The Gap in Circumventing the Consensus Impossibility

Rachid Guerraoui

Distributed Programming Laboratory, EPFL, CH-1015, Lausanne, Switzerland
tel: +41 21 693 5272, fax: +41 21 693 7570

Petr Kouznetsov *

Distributed Programming Laboratory, EPFL, CH-1015, Lausanne, Switzerland
tel: +41 21 693 5274, fax: +41 21 693 7570

Abstract

The seminal impossibility of reaching consensus in an asynchronous and crash prone system was established for a weak variant of the problem, usually called weak consensus, where a set of processes need to decide on a common value out of two possible values 0 or 1. On the other hand, abstractions that were shown to be, in some precise sense, minimal to circumvent the impossibility were determined for a stronger variant of the problem, called consensus, where the processes need to decide on one of the values they initially propose (0 or 1). These abstractions include synchronization primitives, namely shared object types, as well as failure detector oracles.

This paper addresses the question of whether these abstractions were actually also minimal to circumvent the impossibility of weak consensus. We first show that any deterministic object type that implements weak consensus also implements consensus. Then we exhibit a non-deterministic type that implements weak consensus, among any number of processes, but not consensus, even among two processes. In modern terminology, this type has consensus power 1 and weak consensus power ∞. Finally, we exhibit a failure detector that implements weak consensus but not consensus.

Key words: Asynchronous distributed system, consensus, weak consensus, FLP impossibility, atomic objects, determinism, failure detectors.

* Corresponding author.
Email addresses: rachid.guerraoui@epfl.ch (Rachid Guerraoui), petr.kouznetsov@epfl.ch (Petr Kouznetsov).

1 Introduction

Background.

A consensus protocol is a distributed algorithm that makes a set of processes decide on a common value out of two possible values: 0 or 1. In 1983, it was shown that no protocol can solve consensus in a basic distributed system model where no synchrony assumption is made (asynchronous system), processes can only communicate by exchanging messages, and at least one process can fail by crashing [1]. The impossibility was extended later to the shared memory model where processes could communicate through read-write atomic objects, i.e., registers [2].

Given the importance of consensus in reliable distributed computing, a lot of work has been devoted to studying abstractions that, when added to the basic distributed model, circumvent the impossibility. In particular, certain abstractions were shown to be, in certain senses, *minimal* to solve consensus. The exploration of such abstractions has been conducted following at least two major research trends.

(1) One trend consisted in augmenting the system model with more sophisticated synchronization abstractions than message passing channels or registers. More precisely, the idea was to study object types that should be used, besides registers, to solve consensus in an asynchronous system assuming an arbitrary number of possible crashes [3]. Types like *queue*, *test-and-set* or *compare-and-swap* can indeed be used to do so and they are said to *implement* consensus. It was observed that certain types could implement consensus among a certain number k of processes but not among $k+1$. For example, instances of type *queue* and registers make it possible to solve consensus among 2 processes but not among 3 processes [3]. In a sense, *queue* is a minimal type to implement consensus among 2 processes: 2 is also said to be the consensus power of *queue*. In comparison, the consensus impossibility of [2] implies that the consensus power of *register* is 1: with registers only, consensus cannot be solved among 2 processes. At the other extreme, the consensus power or type

compare-and-swap is ∞ [3]: instances of this type and registers make it possible to solve consensus among any number of processes. The notion of consensus power leads to define a hierarchy, called the *consensus hierarchy*, with types that have low consensus power at the bottom and those that have high consensus power at the top.

(2) The second trend consisted in augmenting the system model with synchrony assumptions, i.e., timing assumptions on process relative speeds and communication delays [4], and still rely on basic communication abstractions like message passing channels or registers. One can precisely reason about the ability to solve consensus using these synchrony assumptions through the concept of *failure detector* [5]. Informally, a failure detector is a distributed abstraction that provides processes with information about the failure pattern of every execution, i.e., about which process has crashed (resp. has not) at any given point in time. A failure detector is typically implemented using timeouts and, in this sense, it encapsulates synchrony assumptions. In [6], the *weakest* failure detector to implement consensus was determined. This failure detector, denoted by Ω, outputs at any time and every process of the system, exactly one one process so that, eventually, all correct processes output the same correct process. The very fact that Ω is the weakest to solve consensus means that (1) Ω implements consensus, and (2) any failure detector that implements consensus can emulate the output of Ω (i.e., implements Ω). The result of [6], established in the message passing model with a majority of correct processes, was extended to the shared memory model in [7].

