HAL
archives-ouvertes.fr

The Combined Power of Conditions and Information on Failures to Solve Asynchronous Set Agreement

Achour Mostefaoui, Sergio Rajsbaum, Michel Raynal, Corentin Travers

▶ To cite this version:

Achour Mostefaoui, Sergio Rajsbaum, Michel Raynal, Corentin Travers. The Combined Power of Conditions and Information on Failures to Solve Asynchronous Set Agreement. [Research Report] PI 1897, 2008, pp.30. inria-00294706

HAL Id: inria-00294706

https://hal.inria.fr/inria-00294706

Submitted on 10 Jul 2008

THE COMBINED POWER OF CONDITIONS AND INFORMATION ON FAILURES TO SOLVE ASYNCHRONOUS SET AGREEMENT

A. MOSTÉFAOUI   S. RAJSBAUM   M. RAYNAL   C. TRAVERS

The Combined Power of Conditions and Information on Failures to Solve Asynchronous Set Agreement

A. Mostéfaoui[*]   S. Rajsbaum[**]   M. Raynal[***]   C. Travers[****]

Systèmes communicants
Projet ASAP

Abstract: To cope with the impossibility of solving agreement problems in asynchronous systems made up of n processes and prone to t process crashes, system designers tailor their algorithms to run fast in "normal" circumstances. Two orthogonal notions of "normality" have been studied in the past through *failure detectors* that give processes information about process crashes, and through *conditions* that restrict the inputs to an agreement problem. This paper investigates how the two approaches can benefit from each other to solve the *k-set agreement* problem, where processes must agree on at most k of their input values (when $k = 1$ we have the famous consensus problem). It proposes novel failure detectors for solving k-set agreement, and a protocol that combines them with conditions, establishing a new bridge among asynchronous, synchronous and partially synchronous systems with respect to agreement problems. The paper proves also a lower bound when solving the k-set agreement problem with a condition.

Key-words: Asynchronous System, Condition, Consensus, Failure Detection, Input Vector, Legal condition, Set Agreement, Process Crash, Shared Memory, Snapshot.

(Résumé : tsvp)

[*] IRISA, Université de Rennes 1, Campus de Beaulieu, 35042, Rennes Cedex, France, `achour@irisa.fr`
[**] nstituto de Matemáticas, Universidad Nacional Autónoma de México, D. F. 04510, Mexico, `rajsbaum@math.unam.mx`
[***] IRISA, Université de Rennes 1, Campus de Beaulieu, 35042, Rennes Cedex, France, `raynal@irisa.fr`
[****] IRISA, Université de Rennes 1, Campus de Beaulieu, 35042, Rennes Cedex, France, `ctravers@irisa.fr`

La puissance additive des conditions et des détecteurs de fautes pour résoudre les problèmes d'accord

Résumé : Ce rapport étudie la puissance additive des conditions et des détecteurs de fautes pour résoudre les problèmes d'accord.

1   Introduction

Distributed services have to run efficiently and reliably in complex environments with unpredictable processing and communication delays, where components can fail in various ways. It is unavoidable to encounter scenarios where the system performance will degrade, or even manual intervention will be required. Therefore, system designers tailor their applications to run fast in "normal" circumstances while having expensive recovery procedures in the rare cases of "abnormal" circumstances. Two complementary notions of "normality" have been considered, mirroring the traditional computer science duality of *control* and *data*. On the control side we have the *failure detector* approach [8], that abstracts away useful failure pattern information, available in common operating scenarios. On the data side, we have the *condition-based* approach [36], that looks at common input data patterns of a certain distributed problem we are interested in. The aim of this paper is to study how the two approaches interact, and can benefit from each other, with respect to solving *agreement problems*.

1.1   Context of the Paper

Distributed services often rely on an underlying agreement protocol. The most popular and fundamental of the agreement problems is consensus, which is actually indispensable for a lot of services. This paper investigates the possibilities and limitations of solving consensus, and other weaker agreement problems, in a system with failure detectors and conditions.

Agreement problems   The *consensus* problem can informally be stated as follows. Each process proposes a value and the processes that are not faulty have to decide the same value, such that the value decided is one of the proposed values. As a familiar example, it is easy to see that the *atomic broadcast* problem relies on consensus as it requires that all the processes deliver the messages they broadcast in the same order: they have consequently to agree in one way or another on the same message delivery order. However, it is well-known that the consensus problem has no solution in message passing asynchronous systems made up of n processes that need to tolerate a single process crash failure [16].

The k-*set agreement* problem [9] relaxes the consensus requirement to allow up to k different values, out of the proposed values, to be decided; when $k = 1$ we have the consensus problem. Set agreement is an abstraction of problems that are weaker than consensus. Its discovery was motivated by the search for a problem that is solvable when $k - 1$ processes can crash, but not when k can crash, in an asynchronous system. Since then, it has been very valuable in the development of the foundations of distributed computing. The proofs in [6, 29, 49] showing that k-set agreement is not solvable in a system of $k + 1$ processes where k can crash uncovered a deep connection between distributed computing and topology, and motivated a significant amount of subsequent research.

The situation is totally different in synchronous systems, where both consensus and k-set agreement can be solved for any value of t (the maximum number of process crashes) [10, 48]. However, there are limitations on how fast these problems can be solved in a synchronous system, as a function of the number of failures t to be tolerated. It has been shown that consensus requires $t + 1$ rounds in the worst case [15, 33], and there are protocols that meet this lower bound. These results have been generalized for the set agreement problem in [10, 20, 47].

The following two complementary notions of "normality" have been considered[1] to cope with the consensus and set agreement asynchronous impossibility results and synchronous lower bounds.

Control: Enriching the underlying system   The first approach focuses on the behavior of the underlying system. In this case "normal circumstance" means a period during which the system behaves in a relatively synchronous way. Namely, periods during which upper bounds on process execution speeds and on message transmission delays hold (various such *partially synchronous* models have been considered, e.g., [14]), or periods during which message exchange patterns satisfy some properties (e.g., the notion of winning/losing responses introduced in [35]) that allow solving consensus. A *failure detector* abstracts away such low-level assumptions by providing processes with a primitive they can invoke, that returns information on process failures. One of the noteworthy features of failure detectors is the modular approach they favor: one can independently, on one side, solve a problem with the help of a particular class of failure detector, and, on another side, implement the assumed failure detector with the help of the underlying

---

[1]There are other approaches, like randomization or stronger shared memory primitives.

timing or order assumptions. The design, transportability and proof of protocols then become modular and easier to achieve.

Chandra and Toueg introduced the *failure detector* notion [8] and defined eight classes that can be used to solve asynchronous consensus. Together with Hadzilacos, they later showed that one of these classes is the weakest class of failure detectors to solve consensus when $t < n/2$ [7] (n being the total number of processes, and t an upper bound on the number of faulty processes). The weakest failure detector for consensus and any value of t was identified in [12]. These results are of great theoretical interest because they identify the minimum knowledge about failures that needs to be abstracted to solve consensus. Failure detectors to solve set agreement have also been proposed [27, 43, 51] but we do not know yet what is the minimum knowledge about failures that needs to be abstracted to solve this problem.

