal{S}}$ to emulate failure detector $\Diamond\mathcal{S}$. Define the variable *output*$'(\Diamond\mathcal{S})$ as *output*$(\Diamond\mathcal{S}) \setminus \{p_1\}$. Note that both *output*$(\Diamond\mathcal{S})$ and *output*$'(\Diamond\mathcal{S})$ are distributed variables: each process has its own copies of the variables.

   Let F_2 be any failure pattern in E_2: by definition, there is a failure pattern F_1 in E_1 such that $F_2(t) = F_1(t) \setminus \{p_1\}$ for any time t in Φ. In F_1, eventually *output*$(\Diamond\mathcal{S})$ permanently contains, at every correct process of Ω, every crashed process of Ω: hence, in F_2, eventually, *output*$'(\Diamond\mathcal{S})$ permanently contains, at every correct process of $\Omega \setminus \{p_1\}$, every crashed process of $\Omega \setminus \{p_1\}$ (*Strong Completeness*). Moreover, in F_1, there is a correct process p_i in Ω such that eventually, *output*$(\Diamond\mathcal{S})$ does never contain p_i at any correct process in Ω. Since, p_i cannot be p_1 in F_1 (because p_1 is never correct in E_1), then in F_2, there is a correct process p_i in $\Omega \setminus \{p_1\}$ such that eventually, *output*$'(\Diamond\mathcal{S})$ does never contain p_i at any correct process in $\Omega \setminus \{p_1\}$ (*Eventual Weak Accuracy*). Hence *output*$'(\Diamond\mathcal{S})$ emulates $\Diamond\mathcal{S}$ in $\Omega \setminus \{p_1\}$ for environment E_2.

   We can thus derive from $T_{\mathcal{B}\to\Diamond\mathcal{S}}$ an algorithm A that implements $\Diamond\mathcal{S}$ in $\Omega \setminus \{p_1\}$ for the environment E_2 where one process can crash and a majority of the processes are correct.

4. Algorithm A implements $\diamond\mathcal{S}$ in environment E_2 where one process can crash but a majority of the processes are correct. In this environment, algorithm A, together with the Consensus algorithm of [2] using $\diamond\mathcal{S}$, solves Consensus: a contradiction with [5]. □

Proposition 3.5 *NB-AC is not harder than Consensus for any environment where at least two processes can crash but a majority are correct.*

PROOF. Follows from from Lemma 3.3, Lemma 3.4 and the fact that $\diamond\mathcal{S}$ is the weakest failure detector to solve Consensus for any environment where a majority of the processes are correct [3]. □

The following corollary follows from Proposition 3.2 and Proposition 3.5.

Corollary 3.6 *NB-AC and Consensus are incomparable for any environment where at least two processes can crash but a majority are correct.*

4 The Anonymously Perfect Failure Detector

We have introduced failure detector $\mathcal{B}$ to show that NB-AC and Consensus are incomparable. A closer look at the definition of $\mathcal{B}$ reveals that this failure detector is somehow *bogus*. Although it does indeed comply with the original failure detector definition of [2], $\mathcal{B}$ does not *encapsulate* the synchrony of the system, as *good* failure detectors should do. Failure detector $\mathcal{B}$ cannot be implemented even in a completely synchronous system. Consequently, solutions to NB-AC based on $\mathcal{B}$, including the one we considered in Section 3, are somehow *bogus* as well.

As we pointed out in the introduction, NB-AC can be solved with the *Perfect* failure detector $\mathcal{P}$, which can indeed be implemented in a synchronous system. One might naturally wonder whether $\mathcal{P}$ is indeed the weakest failure detector for NB-AC, among failure detectors that are implementable in a synchronous system. We show in the following that the answer is "no" and we derive an interesting observation on the practical solvability of NB-AC.

We define the *Anonymously Perfect* failure detector $?\mathcal{P}$, which is weaker than $\mathcal{P}$. We show that, to solve NB-AC, (1) $?\mathcal{P}$ is necessary (for any environment); (2) $?\mathcal{P} + \diamond\mathcal{S}$ is sufficient for any environment with a majority of correct processes. We then show that (3) $\mathcal{P}$ is strictly stronger than $?\mathcal{P} + \diamond\mathcal{S}$ for any environment where at least two processes can crash in a system of at least three processes.