Motivation.

The motivation of this work is the simple observation that the original consensus impossibility [1, 2] was stated for a *weak* variant of consensus, whereas abstractions to circumvent the impossibility, be them object types or failure detectors, have been studied with a stronger consensus variant in mind.

In a weak consensus protocol, the processes can decide any value (0 or 1), provided that there is an execution of the protocol where 0 is decided and one where 1 is decided. In the stronger variant of consensus, which is simply

called consensus in the literature, the value decided must be one of the values proposed. In particular, if all processes initially propose 1 (resp. 0), the decision value must be 0 (resp. 1).

It is indeed natural to state an impossibility result on the weak variant of consensus and, when seeking for abstractions that circumvent the impossibility, consider abstractions that also lead to solve a stronger variant of consensus. However, determining that some abstraction is, in some sense, minimal to implement (the strong variant of) consensus does not mean that the abstraction is indeed minimal to circumvent the impossibility (of weak consensus). The motivation of this work was to determine whether, computationaly speaking, there is a gap between the two problems. More precisely, we seek to address the two following questions about this gap:

(1) *(The type perspective on the gap).* If a type implements weak consensus, does it also implement consensus? In particular, is the consensus power of a type the same as its weak consensus power.
(2) *(The failure detector perspective on the gap).* If a failure detector D implements weak consensus, does it also implement consensus? In particular, is Ω also the weakest failure detector to implement weak consensus. [1]

Contributions.

To address the first question, we distinguish between deterministic types and non-deterministic ones. In short, a deterministic type is one such that the output and resulting state of any invocation of an object of that type, performed in the absence of concurrency and failures, is uniquely determined by (a) the state of the object prior to the invocation and (b) the invocation itself.

(1) We show that, for any number of processes, any deterministic type that implements weak consensus also implements consensus. In a sense, the consensus and weak consensus powers of a deterministic type are the same. Said differently, the consensus and weak consensus hierarchies, when restricted to deterministic types, are the same. To prove this result, we exploit the inherent computation power of deterministic types.

[1] This question was partially addressed in [6]; we will come back to this in Section 5.

In short, we observe that any protocol that solves weak consensus using objects of a deterministic type boils down to reaching a *critical state* s of some object X, such that, applying different operations to s leads to distinguishable states of X. Since X is deterministic, there is a protocol which brings X to state s. We use this observation to derive a protocol that solves another variant of consensus, named *team consensus* [8], and then derive the fact that the protocol also solves consensus [8, 9].

(2) We show that this is not the case with non-deterministic types. Basically, we exhibit a new non-deterministic type, which we call rambler, that implements weak consensus for an arbitrary number of processes, but cannot implement consensus even among two processes. In other words, we exhibit a non-deterministic type which has, as a weak consensus power ∞, and as consensus power 1. Type rambler is constructed in such a way that, using any number of its instances, no process can fetch any meaningful information about other processes: the instances might exhibit the very same behavior for an arbitrary sequence of invocations. Intuitively, this means that type rambler cannot implement consensus even among two processes. On the other hand, the type has some non-trivial agreement properties, and these make it possible to solve weak consensus among any number of processes using just one instance of type rambler.

To address the gap question in terms of failure detectors, we first observe that the implementability of weak consensus cannot actually be studied in the original formalism of [5, 6]. More precisely, we point out the fact that, in the original failure detector formalism of [5, 6], weak consensus would have an asynchronous solution, contradicting [1]. We first propose an extension of the basic failure detector formalism that addresses this issue by putting both consensus and weak consensus on the same stable ground. Then we exhibit a failure detector that implements weak consensus but not consensus, i.e., this failure detector is strictly weaker than Ω.

Roadmap. In Section 2, we present the system model. In Section 3, we recall the consensus and the weak consensus problems, as well as another variant of consensus, *team consensus*, which is a key element of one of our proofs. In Section 4, we show that any deterministic type that implements weak consen-

sus also implements consensus. In Section 5, we show that this is not the case with non-deterministic types. In Section 6, we discuss the implementability of weak consensus in a model with failure detectors. Then we exhibit a failure detector that implements weak consensus but not consensus. In Section 7 with conclude the paper with some general observations on the questions raised in this paper.