The failure detector approach has favored the design of indulgent protocols [21]. A protocol is *indulgent* with respect to its failure detector if it never violates its safety property. This means that when the underlying failure detector satisfies its specification (normal circumstances), the protocol terminates correctly; and, when the underlying failure detector does not satisfy its specification (abnormal circumstances), it is possible that the protocol does not terminate, but if it terminates, it does so correctly. Various indulgent failure detector-based consensus protocols have been proposed (e.g., [8, 23, 31, 42, 45, 50]).

Data: Restricting the inputs   The *condition-based* approach [36] consists in looking at certain combinations of input values of a given distributed problem. It is often the case in practice that some combinations of input values of processes occur more frequently than others. For example, in an election, it is often the case that the difference in the number of votes that candidates receive is significant. More precisely, an *input vector* contains the values proposed by the processes in an execution. A *condition* C is a set of input vectors, each representing a common combination of inputs to the problem. If a protocol solves k-set agreement for C, then whenever the input vector belongs to C, all the correct processes decide. The solution should be indulgent in the sense that if correct processes decide while the input vector does not belong to the condition, they do not decide more than k values.

It was discovered in [36] that there is a family of conditions, called *x-legal*, that tie together asynchronous and synchronous systems with respect to consensus solvability. Informally, in an x-legal condition any two input vectors $I1, I2$ that force different decisions have $d(I1, I2) > x$ (Hamming distance); assuming $n > x$. Thus, in a sense, x is the "power" of the condition; larger values of x make it easier to solve consensus. Assuming up to t process crashes, and $d \leq t$ (d can have a negative value), let $\mathcal{S}_t^{[d]}$ be the set of all x-legal conditions, $x = t - d$ (e.g., $\mathcal{S}_t^{[0]}$ consists of the t-legal conditions). Then

$$\mathcal{S}_t^{[-t]} \subset \cdots \subset \mathcal{S}_t^{[-1]} \subset \mathcal{S}_t^{[0]} \subset \mathcal{S}_t^{[1]} \subset \cdots \subset \mathcal{S}_t^{[t]},$$

where $\mathcal{S}_t^{[t]}$ includes the condition made up of all the possible input vectors. For a condition $C \in \mathcal{S}_t^{[d]}$, $-t \leq d \leq t$ and a system prone to t process crashes, we have:

- For values of $d \leq 0$, for inputs in C consensus is solvable by more and more efficient protocols in a shared memory asynchronous system as we go from $d = 0$ to $d = -t$ [40].

- For values of $d > 0$, consensus is not solvable in an asynchronous system, but, for inputs in C, it is solvable in a message passing synchronous system with more and more rounds, as we go from $d = 1$ (two rounds) to $d = t$ ($t + 1$ rounds), and this is tight [37] (namely, when $C \in \mathcal{S}_t^{[d]}$ and $C \notin \mathcal{S}_t^{[d-1]}$, $(d + 1)$ rounds are sufficient, and necessary in worst case scenarios).

- $d = 0$ is the borderline case. On one hand, asynchronous consensus can be solved (despite up to t faulty processes) for a condition C if, and only if, C is t-legal [36]. On the other hand, consensus can be solved optimally (2 rounds) in a message passing synchronous system [37] for any t-legal condition.

The condition-based approach has been considered also for set agreement (and for other problems), but a characterization of the conditions that allow solving set agreement was not known (see Section 1.3 for more about this and other related works).

1.2   Motivation and Results

As we have seen, failure detectors and conditions are two orthogonal approaches to cope with the impossibility of solving agreement problems in asynchronous systems prone to t process crashes. So, the natural question that comes to mind is:

> *What is the relation between the condition-based approach and the failure detection-based approach when solving asynchronous agreement problems?*

More specifically, we are interested in studying how the two approaches can cooperate to solve set agreement problems. We would like to understand which combinations of failure detectors and conditions can be used to solve k-set agreement for a given value of k.

> *For a given condition C, what is a failure detector that abstracts away the synchrony needed to solve k-set agreement?*

These and similar questions are the topic addressed in the paper.

The paper contains three main contributions. While trying to answer the previous questions, we discovered a new class of failure detectors. We present an asynchronous condition-based set agreement protocol based on this kind of failure detectors. We present a lower bound showing that our protocol is optimal. The next three sections describe these results in more detail.

1.2.1   A new class of failure detectors

The first contribution of the paper is the definition of a new class of failure detectors that we denote ϕ_t^y, $0 \leq y \leq t$. A failure detector of ϕ_t^y provides a primitive, denoted $\text{QUERY}_y(S)$, that can be invoked by a process with a set of process ids S to be informed whether they have crashed or not. Roughly speaking, $\text{QUERY}_y(S)$ returns *true* only when all the processes in S have crashed. If at least one process in S is alive the output should be *false*. If $|S|$ is outside the range $t - y < |S| \leq t$, the query returns no useful information.

Notice that the nature of our failure detectors is different from the standard failure detectors of [8], that return a set of processes suspected to have crashed, and accept no input parameter. The motivation is that often a process p_i is interested in the failures of only a specific part of the network, namely S, while the standard failure detectors must find out the failure status of all processes in the network, even if p_i cares only about the state of a single process p_j.

For each value of y between 0 and t, there is a class of failure detectors, ϕ_t^y. The class ϕ_t^y provides more information on failures than the class ϕ_t^{y-1}. So, the class ϕ_t^t is the most powerful, while ϕ_t^0 is the weakest (it actually provides no dependable information on failures). Indeed, as shown in the paper, ϕ_t^y can be used to solve k-set agreement for smaller values of k than ϕ_t^{y-1}.

The paper also compares the power of the ϕ_t^y failure detectors and of the classic failure detector classes introduced by Chandra and Toueg [8]. It is shown that it is possible to build any class ϕ_t^y, $0 \leq y \leq t$, from a perfect failure detector as defined in [8]. (A perfect failure detector eventually detects all crashed processes, and never suspects erroneously a non-crashed process.) In contrast, none of the other classes of classic failure detectors can be used to build a failure detector of a class ϕ_t^y, $1 \leq y \leq t$. When we consider the construction in the other direction, we show that no class ϕ_t^y, $0 \leq y < t$, can be used to build any of the classic failure detector classes. When $y = t$, ϕ_t^t can be used to build a failure detector of the class $\mathcal{P}$.

1.2.2   A condition-based set agreement algorithm using failure detectors

A second contribution of the paper is the design of a condition-based set agreement protocol with access to a failure detector of the class ϕ_t^y, $0 \leq y \leq t$. The considered model is the classical asynchronous read/write shared memory distributed system prone to at most t process crashes. The protocol can be instantiated with any condition $C \in \mathcal{S}_t^{[d]}$, $0 \leq d \leq t$. As we have seen, $x = t - d$ represents the power of the condition. That is, once n and t are fixed, the protocol is parameterized by the power of the failure detector (captured by y), and the power of the condition (captured by $x = t - d$).