4.1 Anonymous Perfection

Each module of failure detector $?\mathcal{P}$ outputs either the empty set or the singleton $\{\top\}$. When failure detector module $?\mathcal{P}_i$ outputs $\{\top\}$, we say that p_i *detects a crash*. For each failure pattern F, $?\mathcal{P}(F)$ is the set of failure detector histories H that satisfy the following properties:

- *Anonymous Completeness:* If some process crashes, then there is a time after which every correct process permanently detects a crash. More precisely:
  - $\text{crashed}(F) \neq \emptyset \Rightarrow \exists t \in \Phi, \forall t' \geq t, \forall p_i \in \text{correct}(F),\ H(p_i, t') = \{\top\}$.
- *Anonymous Accuracy:* No crash is detected unless some process crashes. More precisely:
  - $\forall t \in \Phi, \forall p_i \in \Omega$: $H(p_i, t) \neq \emptyset \Rightarrow \text{crashed}(F) \neq \emptyset$

Note that *Anonymous Completeness* does not require the actual crashed process to be eventually suspected. It only requires that, if some process crashes, then

/ Algorithm executed by every process p_i */*

$k := 1;$

$output(?\mathcal{P})_i = \emptyset;$

while (nbac(k, yes) = $commit$) **do**

  $k := k + 1;$

$output(?\mathcal{P})_i = \{\top\};$

Figure 1: Emulating $?\mathcal{P}$ using NB-AC.

eventually, the failure detector module of every correct process p_i keeps permanently outputting $\{\top\}$. If a process p_i outputs $\top$, then p_i does accurately know that some process has crashed or will crash, but p_i does not necessarily know which one (or how many processes crash). Note also that *Anonymous Accuracy* does not preclude the possibility for a crash to be detected *before* any process has actually crashed.

4.2 The Necessary Condition

We show here that if a failure detector $\mathcal{D}$ solves NB-AC then $\mathcal{D}$ can be transformed into $?\mathcal{P}$. We give an algorithm in Figure 1 that uses NB-AC to emulate, within a distributed variable $output(?\mathcal{P})$, the behaviour of failure detector $?\mathcal{P}$.

We assume the existence of a function *nbac()*. Different instances of this function are distinguished with an integer k. Each process p_i has a local copy of $output(?\mathcal{P})$, denoted by $output(?\mathcal{P})_i$, which provides the information that should be given by the local failure detector module of $?\mathcal{P}$ at process p_i.

The basic idea of our algorithm is the following. The value of $output(?\mathcal{P})_i$ is initially set to $\emptyset$. Every process p_i performs a sequence of rounds $1, ..k, ...$ Within each round k, p_i invokes $nbac(k, yes)$ and waits until a decision is returned. If p_i

decides *commit*, then p_i directly moves to the next round. Otherwise, p_i puts $\top$ into $output(?\mathcal{P})_i$.

Lemma 4.1 *The algorithm of Figure 1 uses NB-AC to implement $?\mathcal{P}$.*

PROOF: We show below that $output(?\mathcal{P})$ satisfies the *Anonymous Completeness* and *Anonymous Accuracy* properties of $?\mathcal{P}$.

1. Consider completeness. Let p_j be any process that crashes at time t and let p_i be any correct process. Assume by contradiction that p_i does never put $\top$ into $output(?\mathcal{P})_i$. This means that p_i remains in the while loop of the algorithm. By the *Termination* property of NB-AC, p_i does never remain blocked forever waiting for the $nbac()$ function to return. This means that p_i keeps indefinitely incrementing k, invoking $nbac(k, yes)$ and deciding *commit*.

   Since p_j crashes at time t, there is an integer k such that for every $k' \geq k$, p_j does not invoke instance k' of NB-AC: in particular, p_j does not vote *yes* for that instance of NB-AC. By the *A-Validity* properties of of NB-AC, instance k' of NB-AC returns *abort*: a contradiction.