2 Model

The model we mainly consider in this paper is the one of [10, 11]: a set of asynchronous processes communicating through shared objects. We recall below the details of the model which are substantial for our results. In Section 6 we augment this model with the failure detector abstraction.

Processes.

We consider a set Π of $n+1$ *processes* $p_0, \ldots, p_n$ ($n \geq 1$) that communicate using shared objects. The processes might fail by crashing, i.e., stop executing their steps. A process that never crashes is said to be *correct*. A process that is not correct is said to be *faulty*. Processes are asynchronous in the sense that we do not make any assumption on their relative speeds.

Objects and types.

An *object* is a data structure that can be accessed concurrently by the processes. Every object is an instance of a *type* which is defined by a tuple (Q, O, R, δ). Here Q is a set of *states*, O is a set of *operations*, R is a set of *responses*, and $\delta \subseteq Q \times O \times Q \times R$ is a relation, known as the *sequential specification* of the type. We assume here that every sequential specification δ is *total*: for each pair $(q, o) \in Q \times O$, there exists a pair $(q', r) \in Q \times R$ such that $(q, o, q', r) \in \delta$. We distinguish here *deterministic* types, of which the sequential specification is a function $\delta : Q \times O \rightarrow Q \times R$, and *non-deterministic* types, of which the sequential specification carries each state and operation to a *set* of response and state pairs.

We consider here *linearizable* [12] objects: operations on the objects must appear in one-at-a-time order consistent with their real time order. We assume that the objects are *wait-free* [3]: any process completes any operation in a finite number of steps, regardless of delays or failures of other processes. We call wait-free linearizable object *atomic objects*. If an atomic object instantiates a deterministic type, we say that the object is *deterministic*.

Protocols.

A *protocol* is a distributed deterministic automaton that defines, for every state of each process p_i, the next *step* p_i is going to take, i.e., the next operation p_i is going to execute. We assume that each operation is executed instantaneously, so a *protocol execution* can be seen as a sequence of invocation-response pairs. We say that an execution e of P is an *i-solo execution* if p_i is the only process that takes steps in e.

Schedules.

A *schedule* is a (finite or infinite) sequence of identifiers of processes in Π. For a given protocol P, we say that a schedule σ *triggers an execution* e of P, if, in e, processes take steps of P in the order defined by σ. Clearly, if processes access only deterministic objects, a given schedule triggers exactly one execution. On the other hand, if non-deterministic objects can be accessed, a schedule might trigger a number of executions.

3 Variants of consensus

Weak consensus.

The seminal *consensus* problem consists for a set of processes in reaching a common decision based on the initial states of the processes. Traditionally [1], the binary version of the problem was defined through a set of properties which includes:

- Termination: every process that takes an infinite number of computation

steps eventually decides on a value in $\{0, 1\}$;

- Agreement: no two processes decide on different values.

Clearly, the problem defined *only* through these two properties has a trivial solution: e.g., every process always decides 0. To filter out such protocols, the following *non-triviality* property was defined [1]:

- Weak validity: every protocol has an execution in which 0 is decided and an execution in which 1 is decided.

The problem defined through termination, agreement and weak validity, called here *weak consensus*, is shown to be impossible to solve in an asynchronous system in the presence of at least one faulty process [1, 2].

Consensus.

This impossibility result of [1, 2] clearly holds also for the *consensus* problem in which every process initially has a *proposal* value in $\{0, 1\}$ and, instead of weak validity, the following property is ensured:

- Validity: any decided value is the initial value of some process.

Weak consensus is trivially reduced to consensus: any solution of consensus has an execution in which 0 is decided (e.g., when all processes propose 0) and an execution in which 1 is decided (e.g., when all processes propose 1).

Consensus solvability and initial states.

We say that a protocol P *solves* (weak) consensus if the set of all possible executions of P satisfies *termination*, *agreement* and *(weak) validity*. A set $\mathcal{S}$ of types is said to implement (weak) consensus if there exists a protocol P that solves (weak) consensus and, in every execution of P, processes access only objects of types in $\mathcal{S}$.