We use the following terminology. We say that "a protocol solves k-set agreement problem" if the correct processes always decide; and we say that "*a protocol solves the condition-based k-set agreement*" problem" if the correct

processes decide at least "in normal circumstances", where "*normal circumstances*" means when (1) the input vector belongs to the condition C, or when (2) a process decides, or less than k processes crash, or when (3) at least $t - d$ processes crash initially.

The proposed protocol solves condition-based k-set agreement for $k = 1 + \max(0, d - y)$. Making more explicit the power y of the failure detector, and the power $x = t - d$ of the condition, we have $k = 1 + \max(0, t - (x + y))$. This shows how, by adding the power of the condition, and the power of the failure detector, we can counterbalance the power t of the "adversary", in order to reduce the value of k. When we consider the boundary values of y and d, the protocol solves the following problems:

- $d = t$ corresponds to the case where there is no additional power provided by the condition, as then the condition C may contain all possible input vectors. But, as any input vector belongs to this trivial condition, all correct processes always decide, and consequently the protocol solves the k-set agreement problem. More precisely:
    - If $y = t$ (strongest failure detector), the protocol solves the consensus problem, $k = 1$.
    - If $y = 0$ (no failure detector), the protocol solves the trivial $(t + 1)$-set agreement problem.
    - If $0 < y < t$, protocol solves k-set agreement with $k = t + 1 - y$. When we compare to the previous case, this shows the benefit provided by a failure detector of the class ϕ_t^y. The number of decided values linearly decreases according to the power of the failure detector, as measured by y.

- $d = 0$ means that the condition C is t-legal, which means that condition-based consensus can be solved despite asynchrony and up to t crashes, with no failure detector. So, at most one value is decided, and all the correct processes terminate in normal circumstances. So, the protocol then solves condition-based consensus. (Let us notice that this is independent of the value of y.)

- $y = 0$ (no failure detector), the protocol then relies only on the condition and solves the condition-based k-set agreement problem for $k = d + 1$. No more than $(d + 1)$ values are decided, and the termination of the correct processes is guaranteed at least in normal circumstances: the number of decided values decreases linearly according to the parameter d defining the condition.

    This case is particularly interesting as it exhibits a new link relating synchronous and asynchronous systems. More precisely, when the condition C belongs to $\mathcal{S}_t^{[d]}$, and the input vector belongs to C, (1) it is possible to solve consensus in at most $(d+1)$ rounds in a synchronous system [37]; and (2) it is possible to solve $(d+1)$-set agreement in an asynchronous system, both systems being prone to t crashes. The optimal time bound for synchronous condition-based consensus is equal to the number of decided values in asynchronous condition-based set agreement. This time (in synchronous systems) *vs* number of decided values (in asynchronous systems) relation sheds a new light on the global picture concerning the relations between synchronous and asynchronous systems.

When we look at the general case, where the condition-based k-set agreement problem is solved with $k = 1 + \max(0, d - y)$, we see that when $y \geq d$, condition-based consensus is solved. This means that, if d is fixed, we only need to take a failure detector of the class ϕ_t^d to solve condition-based consensus (failure detectors of any class ϕ_t^y with $y > d$ are stronger than necessary). A similar reasoning can be done when y is fixed, and we have the choice of the condition class.

The proposed protocol is indulgent [21, 23]: it never violates its safety requirement (no more than $k = 1 + \max(0, d - y)$ values are decided), and the correct processes always terminate when the input vector belongs to the condition ("normal circumstances"). Interestingly, a simple modification provides a protocol version in which all the correct processes always terminate. This is obtained at the price of an increase in the number of values that can be decided when the input vector does not belong to the $(t - d)$-legal condition C, namely up to $k' = t + 1 - y$ different values can then be decided. When the system is equipped with a failure detector of the class ϕ_t^t, this protocol variant solves the consensus problem whatever the condition it is instantiated with.

1.2.3 A lower bound

A third contribution of the paper is a lower bound result showing that no protocol with access to a failure detector of the class ϕ_t^y can solve k-set condition-based agreement for $k \leq \max(0, d - y)$, if the condition is in $\mathcal{S}_t^{[d]}$. The proof is by reduction to the standard t-resilient k-set agreement problem, that is known to be impossible if $t \geq k$ [6, 28, 29, 49]. This lower bound result has two nice corollaries. One states that (in the absence of a failure detector)

there is no condition-based k-set agreement protocol such that $k \leq d$ for any $(t - d)$-legal condition (a previously open problem). The second one states that among all the failure detector classes of the family $(\phi_t^y)_{0 \leq y \leq t}$, the class ϕ_t^y is the weakest that allows solving the k-set agreement problem for $k > t - y$.

1.3   Related Work

The condition-based approach for consensus and set agreement   The condition-based approach has been applied to problems other than consensus, like interactive consistency [17], and more related to our work, set agreement [3, 39]. The paper [3] characterizes the set of input vectors that allow to solve $(n - 1)$-set agreement, wait-free, namely, when $t = n - 1$. Their notion of solvability is different from ours, since they assume a protocol never receives input vectors outside of the condition. In [39] another family of conditions for set agreement is defined, but no general lower bounds were proved. Randomization as a means of circumventing the set agreement asynchronous impossibility result has been considered in [44].

Failure detectors   Most of the research about failure detectors has been directed at solving consensus, but there have also been proposals of failure detectors for solving other problems. Failure detectors for implementing various objects and for solving non-blocking atomic commit have been studied (e.g., [12, 22, 46]). The weakest class of failure detectors to solve consensus was identified in [7, 12]. For our work, weaker classes of failure detectors are especially relevant, since set agreement is an easier problem than consensus (and if conditions are considered, it becomes even easier). Weaker classes of failure detectors were considered in [18, 43, 46, 51].

Failure detectors for set agreement   Among the failure detectors which are not strong enough to solve consensus, the *limited scope accuracy* failure detectors [25, 43, 51] have been studied with respect to set agreement. To illustrate this notion, let us consider the class denoted $\mathcal{S}_x$. A failure detector of that class satisfies the following two properties. The completeness property states that the processes that crash are eventually suspected in a permanent way. The limited scope accuracy property states that there is a correct process that is not suspected by a set -cluster- of x processes (some of these x processes may be correct, while others may be faulty). An $\mathcal{S}_x$-based k-set agreement protocol is presented in [43]. This protocol assumes $t < k + x - 1$ (which means that $(t + 1) - (x - 1)$ is the smallest value of k that it can tolerate). Using topological methods, it has been shown in [27] that this is actually a lower bound for any $\mathcal{S}_x$-based k-set agreement protocol (from which it follows that the previous protocol is optimal with respect to the number of faulty processes that can be tolerated). When the limited scope accuracy property has to hold only after some unknown but finite time, we get the class denoted $\diamond\mathcal{S}_x$. It is shown in [27] that any $\diamond\mathcal{S}_x$-based k-set agreement protocol requires $t < \min(n/2, k + x - 1)$. A $\diamond\mathcal{S}_x$-based protocol meeting this lower bound is also presented in [27]. It is shown in [2] that $t < x$ is a necessary and sufficient requirement to transform any failure detector of the class $\diamond\mathcal{S}_x$ into a failure detector of the class $\diamond\mathcal{S}_y$ for $y > x$.