   By the algorithm, once p_i puts $\top$ into $output(?\mathcal{P})_i$, p_i does never change the value of $output(?\mathcal{P})_i$. Hence, there is a time after which $output(?\mathcal{P})_i$ permanently contains $\top$, which means that $output(?\mathcal{P})$ satisfies *Anonymous Completeness*.

2. Consider now accuracy. Let p_i be any process and assume that $output(?\mathcal{P})_i$ contains $\top$ at a time t. By the algorithm of Figure 1, this can only be possible if some instance k of $nbac()$ returns *abort* at process p_i. Given that all processes propose *yes*, then by the *A-Validity* property of NB-AC, *abort*

can be returned only if some process crashes. Hence *output*($?\mathcal{P}$) satisfies *Anonymous Accuracy.* □

The following proposition follows directly from Lemma 4.1.

Proposition 4.2 *If any failure detector $\mathcal{D}$ solves NB-AC, then $?\mathcal{P} \preceq \mathcal{D}$.*

4.3 The Sufficient Condition

Figure 2 describes a simple algorithm that transforms Consensus into NB-AC using failure detector $?\mathcal{P}$. The algorithm uses Consensus as a black-box, represented by a function *consensus()*: a process calls the function with *commit* or *abort* as a parameter, and the function eventually returns *commit* or *abort*. The function satisfies the *Agreement, Termination* and *Validity* properties of Consensus.

The basic idea of our NB-AC algorithm of Figure 2 is the following. Every process sends its vote to all participants (including itself). A process that either receives a vote *no* or detects a crash, invokes *consensus()* with *abort*, and decides the outcome returned by *consensus()*. A process that receives *yes* votes from all processes, invokes *consensus()* with *commit*, and decides the outcome returned by *consensus()*.

We assume that every process p_i, either crashes, or invokes *nbac()* in Figure 2. The vote of participant p_i is denoted $vote_i$. We assume that, at every process p_i, $vote_j$ is initialised to *nil*. Function *nbac()* terminates by the execution of a "**return** *outcome*" statement, where *outcome* is either *commit* or *abort*: when p_i executes **return** *outcome*, we consider that p_i decides *outcome*.

Lemma 4.3 *The algorithm of Figure 2 uses $?\mathcal{P}$ to transform Consensus into NB-AC.*

```
function nbac(vote_i)   /* Algorithm executed by every process p_i */
    send (p_i, vote_i) to all
        wait until [(for j = 1 to n: received (p_j, vote_j)) or ⊤ ∈ ?P_i] ;
            if (⊤ ∈ ?P_i or ∃j ∈ [1, n] s.t.: vote_j = no) then
                outcome_i := consensus(abort) ;
            else
                outcome_i := consensus(commit) ;
    return outcome_i ;
```

Figure 2: Transforming Consensus into NB-AC with $?\mathcal{P}$.

Proof. We consider the properties of NB-AC separately.

- *Agreement.* Any participant that decides *outcome* must have decided *outcome* through *consensus()*. By the *Agreement* property of Consensus, no two processes can decide differently, which implies the *Agreement* property of NB-AC.

- *Abort-Validity.* By the *Validity* property of Consensus, a process decides *commit* only if some process p_i has invoked *consensus()* with *commit*. To do so, p_i must have received *yes* votes from all. Hence *A-Validity* of NB-AC is satisfied.

- *Commit-Validity.* If no process crashes, then by the *Anonymous Accuracy* property of $?\mathcal{P}$, no process detects a crash. If in addition all processes vote *yes*, then no process proposes *abort* to *consensus()*. By the *Validity* property of Consensus, no process will be able to decide *abort*, which implies the *C-Validity* property of NB-AC.