We assume here that every atomic object has a predefined *initial state*. A state s of an object is called *reachable* if there is a sequence of operations that brings the object from the initial state to s. We also use the following result on deterministic atomic objects [9]:

Lemma 1 *Let $\mathcal{S}$ be any set of deterministic types. If $\mathcal{S}$ implements consensus when objects of types in $\mathcal{S}$ are initialized to some reachable state s, then $\mathcal{S}$ also implements consensus when objects of types in $\mathcal{S}$ are initialized to the initial state.*

Team consensus.

To prove our first result (next section), we use a restricted form of consensus, *team consensus* [8]. This variant of consensus ensures *agreement* only if the input values satisfy certain conditions. More precisely, assume that there exists a (known a priori) partition of Π into two non-empty sets (teams). Team consensus guarantees agreement if all processes of the same team have the same input value. Obviously, team consensus can be solved whenever consensus can be solved. Surprisingly, the converse is also true [8]:

Lemma 2 *Let $\mathcal{S}$ be any set of types. If $\mathcal{S}$ implements team consensus, then $\mathcal{S}$ also implements consensus.*

4 Deterministic types

In this section, we show that, with respect to deterministic types, weak consensus is equivalent to consensus: any set of types that implements weak consensus also implements consensus.

Theorem 3 *Let $\mathcal{S}$ be any set of deterministic types that includes* register*. If $\mathcal{S}$ implements weak consensus, then $\mathcal{S}$ also implements consensus.*

Proof: Let P be any protocol that solves weak consensus using objects of types in $\mathcal{S}$.

Let G be the execution graph of P: the vertexes of G are all possible states of P (defined by the states of the processes and all shared objects), and vertexes s and s' are linked with an edge directed from s to s' if and only if there is a step of P that, applied to s, results in s'.

Following [1], we use the notion of *valence* of a vertex of G. A state s has a valence $v \in \{0, 1\}$ if there is a state s' reachable from s (i.e., there exists a path in G from s to s') in which some process decides v. If a state has both valences 0 and 1, it is called *bivalent*. If a state has only one valence v, it is called *v-valent*. A state is univalent if it is 0-valent or 1-valent. Termination of weak consensus ensures that any state of P is either bivalent or v-valent for some $v \in \{0, 1\}$.

We proceed through the following arguments:

(1) There exists a *critical* state in G, i.e., a bivalent state s such that every step of P applied to s results in a univalent state [1].

(2) Assume that the system is in a critical state s. Consider a step of P applied to s as follows. Since protocol P and all objects that we use are deterministic, the step of a given process triggers exactly one transition in graph G. Thus, the valence of the resulting state is defined by the identity of the process that takes the step. Now we partition the system into two teams Π_0 and Π_1: Π_v ($v \in \{0, 1\}$) consists of the processes whose steps applied to s result in a v-valent state. Since s is bivalent, the two teams are non-empty.

Now we can solve *team consensus* for teams Π_0 and Π_1 using $\mathcal{S}$ as follows. We associate each team Π_v ($v \in \{0, 1\}$) with register r_v. Every process writes its input value into its team's register and then runs P starting from its state in s until it decides. Agreement and validity are ensured as long as members of the same team propose the same value.

Since team consensus is equivalent to consensus (Lemma 2), we obtain a solution to consensus from $\mathcal{S}$ as long as the objects are initialized to their states in s. But any solution to consensus using deterministic objects initialized to a given reachable state can be transformed into a solution of consensus for the initial state (Lemma 1). Thus, $\mathcal{S}$ implements consensus. □

5 Non-deterministic types

It turns out that some *non-deterministic* atomic objects capable to implement weak consensus are too weak to implement consensus. To illustrate this, we introduce a new non-deterministic type which we call rambler. Through accessing objects of type rambler, no process can fetch any meaningful information about other processes: the objects might exhibit the very same behavior for an arbitrary sequence of accesses. Intuitively, this means that, combined with registers, the objects of type rambler cannot solve consensus even among two processes. On the other hand, the type is strong enough to solve weak consensus.