The Class of Anonymously Perfect Failure Detectors   A failure detector of our class ϕ_t^y returns a binary output and can be invoked with a parameter S, that contains a set of process ids. In contrast, the classic failure detectors of [8] return a set of ids, and are invoked with no parameter. A failure detector class whose output is binary has been introduced by Guerraoui to solve the non-blocking atomic commit problem [22], but differently from ours, a failure detector of this class does not accept a parameter to invoke it. This class, called *anonymously perfect* failure detectors and denoted $?\mathcal{P}$, is defined as follows. Each process has a flag (initially equal to *false*) that is eventually set to *true* if and only if a process has crashed (the identity of the crashed process is not necessarily known, hence the name "anonymous").

   The definition of $?\mathcal{P}$ has been extended in [18] to take into account the fact that ℓ processes have crashed (instead of a single one). This class, denoted $?\mathcal{P}^\ell$, provides each process with a flag that is eventually set to *true* if and only if at least ℓ processes have crashed (observe that $?\mathcal{P}$ is $?\mathcal{P}^1$).

   So, a failure detector of the class $?\mathcal{P}^\ell$ answers *true* only if there is a set S of ℓ processes that have crashed. The set S is not known to the processes. Differently, when we consider ϕ_t^y, the set S is user-defined and specific to each invocation.

A variant of Ω   A generalization of the class of leader failure detectors denoted Ω has been introduced in [46]. More explicitly, Ω_z is the class of all failure detectors that provide the processes with a primitive LEADER() satisfying the

following properties. First, LEADER() always returns a set of at most z process identities. Second, there is a time τ such that, after τ, all the invocations of LEADER() by the correct processes return the same set of processes and this set includes at least one correct process. It is easy to see that Ω_1 is Ω, and Ω_n provides no information on failures. That is, in general, Ω_z is weaker than the weakest failure detector for consensus. However, Neiger introduced them to study questions about augmenting the synchronization power of types in the wait-free hierarchy [26], and their relation to set agreement was not studied.

In a follow up paper [38], we study the relation of Ω_z to set agreement. Moreover, we study our new failure detectors, with respect to $\diamond \mathcal{S}_x$, Ω_z, and show which reductions among these classes are possible and which are not.

1.4   Organization of the Paper

The paper is made up of 9 sections. After this introduction, and a short section introducing the computation model considered in the paper, Section 3 presents the new failure detector classes ϕ_t^y. (Section 8 compares them to classic failure detectors by Chandra and Toueg.) Section 4 provides a quick overview of the most relevant notions (for this paper) of the condition-based approach, including a definition of the condition-based k-set agreement problem. Section 5 presents a generic k-set agreement protocol that is based on the combined power of a failure detector of the class ϕ_t^y and a condition of the class $\mathcal{S}_t^{[d]}$. The protocol is discussed in Section 6 where, at the price of an increase of the number of decided values, an always terminating version is presented. Section 7 focuses on the lower bound result. Finally, Section 9 summarizes the content of the paper.

2   About the Model of Computation

This paper considers the usual *asynchronous model* with n processes $p_1, \ldots, p_n$, where at most t can crash, $1 \le t < n$. The processes communicate through a shared memory made up of single-writer, multi-readers atomic registers.

We assume processes have access to an oracle that provides possibly unreliable information on process failures. A *failure detector* provides processes with a primitive they can invoke to get information from the oracle on process failures.

3   The Failure Detector Classes $\{\phi_t^y\}_{0 \le y \le t}$

3.1   Definition

This section introduces a new class of failure detectors, parameterized by an integer y, $0 \le y \le t$, denoted ϕ_t^y. (A comparison to classic failure detectors is done in Section 8.) The power of a such a failure detector depends on the value of y. As we are about to see, a failure detector is more powerful for larger values of y, because it can return information about more specific regions of the network, namely, about smaller sets S of processes with $|S| > t - y$.

More precisely, a failure detector of the class ϕ_t^y provides a primitive QUERY$_y(S)$ that returns a boolean answer. A process invokes it with the parameter S, a set of processes specific to each invocation. Intuitively, if p_i invokes QUERY$_y(S)$ the answer will be *true* only when all processes in S have crashed. In that sense, these failure detectors are different from the standard failure detectors introduced by Chandra and Toueg [8], that return a set of processes suspected to have crashed, and do not accept an input parameter[2]. The motivation is that often a process p_i is interested in the failures of only a specific sector of the network, namely S, while the Chandra-Toueg failure detectors must find out the failure state of all processes in the network, even if p_i cares only about the state of only one process p_j.

A query QUERY$_y(S)$ such that $t - y < |S| \le t$ is *relevant*, otherwise it is *irrelevant*. Intuitively, "relevant" means that it provides dependable information on failures. The class ϕ_t^y is defined by the following properties:

- *Triviality property.* If $|S| \le t - y$, then QUERY$_y(S)$ returns *true*. If $|S| > t$, then QUERY$_y(S)$ returns *false*.
- *Safety property.* If QUERY$_y(S)$ is relevant, then if at least one process in S has not crashed when QUERY$_y(S)$ is invoked, the invocation returns *false*.

---

[2]We have shown in [38] that there are transformations between the ϕ failure detectors and a version with no input parameter. It is consequently possible to define them according to the failure pattern only.

- *Liveness property.* Let $\text{QUERY}_y(S)$ be a relevant query. Let τ be a time such that, at time τ, all the processes in S have crashed. There is a time $\tau' \geq \tau$ such that all the invocations of $\text{QUERY}_y(S)$ after τ' return *true*.

The triviality property says that the invoking process gets back a *true* output when the set S is too small, because in this case the failure detector is not powerful enough to answer reliably on a region of the network that is too focused. If the set S is too big, the output is *false*, because by definition, no more than t processes can fail. The safety property states that if the output of a relevant query is *true*, then all the processes in S have crashed. The liveness property states that $\text{QUERY}_y(S)$ eventually outputs *true* when all the processes in S have crashed (and the query is relevant).