- *Termination.* Consider now the *Termination* property of NB-AC. Let p_i be any correct process. If p_i receives votes from all, then p_i invokes *consensus()*. Otherwise, by the assumption that all correct processes execute the algorithm of Figure 2, and the assumption of reliable channels, the only reason for a process p_i not to receive the vote of a process p_j, is if p_j crashes. In this case, by the *Anonymous Completeness* property of $?\mathcal{P}$, p_i eventually detects the crash and invokes *consensus()*. Hence every correct process eventually invokes *consensus()*. By the *Termination* property of Consensus, every correct process eventually decides *commit* or *abort* which ensures the *Termination* property of NB-AC. $\square$

We define here the failure detector $?\mathcal{P} + \diamond\mathcal{S}$. Each module of $?\mathcal{P} + \diamond\mathcal{S}$ outputs a subset of $\Omega \cup \{\top\}$. Failure detector $?\mathcal{P} + \diamond\mathcal{S}$ satisfies the *Anonymous Completeness* and *Anonymous Accuracy* properties of $?\mathcal{P}$, together with the *Strong Completeness* and *Eventual Weak Accuracy* properties of $\diamond\mathcal{S}$. Since Consensus is solvable with $\diamond\mathcal{S}$ [2] for any environment with a majority of correct processes, then the following proposition follows from Lemma 4.3:

Proposition 4.4 *$?\mathcal{P} + \diamond\mathcal{S}$ solves NB-AC for any environment with a majority of correct processes.*

4.4 Anonymous Perfection is not Perfection

Obviously, failure detector $\mathcal{P}$ can be used to emulate $?\mathcal{P} + \diamond\mathcal{S}$ for any environment, i.e., $?\mathcal{P} + \diamond\mathcal{S} \preceq \mathcal{P}$. We state in the following that the converse is not true for any environment where at least two processes can crash in a system of at least three processes.

Proposition 4.5 *$\mathcal{P} \not\preceq\ ?\mathcal{P} + \diamond\mathcal{S}$ for any environment where at least one process can crash in a system of at least three processes.*

PROOF: (By contradiction). We assume that there is an algorithm $T_{?\mathcal{P}+\diamond\mathcal{S}\to\mathcal{P}}$ that transforms $?\mathcal{P} + \diamond\mathcal{S}$ into failure detector $\mathcal{P}$. Then we use the fact that $\mathcal{P}$ satisfies *Strong Completeness* to show that it does not satisfy *Strong Accuracy*: a contradiction. We denote by $output(\mathcal{P})$ the variable used by $T_{?\mathcal{P}+\diamond\mathcal{S}\to\mathcal{P}}$ to emulate failure detector $\mathcal{P}$ ($output(\mathcal{P})_i$ denotes the value of that variable at a given process p_i).

Consider three different processes p_1, p_2 and p_3 in Ω. Let F_1 be the failure pattern where process p_1 crashes at time 0 and no other process crashes. Let H be the failure detector history where all processes permanently output $\{\top, p_1, p_2\}$. Clearly, H belongs to $?\mathcal{P} + \diamond\mathcal{S}(F_1)$.

Since variable $output(\mathcal{P})$ satisfies *Strong Completeness*, then there is a partial run of $T_{?\mathcal{P}+\diamond\mathcal{S}\to\mathcal{P}}$, $R_1 = \langle F_1, H, C, S, T\rangle$, such that, at $T[|T|]$, $\{p_1\} \subset output(\mathcal{P})_3$.

Now consider F_2 the failure pattern where process p_2 crashes at time $T[|T|]+1$ and no other process crashes. Clearly, H belongs to $?\mathcal{P} + \diamond\mathcal{S}(F_2)$.

Since $?\mathcal{P}+\diamond\mathcal{S}$ outputs exactly the same values in F_1 and in F_2 (history H), then $R_2 = \langle F_2, H, C, S, T\rangle$ is also a partial run of $T_{?\mathcal{P}+\diamond\mathcal{S}\to\mathcal{P}}$. Indeed: (1) $|S| = |T|$, (2) S is applicable to C, and (3) for all $i \leq |S|$ where $S[i] = (p_j, m, d, A)$, since $p_j \notin F_1(T[i])$ and $d = H(j, T[i])$, we have $p_j \notin F_2(T[i])$ and $d = H(j, T[i])$. In R_2, at $T[|T|]$, $p_1 \in output(\mathcal{P})_3$ and $p_1 \in correct(F_2)$, which means that $\mathcal{P}$ violates *Strong Accuracy*: a contradiction. $\square$

5 A Practical Observation

We draw here some practical observations from the very structure of $?\mathcal{P} + \diamond\mathcal{S}$ on the solvability of NB-AC.