More precisely, type rambler is defined by the tuple (Q, O, R, δ), where:

- $Q = \{\perp, t_0, t_1, 0, 1\}$ is the set of its states;
- $O = \{o_0, o_1\}$ is the set of its operations;
- $R = \{0, 1\}$ is the set of its responses;
- and δ, its sequential specification, is

$$\delta = \{(\perp, o_i, t_j, j), (\perp, o_i, t_j, 1-j),$$
$$(t_0, o_i, i, i), (t_1, o_i, 1-i, 1-i), (j, o_i, j, j)\}_{i,j \in \{0,1\}}$$

The state transition graph of a rambler object is depicted in Figure 1. The nodes of the graph define the states of the object and the edges define operations applied in the states and the corresponding responses.

Note that type rambler is built in such a way that there is no way for a process to get a clue about other processes through accessing an object of this type initialized to $\perp$. More precisely, objects of type rambler satisfy the following property:

Lemma 4 *Let P be any protocol that uses atomic objects of types in {rambler, register} and σ be any schedule. Then there is an execution e of P triggered by σ in which all objects of type rambler return 0 for all invocations.*

Proof: We construct execution e triggered by σ as follows. Processes take

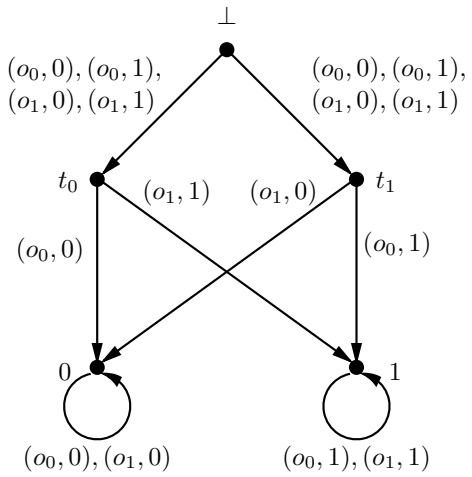

Fig. 1. State transition graph of rambler.

steps according to σ until an object of type rambler is accessed for the first time. We assume that the object returns 0 (this is possible for any invocation) and we do not specify its state until the object is accessed for the second time (it can be any state in $\{t_0, t_1\}$). Assume that the operation with which the object is accessed for the second time is o_i ($i \in \{0, 1\}$). Then we postulate that, after the first invocation, the object has state t_i. By the specification of type rambler, the object returns 0 on the second and all subsequent invocations.

By repeating the argument for every object of type rambler, we obtain an execution in which all such objects return nothing but 0. □

Weak consensus with rambler.

Despite the weak "synchronization power" of objects of type rambler emphasized in Lemma 4, a single object of the type, initialized to $\perp$, can implement weak consensus: p_i just invokes $o_{i \bmod 2}$ twice on the object and decides on the last returned value. After the first operation, the object is brought to a state in $\{t_0, t_1\}$ and becomes deterministic. Assume that the object is in state t_0. Now if p_0 is the first to access it with operation o_0, then the decision value is 0. If p_1 is the first to access it, then the decision value is 1. The case when the state of the object is t_1 is symmetric. Thus, there exists a 0-valent and a 1-valent executions, so weak validity is ensured. The protocol returns at most one value in $\{0, 1\}$ in any execution, so agreement is also ensured.

Impossibility of consensus with rambler.

We proceed by contradiction. Let P be a protocol that solves consensus using atomic objects of types in {rambler,register}. Consider an execution of P in which a process p_i decides. We call the state of p_i just before the decision a *resulting* state of p_i. Similar to [9,13,14], we define the *view graph* of P, denoted by $\mathcal{C}(P)$, as follows. Vertexes of $\mathcal{C}(P)$ represent the resulting states of processes $p_0, \ldots, p_n$. Two vertexes are connected by an edge if the corresponding states can result from the same execution of P.

Before proving that consensus is impossible with registers and objects of type rambler, we observe that the following property holds:

Lemma 5 *Let i and j be any two integers in $0, \ldots, n$, $i \neq j$, v_i be a vertex of and v_j be any two vertexes of $\mathcal{C}(P)$ corresponding, respectively to i-solo and j-solo executions of P. Then v_i and v_j belong to separate connected components of $\mathcal{C}(P)$.*

Proof: Assume, by contradiction, that there are two processes p_i and p_j whose solo executions give two vertexes v_i and v_j of the same connected components of $\mathcal{C}(P)$: there is a path from v_i to v_j in $\mathcal{C}(P)$. Assume p_i and p_j proposed, respectively, 0 and 1. By validity of consensus, p_i decides 0 in any i-solo execution, and p_j decides 1 in any j-solo execution. Thus, there is an execution on the path from v_i to v_j in which different values are decided by different processes — a contradiction. □

Theorem 6 *No protocol can solve consensus with objects of types in {rambler,register}.*

Proof: We establish a contradiction through the following steps.