3.2   Ranking the Classes $\{\phi_t^y\}_{0 \leq y \leq t}$

A failure detector of the class ϕ_t^0 provides no information related to failures as the invocation $\text{QUERY}_y(S)$ answers always *true* if $|S| \leq t$, and *false* if $|S| > t$. At the other extreme, with a failure detector of the class ϕ_t^t a process can query about the failure status of a single specific process, since $\text{QUERY}_y(S)$ may return significant failure information about sets S of any size, $1 \leq |S| \leq t$. That is, ϕ_t^0 and ϕ_t^t are two extreme classes. This section compares the power of distinct classes of failure detectors denoted ϕ_t^{y1} and ϕ_t^{y2}.

Definition 1 *For two classes of failure detectors A, B, we denote $A \leq B$, and say that B is* at least as strong *as A, if any failure detector in B can be used to build a failure detector in A. We also say that B is* stronger *than A (denoted $A < B$), if $A \leq B$ and $B \nleq A$. The classes A and B are* equivalent, *(denoted $A \equiv B$), if $A \leq B$ and $B \leq A$.*

We shall see that ϕ_t^{y1} is stronger than ϕ_t^{y2} if $y1 > y2$, since ϕ_t^{y1} provides more information about failures than ϕ_t^{y2}. Given a run of the processes, let $\text{QUERY}_y(S)$ be a failure detector query invocation that from some time on is indefinitely repeated. Let us examine the outputs returned by the infinite sequence of queries when the failure detector belongs to ϕ_t^{y1} and ϕ_t^{y2}, respectively. Notice that $t - y2 > t - y1$ (since $y1 > y2$).

- Case 1: $|S| > t$. Both outputs are systematically equal to *false*.

- Case 2: $|S| \leq t - y1$. Both outputs are systematically equal to *true*.

- Case 3: $t - y2 < |S| \leq t$ (so, we also have $t - y1 < |S| \leq t$), for a relevant query. If at least one process of S never crashes, both outputs are always equal to *false*. If all the processes of S crash, eventually both outputs are permanently equal to *true*.

- Case 4: $t - y1 < |S| \leq t - y2$. In this case, the output is always *true* if the failure detector belongs to the class Φ_t^{y2}. If it belongs to Φ_t^{y1}, the output is as in Case 3 (it depends on the failures).

The last case, namely, when $t - y1 < |S| \leq t - y2$, exhibits a noteworthy difference between ϕ_t^{y1} and ϕ_t^{y2}: ϕ_t^{y1} provides information on failures while ϕ_t^{y2} does not. Indeed, for $y1 > y2$, it is impossible to build a failure detector in ϕ_t^{y1} from one in ϕ_t^{y2}. On the other hand, any failure detector in ϕ_t^{y1} can be used to build a failure detector in ϕ_t^{y2}, by returning *true* if $|S| \leq t - y2$; returning *false* if $|S| > t$; and returning the output of ϕ_t^{y1} if $t - y2 < |S| \leq t$. (Formally, the next theorem is a consequence of Corollary 2 of Section 7.)

Theorem 1 $(y1 > y2) \Rightarrow (\phi_t^{y2} < \phi_t^{y1})$.

4   The Condition-Based Approach

The condition-based approach was introduced in [36] to study conditions restricting the inputs to consensus that make the problem solvable in an asynchronous system where t processes can crash. This line of research has been extended to study conditions for other problems, and in other distributed computing models [3, 17, 32, 36, 37, 39, 52]. In this paper, we are interested in conditions for the set agreement problem in an asynchronous system.

4.1 Conditions

Let $\mathcal{V}$ be the set of values that can be proposed by the processes. Moreover, let $\perp \notin \mathcal{V}$ be a default value. An *input vector* is a size n vector over $\mathcal{V} \cup \{\perp\}$. The input vector J *proposed* in an execution has in its i-th entry $J[i]$ the value of $\mathcal{V}$ proposed by p_i, or $\perp$ if p_i did not take any step in the execution. We usually denote by I an input vector with all entries in $\mathcal{V}$, and with J an input vector that may have some entries equal to $\perp$; such a vector J is called a *view*. The set $\mathcal{V}_x^n$ consists of all the input vectors with at most x entries equal to $\perp$, and $\mathcal{V}^n = \mathcal{V}_0^n$.

Definition 2 *A condition C is a subset of $\mathcal{V}^n$.*

Notation For any pair of vectors $J1, J2 \in \mathcal{V}_x^n$, $J1$ is *contained* in $J2$, denoted $J1 \leq J2$, if $\forall k : J1[k] \neq \perp \Rightarrow J1[k] = J2[k]$. Moreover, $J1 < J2$ if $J1 \leq J2$ and $J1 \neq J2$, which means that J_2 has at least one non-$\perp$ values that $J1$ does not have. Also, $\#_a(J)$ denotes the number of occurrences of a value a in the vector J, with $a \in \mathcal{V} \cup \{\perp\}$. For a set of input vectors $C \subseteq \mathcal{V}^n$, $\mathcal{C}_x$ is the set of all vectors J with at most x entries equal to $\perp$, and such that $J \leq I$ for some $I \in C$. Finally, $dist(J, J')$ is the Hamming distance separating J and J', where J and J' are two vectors of $\mathcal{V}_x^n$.

4.2 Legality of a Condition

The main result of the condition-based approach to solve asynchronous consensus is based on the following definition as formulated in [17, 52]:

Definition 3 *A condition C is x-legal if there exists a function $h : C \mapsto \mathcal{V}$ with the following properties:*

- $\forall I \in C$: $h(I) = a \Rightarrow \#_a(I) > x$*, and*
- $\forall I1, I2 \in C$: $h(I1) \neq h(I2) \Rightarrow dist(I1, I2) > x$.

A fundamental result of the condition-based approach is a characterization of the conditions C for which consensus can be solved (for a precise definition of solving consensus for C see Definition 5, with $k = 1$):

Theorem 2 [36] *There is a t-fault tolerant protocol solving consensus for C if and only if C is t-legal.*

A general method to define t-legal conditions is described in [40], and two natural t-legal conditions are described in [36].

It is convenient to extend h to vectors J with $\perp$ values. The lemma that follows shows that this is easy, provided $J \in \mathcal{C}_x$.

Lemma 1 *Let C be an x-legal condition, and $I1, I2 \in C$, $J \in \mathcal{C}_x$ such that $J \leq I1$ and $J \leq I2$. Then $h(I1) = h(I2)$.*

Proof Assume for contradiction that $h(I1) \neq h(I2)$. We have $dist(I1, I2) > x$ because C is x-legal. From the fact that J has at most x entries equal to $\perp$ and $J \leq I1$, we have $dist(J, I1) \leq x$ (similarly, we also have $dist(J, I2) \leq x$). From these inequalities, the fact that the entries of J that differ in $I1$ and $I2$ are only its $\perp$ entries, and again the fact that J has at most x entries equal to $\perp$, we conclude that $dist(I1, I2) \leq x$. A contradiction. $\square_{Lemma\ 1}$

Using this lemma we have a consistent definition:

Definition 4 *Let C be an x-legal condition, and J be any vector in $\mathcal{C}_x$. The function h is extended to J by taking any $I \in C$ with $J \leq I$ and letting $h(J) = h(I)$.*

Assuming up to t process crashes, and $-t \leq d \leq t$, let $\mathcal{S}_t^{[d]}$ be the set of all $(t - d)$-legal conditions (thus $\mathcal{S}_t^{[0]}$ consists of the t-legal conditions). It is easy to check that

$$\mathcal{S}_t^{[-t]} \subset \cdots \subset \mathcal{S}_t^{[-1]} \subset \mathcal{S}_t^{[0]} \subset \mathcal{S}_t^{[1]} \subset \cdots \subset \mathcal{S}_t^{[t]},$$

where $\mathcal{S}_t^{[t]}$ includes the condition made up of all the possible input vectors.