A corollary of our result above is that we can exhibit a failure detector that is strictly weaker than $\mathcal{P}$, and yet that solves NB-AC. Is this only of theoretical interest? We believe not, as we will discuss below.

In practice, failure detectors are typically approximated using time-outs. To implement a failure detector that approximates $\mathcal{P}$, one needs to choose a large time-out value in order to avoid false failure suspicions. A NB-AC algorithm based on $\mathcal{P}$ (e.g., the 3PC algorithms [14]) might for instance violate agreement at the least false suspicion. The drawback with using a large time-out value is the slow fail-over time. In contrast, an algorithm like the one of Figure 2 (or like those of [15, 13]) would never violate agreement in case of false failure suspicions, precisely because the agreement part relies on failure detectors of the form $\diamond\mathcal{S}$ [9].

Interestingly, the inherent nature of $?\mathcal{P} + \diamond\mathcal{S}$ helps deconstruct NB-AC as suggested in Figure 2: a *preparation* phase where an outcome value for the transaction is chosen (using $?\mathcal{P}$), and an *agreement* phase where the processes must decide on a common outcome (using $\diamond\mathcal{S}$ - underlying Consensus). Hence, two time-out values could be used, one for each phase of the algorithm (i.e., for each failure detector implementation): the first time-out value to approximate $?\mathcal{P}$ and a second time-out value to approximate $\diamond\mathcal{S}$. The first time-out value should be large, in order to avoid false detection and, in the context of NB-AC, avoid to abort transactions that should otherwise be committed. The second time-out value could be relatively short in order to fastly react to failures during agreement: false suspicions might lead to delay decisions but have no impact on the actual outcome of the transaction. One might here even use a dynamic time-out value that keeps increasing until a decision is reached [2].

It is also important to notice that, at least in *normal runs* where no process crashes or is suspected to have crashed, solving NB-AC with $?\mathcal{P} + \diamond\mathcal{S}$, instead of $\mathcal{P}$, does not impact the performance of a NB-AC algorithm. These are the most frequent runs in practice and for which algorithms are usually optimised.

Typically, one can devise algorithms that have the same communication patterns as 3PC-like algorithms [14] to solve NB-AC with $?\mathcal{P} + \diamond\mathcal{S}$. For example, along the lines of the algorithm of Figure 2, one can build a centralised algorithm that, if used with the Consensus algorithm of [2], has the same communication pattern as the centralised 3PC algorithm [14] (in normal runs).

6 Concluding Remarks

According to Jim Gray [10], the *Atomic Commit* problem was first identified by Niko Garzado (IBM) in 1971. The problem has since then sparked off the interest of many researchers. In [10, 12], the problem was compared to the *Byzantine Generals* problem. Jim Gray pointed out in [10] the differences in the underlying assumptions, whereas Leslie Lamport stated in [12] that, despite these differences, some timing lower bounds on the solvability of the Byzantine Generals problem also apply to Atomic Commit. In 1981, a new variant of the problem was introduced by Dale Skeen: *Non-Blocking Atomic Commit (NB-AC)* [14]. Vassos Hadzilacos pointed out in [11] that any solution to NB-AC is also a solution to *Weak Consensus* [5].

In [7], it was shown that Consensus is sometimes solvable where NB-AC is not.[7] In [6], it was shown that the weakest failure detector for NB-AC is the *Perfect* failure detector $\mathcal{P}$, if we consider NB-AC to be defined among every pair of processes. If we consider however NB-AC to be defined among a set of at

[7] A weaker specification of the problem was also given there. The new specification was called *NB-WAC (Non-Blocking Weak Atomic Commit)*, and defined with the *Agreement*, *Termination*, *Abort-Validity* properties of NB-AC, plus the following *Commit-Validity* property: *Commit* is the only possible decision if no process is suspected or votes *no*. Although the specification makes sense in practice, it can be implemented with a failure detector that always suspects all processes.

least three processes and at least one can crash, [8] shows that $\mathcal{P}$ is not necessary for NB-AC. This paper goes a step further by showing that even the *Eventually Strong* failure detector ($\diamond\mathcal{S}$) is actually not necessary for NB-AC. An interesting consequence of this result is that NB-AC and Consensus are actually incomparable.