(1) By Lemma 4, for any schedule σ, there is an execution e of P triggered by σ in which objects of type rambler return 0 for any invocation. Let $\mathcal{C}'$ be the subgraph of $\mathcal{C}$ corresponding to such executions e.

(2) For all $i \in \{0, 1\}$, let v_i be the vertex of $\mathcal{C}'$ corresponding to an i-solo execution. We show in the following that v_0 and v_1 belong to the same connected component of $\mathcal{C}'$.

Assume, by contradiction, that v_0 belongs to a component $\mathcal{C}_0$ of $\mathcal{C}'$,

and v_1 belongs to a component $\mathcal{C}_1$ of $\mathcal{C}'$, disconnected from $\mathcal{C}_0$. Then two processes can solve consensus by using only registers as follows. Every process p_i writes its value in register r_i and runs protocol P but, instead of accessing an object of type rambler, p_i assumes that 1 was returned for the invocation. Finally, the process ends up with a view in $\mathcal{C}'$. If the resulting view belongs to $\mathcal{C}_0$, the value read in r_0 is decided. Otherwise, the value read in r_1 is decided. Termination and agreement are trivially ensured. Validity follows from the fact that no process can reach a state in $\mathcal{C}_i$ unless p_i has taken at least one step of P. By the protocol, before taking a step of P, p_i writes its input value in r_i. Thus, no value is decided unless it is an input value of some process.

So two processes solve consensus using only registers, contradicting [1, 2]. Therefore, v_0 and v_1 belong to the same connected component of $\mathcal{C}'$.

(3) It follows from Lemma 5 that v_0 and v_1 do not belong to the same connected component of $\mathcal{C}'$ — a contradiction with (2).

Thus, we conclude that no protocol can solve consensus using objects of types in {rambler, register}. □

6 Failure detectors

The previous section highlights the existence of a gap between weak consensus and consensus in a distributed system model augmented with (non-deterministic) object types abstractions. It is natural to ask whether such a gap also exists in a model where the consensus impossibility is circumvented using failure detector abstractions [5].

In the following, we first discuss this question in the original model of [5,6]: a set of asynchronous processes, some of which can fail by crashing, communicate by message passing and can consult failure detector abstractions to get information about the failure pattern of their current execution. The discussion also applies to a model where, instead of message passing, the processes would communicate using registers [7].

We motivate below the need for slightly revisiting the model. Then we exhibit a failure detector that implements weak consensus but not consensus, i.e., we also highlight a failure detector gap.

An issue with the model.

In fact, the weak consensus problem was also discussed in [6], where a failure detector that implements weak consensus but not consensus was exhibited. For any failure pattern, this failure detector either permanently outputs 0 at all processes, or permanently outputs 1 at all processes. It is claimed in [6] that this failure detector, which we denote here by $\mathcal{X}$, trivially implements weak consensus: every process just decides on the value obtained from the failure detector. However, a deeper look a $\mathcal{X}$ reveals a fundamental contradiction with the consensus impossibility [1]: failure detector $\mathcal{X}$ can be *emulated* in an asynchronous system. Indeed, a "fake" failure detector that outputs 0 for any failure pattern, emulates $\mathcal{X}$. As a consequence, the fact that $\mathcal{X}$ solves weak consensus would imply that weak consensus is solvable in an asynchronous system, which, in turn, would contradict [1, 2]. Strange, isn't it?

The issue here has to do with the very definition of a *problem*. In [6], a problem is defined through a set of properties which every execution of a protocol that solves this problem should satisfy. Strictly speaking, weak consensus is not a problem in this sense: it is defined through a set of properties which *the whole set* of executions of a protocol that solves the problem should satisfy. So in order to decide whether a given protocol A solves weak consensus, it is not sufficient to ensure that any execution of A satisfies some properties. What we need to show is that the set of *all* executions of A satisfies some properties.