Notation   In the rest of the paper, C_t^d denotes a condition that belongs to $\mathcal{S}_t^{[d]}$.

4.3   The k-Set Agreement Problem

k**-Set agreement**   Consensus is a fundamental problem in distributed computing that is impossible in an asynchronous system even with a single crash failure. While consensus requires all processes to decide on the same value, k-*set agreement* [9] permits the processes to choose up to k different values. The problem is solvable when $k - 1$ processes can crash, but not when k can crash. The proofs [6, 29, 49] of this result uncovered a deep connection between distributed computing and topology, and motivated a significant amount of subsequent research.

The set of values $\mathcal{V}$ that can be proposed is assumed to be such that $|\mathcal{V}| > k$. Each process starts an execution with an arbitrary input value from $\mathcal{V}$, the value it *proposes*, and all correct processes have to decide on a value such that (1) any decided value has been proposed, and (2) no more than k different values are decided. The *consensus* problem is k-set agreement for $k = 1$.

Condition-based set agreement   We are interested in conditions C that, when satisfied, make k-set agreement solvable in an asynchronous system where at most t process can crash. As we shall see, k-set agreement is solvable for $C \in \mathcal{S}_t^{[d]}$, if $k \geq d + 1$.

Notice that if an input vector $J \in \mathcal{C}_t$ occurs in an execution of a protocol, then as far as the processes with non-$\perp$ values in J can tell, the input vector could belong to $I \in C$, because they cannot distinguish from another execution where the other processes wake-up and propose their values after the former processes have made their decision. Given $C \in \mathcal{S}_t^{[d]}$, we say that C is d-*satisfied* for input vector J if $J \in \mathcal{C}_t$ or $\#_\perp(J) \geq t - d$.

Definition 5 *A t-fault tolerant protocol solves the k-set agreement problem for a condition $C \in \mathcal{S}_t^{[d]}$ if in every execution with input vector J, the protocol satisfies the following properties:*

- Validity. *Every decided value is a proposed value[3].*
- Agreement. *No more than k different values are decided.*
- Termination. *Every correct process must decide if (1) C is d-satisfied for J and no more than t processes crash, or (2.a) a process decides, or (2.b) fewer than k processes crash.*

The first two are the safety requirements of the standard set agreement problem, and they should hold even if the input pattern does not belong to C. The third item requires termination under "normal" operating scenarios: (1) inputs that could belong to C or at least $t - d$ processes crash, and (2.a) executions where a process decides, or (2.b) fewer than k processes crash (a situation where k-set agreement is solvable without conditions).

Notice that if set agreement is solvable for a condition C, then it is solvable for any C' contained in C: the same protocol works. As mentioned above, when $t < k$, k-set agreement is solvable for $\mathcal{V}^n$, hence for any condition C.

5   Combining Conditions with ϕ_t^y to Solve Set Agreement

This section presents a set agreement protocol with access to a failure detector of the class ϕ_t^y, $0 \leq y \leq t$, and instantiated with the function h of a $(t - d)$-legal condition $C \in \mathcal{S}_t^{[d]}$, $0 \leq d \leq t$. It is a t-fault tolerant protocol that solves k-set agreement for C_t^d, where $k = 1 + \max(0, d - y)$ (recall Definition 5). Thus, all the pairs (d, y) such that $d \leq y$ allow solving condition-based consensus.

5.1   Base Objects

In order to make the protocol simpler to understand, it is presented in a modular way. More specifically, it relies on the following base objects: three arrays of atomic registers, a consensus object, an adopt-commit-abort object, and a condition-set agreement object. An adopt-commit-abort object and a condition-set agreement object can always be implemented on top of base read/write registers. As far as the consensus object is concerned, as we will see, it can always be implemented in the particular context in which it is used by the processes.

---

[3]It is shown in [13] that the solvability of k-set agreement is highly sensitive to the validity property adopted.

The shared memory   The shared memory is made up of three arrays (denoted $V[1..n]$, $W[1..n]$ and $DEC[1..n]$) of single-writer multi-reader atomic registers. All are initialized to $[\bot, \dots, \bot]$. The j-th entry of an array $X[1..n]$ can be read by any process, but only p_i can write to the i-th component $X[i]$. To simplify the presentation we assume that, in addition to these atomic read and write operations, a process p_i can also invoke the non-primitive operation snapshot(X) that allows it to read the content of all the registers of the array X as if this reading was done instantaneously. (Such an operation can be implemented in shared memory systems made up of single-writer, multi-readers atomic registers despite any number of process crashes ($1 \leq t < n$) [1, 4].) In accordance with the terminology defined in [30], the read, write and snapshot() operations are linearizable (i.e., they appear as if they had been executed one after the other, in agreement with their real-time occurrence order).

The underlying consensus object   A consensus object is accessed by a process p_i when p_i invokes the operation alg_cons(v_i) where v_i is the value proposed by p_i. Such an object allows any subset of processes to invoke alg_cons(). Its properties are the following:
- **Termination.** Any correct participating process decides a value.
- **Validity.** A decided value is a proposed value.
- **Agreement.** No two different values are decided.

As we will see, the underlying consensus object is used when more than $t - y$ process crash. It is shown in Theorem 8 that, in this case, a failure detector of the class $\mathcal{P}$ (the class of perfect failure detectors [8], see Section 8) can be built from a failure detector of the class ϕ_t^y (such a construction is described in the proof of Theorem 8), and consensus can be solved in a single-writer multi-reader atomic register asynchronous system enriched with such a failure detector[4].

The underlying adopt-commit-abort object   The adopt-commit-abort object we use here is a simple variant of the adopt-commit-abort object introduced in [19, 51] in the context of shared memory systems, and an object introduced in [42] in the context of message-passing systems[5]. Such an object has a single operation, denoted adopt_commit(). A process p_i invokes adopt_commit(v_i) where v_i is the value it proposes to the adopt-commit-abort object, and obtains a pair (d, v) as a result, where d is control tag and v a value. The object is defined by the following properties.
- **Termination.** Any correct participating process decides a pair (d, v).
- **Validity.** If a process decides (d, v), then $d \in \{commit, adopt, abort\}$, and v is a proposed value.
- **Agreement.** If a process decides $(commit, v)$, then any other process that decides, decides (d, v) with $d \in \{commit, adopt\}$.
- **Obligation.** If all the participating processes propose the same value v, then only the pair $(commit, v)$ can be decided.