This paper also introduces the *Anonymously Perfect* failure detector $?\mathcal{P}$, and shows that: (1) $?\mathcal{P}$ is necessary to solve NB-AC, and (2) $?\mathcal{P} + \diamond\mathcal{S}$ is sufficient to solve NB-AC when a majority of the processes are correct. Interestingly, failure detector $?\mathcal{P} + \diamond\mathcal{S}$ helps deconstruct NB-AC: intuitively, $\diamond\mathcal{S}$ conveys the pure agreement part of NB-AC whereas $?\mathcal{P}$ conveys the specific nature of atomic commit. Besides better understanding the problem, this deconstruction provides some practical insights about how to adjust failure detector values in atomic commit protocols.

We leave the question of determining the weakest failure detector to solve NB-AC open for further investigation.

Acknowledgments

I thank Tushar Chandra for interesting discussions about the comparison between NB-AC and Consensus. I am also very grateful to the reviewers for many useful suggestions on the results and their presentation.

References

[1] O. Babaoglu and S. Toueg. *Non-Blocking Atomic Commitment.* In Distributed Systems, pages 147-166. Sape Mullender ed, ACM Press, 1993.

[2] T. Chandra and S. Toueg. *Unreliable Failure Detectors for Reliable Distributed Systems.* Journal of the ACM, 43(2), March 1996.

[3] T. Chandra, V. Hadzilacos and S. Toueg. *The Weakest Failure Detector for Solving Consensus.* Journal of the ACM, 43(4), July 1996.

[4] B. Coan and J. Welch. *Transaction commit in a realistic timing model.* Distributed Computing, 4(2), 1990.

[5] M. Fischer, N. Lynch, and M. Paterson. *Impossibility of Distributed Consensus with One Faulty Process.* Journal of the ACM, 32 (2), pages 374-382, April 1985.

[6] E. Fromentin, M. Raynal, and F. Tronel. *About Classes of Problems in Asynchronous Distributed Systems with Process Crashes.* Technical Report IRISA 1178. Also in proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS), 1999.

[7] R. Guerraoui. *Revisiting the relationship between non-blocking atomic commitment and consensus.* In Distributed Algorithms (WDAG'95), Springer Verlag (LNCS 972), September 1995.

[8] R. Guerraoui. *On the Hardness of Failure-Sensitive Agreement Problems.* Information Processing Letters (IPL), to appear. Also, Technical Report SSC-2000-008, Swiss Federal Institute of Technology (EPFL), Lausanne.

[9] R. Guerraoui. *Indulgent Algorithms.* In proceedings of the ACM Symposium on Principles of Distributed Computing, ACM Press, July 2000.

[10] J. Gray. *A Comparison of the Byzantine Agreement Problem and the Transaction Commit Problem.* In Fault-Tolerant Distributed Computing, B. Simons and A. Spector (ed), Springer Verlag (LNCS 448), 1987.

[11] V. Hadzilacos. *On the relationship between the atomic commitment and consensus problems.* In Fault-Tolerant Distributed Computing, B. Simons and A. Spector (ed), Springer Verlag (LNCS 448), 1987.

[12] L. Lamport. *The Weak Byzantine Generals Problem.* Journal of the ACM, 30 (3), 1983.

[13] I. Keidar and D. Dolev. *Increasing the Resilience of Atomic Commit, at No Additional Cost*. In proceedings of the ACM Symposium on Principles of Database Systems, ACM Press, May 1994.

[14] D. Skeen. *NonBlocking Commit Protocols*. In proceedings of the ACM SIGMOD International Conference on Management of Data, ACM Press, 1981.

[15] D. Skeen. *A Quorum-Based Commit Protocol*. In proceedings of the Berkeley Workshop on Distributed Data Management and Computer Networks, (6), 1982.