To find out whether the weakest failure detector to implement consensus is also the weakest to implement weak consensus, we first need to revisit the very notion of a problem. We do so by (1) defining a problem as a set of properties P over a set of executions, and then (2) stating that *a protocol A solves the problem* if the set of *all* executions of A satisfies P. This definition is strictly more general than the one of [6].

The failure detector gap.

Now we are ready to define a failure detector, denoted by $\mathcal{Y}(F_0, F_1)$ where F_0 and F_1 are two failure patterns, that implements weak consensus in the mentioned sense but does not implement consensus. Like failure detector $\mathcal{X}$ recalled above, for every failure pattern, $\mathcal{Y}(F_0, F_1)$ either permamently outputs 0 at all processes, or permamently outputs 1 at all processes. Further, for failure pattern F_0, $\mathcal{Y}(F_0, F_1)$ outputs only 0, and, for failure pattern F_1, $\mathcal{Y}(F_0, F_1)$ outputs only 1. A trivial protocol solves weak consensus using $\mathcal{Y}(F_0, F_1)$, for any two different F_0 and F_1.

Now we show that some $\mathcal{Y}(F_0, F_1)$ does not implement consensus. Assume, by contradiction, that there is a protocol A that solves consensus using $\mathcal{Y}(F_0, F_1)$ where, in F_0, some process crashes initially (at time $t = 0$). We consider a set of failure patterns in which no process is crashed initially (at time $t = 0$). By our assumption, $\mathcal{Y}(F_0, F_1)$ outputs 1 for all these failure patterns. Thus, we have a solution of consensus in the presence of at least one failure (processes can fail at any time $t > 0$) without any failure detector — a contradiction with [1].

7 Concluding notes

The motivation of this work was the observation that the impossibility of consensus was established for a weak variant of the problem, namely weak consensus, whereas research on circumventing the impossibility has been performed on the stronger consensus variant. A posteriori, this is not surprising because consensus is *universal* whereas weak consensus is not. Indeed, using consensus and registers, any type can be implemented. The existence of a non-deterministic type that implements weak consensus but not consensus precisely implies that this is not the case with weak consensus.

This paper shows that the gap between weak consensus and consensus also holds in a system with failure detectors. We have in particular shown that Ω is not the weakest failure detector to implement weak consensus. Determining this weakest failure detector remains an open question.

Acknowledgments

We are grateful to Partha Dutta for an interesting discussion on the subject, and to Eli Gafni for his observation that "getting 0 when all start with 1 may only be the result of non-determinism, intuitively speaking" [15].

References

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

[2] M. C. Loui, H. H. Abu-Amara, Memory requirements for agreement among unreliable asynchronous processes, Advances in Computing Research (1987) 163–183.

[3] M. Herlihy, Wait-free synchronization, ACM Transactions on Programming Languages and Systems 13 (1) (1991) 124–149.

[4] C. Dwork, N. A. Lynch, L. J. Stockmeyer, Consensus in the presence of partial synchrony, Journal of the ACM 35 (2) (1988) 288 – 323.

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

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

[7] W.-K. Lo, V. Hadzilacos, Using failure detectors to solve consensus in asynchronous shared-memory systems, in: Proceedings of the 8th International Workshop on Distributed Algorithms (WDAG'94), Vol. 857 of LNCS, Springer Verlag, 1994, pp. 280–295.

[8] E. Ruppert, Determining consensus numbers, SIAM Journal of Computing 30 (4) (2000) 1156–1168.

[9] E. Borowsky, E. Gafni, Y. Afek, Consensus power makes (some) sense!, in: Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing (PODC'94), 1994, pp. 363–372.

[10] P. Jayanti, Wait-free computing, in: Proceedings of the 9th International Workshop on Distributed Algorithms (WDAG'95), Vol. 972 of LNCS, Springer Verlag, 1995, pp. 19–50.

[11] P. Jayanti, Robust wait-free hierarchies, Journal of the ACM 44 (4) (1997) 592–614.

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

[13] O. Biran, S. Moran, S. Zaks, A combinatorial characterization of the distributed tasks which are solvable in the presence of one faulty processor, in: Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing (PODC'88), 1988, pp. 263–275.

[14] M. Herlihy, N. Shavit, The asynchronous computability theorem for t-resilient tasks, in: Proceedings of the 25th ACM Symposium on Theory of Computing (STOC), 1993, pp. 111–120.

[15] E. Gafni, Private communication (2003).