Intuitively, the adopt-commit-abort object is an abortable variant of consensus. Let us observe that a process that decides $(abort, -)$, can conclude that no process decides $(commit, -)$. However, when a process decides $(adopt, -)$, it cannot conclude which control tag has been decided by the other processes.

The underlying condition-set agreement object   A condition-set agreement object has a single operation, denoted cond_algo(). This object is designed to solve a set agreement problem with the help of a $(t - d)$-legal condition C. A process uses this object only in the particular context where the input vector J is such $\#_\bot(J) \leq t - y$.

A process p_i invokes cond_algo(V_i) where V_i is its local view of the input vector J (we have then $V_i \leq J$ and $\#_\bot(V_i) \leq t - y$), and only when the views can be ordered by containment, $V_i \leq V_j$ or $V_j \leq V_i$ for all i, j. If it returns from that invocation, p_i obtains a value v. The object is defined by the following properties.
- **Termination.** Every correct process decides if (1) $J \in \mathcal{C}_t$ or $\#_\bot(J) \geq t - d$ (C is d-satisfied for J) or (2.a) a process decides, or (2.b) more than $(n - k)$ correct processes invoke cond_algo().
- **Validity.** A decided value is a value that has been proposed by a process in its input view.
- **Agreement.** At most $k = 1 + \max(0, d - y)$ values are decided.

---

[4]For completeness, a $\Diamond\mathcal{P}$-based alg_cons() protocol is described in Appendix A. (Trivially, any failure detector in $\mathcal{P}$ is also in $\Diamond\mathcal{P}$.) Others shared memory consensus algorirhms based on failure detectors can be found in [5, 24, 34].

[5]A wait-free implementation of an adopt-commit-abort object from single-writer multi-reader atomic registers can be found in [51]. An implementation in message-passing systems where a majority of processes is correct is presented in [42]. For completeness an implementation of an adopt-commit-abort object is described in Appendix B.

5.2   The Set agreement Protocol

Description of the protocol   The k-set agreement protocol based on a condition in $C \in \mathcal{S}_t^{[d]}$ and a failure detector of the class ϕ_t^y is described in Figure 1. The variables subscripted with i are local variables of p_i. A process is made up of two tasks: a main task $T1$, and a background task $T2$. The behavior of the task $T1$ can be decomposed into four parts.

- A process first writes the value v_i it proposes into $V[i]$ (line 1). Then, using the snapshot() operation, it reads the array of proposed values until that array contains "enough" values (line 2). "Enough" means here that there are no more than $(t - y)$ missing values, or there are more than $(t - y)$ process that have crashed; this is known from the invocation QUERY$_y(S_i)$. (Let us recall that, when $|S_i| \leq t - y$, QUERY$_y(S_i)$ answers always $true$).

- Then, the behavior of p_i depends on the number of values it knows. If there are too many crashes (line 4) p_i sets a local variable $prop_i$ to the value $CONS$ to try to decide a value from the underlying consensus algorithm (let us remind that, when there are more than $(t - y)$ crashes, it is possible to solve consensus from ϕ_t^y). In the other case, p_i knows enough proposed values to decide from the condition (line 5); p_i computes consequently a value w_i that could be decided from the condition, and sets $prop_i$ to $COND$.

- The process then uses the underlying adopt-commit-abort object (line 7) in order to try agreeing on the same tag, namely $CONS$ or $COND$. Moreover, each p_i deposits in the array $W[1..n]$ the value it has computed at line 4 or line 5, and reads that array with the snapshot() operation.

- The last part depends on the result returned by the adopt-commit-abort object.

    - If p_i obtains $d_i = abort$ or $res_i = CONS$, it concludes that no value can be decided from the condition. It consequently uses the consensus object to decide a value (lines 8-9).
    - If p_i obtains $res_i = COND$, at least one entry of W_i is not equal to $\perp$. Then, if additionally, $d_i = commit$, p_i concludes that any value deposited in W can be decided from the condition, and it decides it (lines 10-11).
    - If p_i obtains $res_i = COND$ together with $d_i = adopt$, it does not know if the other processes p_j have obtained $d_j = commit$ or $d_j = abort$. So, to be consistent, p_i participates in the underlying consensus to which it proposes a value that could be decided from the condition (lines 12-14). It then decides the value returned by the consensus object.

The aim of the task $T2$ is to guarantee that a correct process always decides as soon as a process decides. To that end, when a process p_j is about to decide in task $T1$ (execution of the return(v) statement), it first writes v in $DEC[j]$ [6]. The task $T2$ of a process p_i is then a simple loop statement that terminates when the predicate ($\exists j : DEC[j] \neq \perp$) becomes true. The execution of the return() statement by a process p_i terminates its execution of k-set_agreement$_{n,t}^{[d,y]}$ (v_i).

Proof of Correctness

Theorem 3   *When instantiated with a failure detector of the class ϕ_t^y and a condition C in $S_t^{[d]}$, the protocol described in Figure 1 solves the condition-based k-set agreement problem where $k = 1 + \max(0, d - y)$.*

Proof   Validity property (a decided value has been proposed by a process). Let us observe that a decided value dec_i is either an initial value v_j proposed to the consensus by a process p_j (line 9) or a value w_i obtained by a process p_i from the condition-set agreement object (lines 11 and 14). The validity property follows immediately from the corresponding validity property of the consensus object and the condition-set agreement object, and hence line 18 preserves validity.

Agreement property (at most k different values are decided). Let us observe that, due to the agreement property of the adopt-commit-abort object, it is not possible for two processes p_i and p_j that have invoked adopt_commit() at line 7, to be such that both the predicate ($res_i = CONS$) $\vee$ ($d_i = abort$) and the predicate ($res_j = COND$) $\wedge$ ($d_j = commit$) are true. It follows from that observation that it is not possible for a process p_i to execute line 9 while another process p_j executes line 11. So, there are only two cases to consider (in addition to the trivial case of line 18).

---

[6]This write plays the same role as the reliable broadcast of the decided value in message-passing systems (e.g., see the consensus protocols in [8, 23, 42, 50]). Their aim is to prevent a process from deadlocking.

```
Function k-set_agreement_{n,t}^{[d,y]} (v_i)

task T1:
(1)    V[i] ← v_i;
(2)    repeat V_i ← snapshot (V); S_i ← {j | V_i[j] = ⊥}
(3)                   until QUERY_y (S_i) end repeat;
(4)    case (#_⊥(V_i) > t − y) then prop_i ← CONS; w_i ← ⊥
(5)          (#_⊥(V_i) ≤ t − y) then prop_i ← COND; w_i ← cond_algo(V_i)
(6)    end case;
(7)    W[i] ← w_i; (d_i, res_i) ← adopt_commit(prop_i); W_i ← snapshot(W);
(8)    case (res_i = CONS) ∨ (d_i = abort)
(9)                  then DEC[i] ← cond_algo(v_i); return (DEC[i])
(10)        (res_i = COND) ∧ (d_i = commit)
(11)                  then DEC[i] ← W_i[j] such that W_i[j] ≠ ⊥; return(DEC[i])
(12)        (res_i = COND) ∧ (d_i = adopt)
(13)                  then est_i ← W_i[j] such that W_i[j] ≠ ⊥;
(14)                        DEC[i] ← cond_algo(est_i); return(DEC[i])
(15)   end case

task T2:
(16)   j ← 0;
(17)   repeat forever j ← (j mod n) + 1;
(18)                  if (DEC[j] ≠ ⊥) then return(DEC[j]) end if
(19)   end repeat
```

Figure 1: A k-set agreement protocol with $k = 1 + \max(0, d - y)$

- No process begins executing 9 or 14. In that case, these processes decide the value returned by the consensus object. Due to the consensus agreement property, there is a single such value.
- No process begins executing line 11 or 14. In that case, these processes decide a value returned from the condition-set agreement object . Due to the condition agreement property, there are at most $k = 1 + \max(0, d - y)$ such values, which proves the case.

Termination property. Let J be the input vector. We have to show that every correct process decides if (1) the condition C is d-satisfied for J, or (2.a) a process decides, or (2.b) fewer than k processes crash.

Let us notice that, as there are at most t process crashes (model assumption), the repeat loop of lines 2-3 always terminates. Moreover, let us also observe that, due to the termination property of the adopt-commit-abort object, any invocation of adopt_commit() issued by a correct process terminates (observation $O1$).

Let us also observe that the underlying consensus protocol is used only when the number of crashes is greater than $t - y$ (line 4), i.e., when a failure detector of the class $\mathcal{P}$ can be built from a failure detector of the class ϕ_t^y (Item 3 of Theorem 8). The termination property of the consensus object ensures that all the correct processes that invoke cons_alg() terminate their operation (observation $O2$). Let us now proceed by a case analysis.

- Case (1): We have to show that any correct process decides when the condition C is d-satisfied for J, where J is the input vector.

  In that case, it follows from Item (1) of the termination property of the condition-set agreement object that any invocation cond_algo() issued by a correct process terminates. This, combined with the observations $O1$ and $O2$, allows to conclude that any correct process terminates when C is d-satisfied for J.

- Case (2.a): We have to show that any correct process decides, as soon as a process decides.

  This property is trivially guaranteed by the management of the array $DEC[1..n]$ and task $T2$.

- Case (2.b): We have to show that any correct process decides when fewer than k processes crash.

  If no correct process accesses the condition-set agreement object, the fact that any correct process decides follows from the observations $O1$ and $O2$. So, let us consider the case where correct processes access the condition object. As $k = 1 + \max(0, d - y)$ and fewer than k processes crash, this means that at most $\max(0, d -$

y) processes crash. Moreover, as $t \geq d$, we have $t - y \geq d - y$. It follows (from the properties of the failure detector ϕ_t^y) that all the processes p_i that execute the repeat loop (lines 2-3) and do not crash while executing that loop, eventually exit it, and we have then $\#_\perp(V_i) \leq t - y$. They consequently all access the condition-set agreement object at line 5. It follows that all the correct processes (there are more than $(n - k)$ of them) invoke the condition-set agreement object. Due to item (2.b) of the termination property of the condition-set agreement object, it follows that any correct process decides in the k-set agreement protocol.

$\square_{Theorem\ 3}$

5.3   Implementation of a Condition-Set Agreement Object

Description of the protocol   A t-fault tolerant protocol implementing a condition-set agreement object is described in Figure 2. This protocol is instantiated with a function h associated with a $(t - d)$-legal condition C. It uses a deterministic function $F()$ and a predicate $P()$. The function $F()$ takes a view J as a parameter and returns a non-$\perp$ value of the vector J. The value $\top$ is a default value not in $\mathcal{V}$ and different from $\perp$. It is assumed that the function h is extended to all views J of C with at most $t - d$ entries equal to $\perp$ as in Definition 4. The predicate $P()$ is true on all such views:

$$P(V_i) \equiv \left(\exists I \in C \text{ such that } V_i \leq I \right).$$

Thus, $P()$ is used to test if p_i's current view V_i of the input vector could originate from a vector of the condition[7].

```
Function cond_algo (V_i)     % We have #_⊥(V_i) ≤ t − y, and V_i ≤ V_j or V_j ≤ V_i for all i, j %

(1)   if (#_⊥(V_i) ≤ t − d)
(2)      then if P(V_i)
(3)           then % The processes executing this line decide the same value %
(4)                w_i ← h(V_i); D[i] ← w_i; return (w_i)
(5)           else if (#_⊥(V_i) = t − d)
(6)                then % The processes executing this line decide the same value %
(7)                     w_i ← F(V_i); D[i] ← w_i; return (w_i)
(8)                else % If processes execute these lines, at most k value can be decided %
(9)                     D[i] ← ⊤;
(10)                    repeat D_i ←snapshot (D) until ( (∃j : D_i[j] ≠ ⊥, ⊤) ∨ (#_⊥(D_i) < k) );
(11)                    if (∃j :  D_i[j] ≠ ⊥, ⊤) then return (D_i[j] such that D_i[j] ≠ ⊥, ⊤)
(12)                                             else  ∀j : if (D_i[j] = ⊤) then Y_i[j] ← V[j]
(13)                                                                        else Y_i[j] ← ⊥ end_if;
(14)                                                   w_i ← F(Y_i); D[i] ← w_i; return (w_i)
(15)                    end_if
(16)               end_if
(17)          end_if
(18)      else w_i ← F(V_i); D[i] ← w_i; return (w_i) % Here (t − d) < (#_⊥(V_i) ≤ t − y) %
(19)          % The processes executing that line decide at most max(0, d − y)) values %
          end_if
```

Figure 2: A condition protocol

The protocol can be seen as a case analysis. The first step is for p_i to check whether $\#_\perp(V_i) \leq t - d$ in order to benefit from the condition. If $\#_\perp(V_i) > t - d$, p_i cannot benefit from it and consequently decides a value from its local view V_i at line 18 (the processes executing that line decide at most $\max(0, d - y)$ different values).

Otherwise we have $\#_\perp(V_i) \leq t - d$, and then p_i has enough non-$\perp$ entries in its view V_i to test if the condition can help it decide. So, p_i enters the lines 2-17. There are three cases. If $P(V_i)$ is satisfied (first case), p_i decides the value from the condition and writes it in the shared array D to help other processes decide (line 4).

If $P(V_i)$ is not satisfied (second case) p_i first checks if $\#_\perp(V_i) = t - d$. If so (second case), it knows that no other process will evaluate P to true in the previous line, and that any other process p_j with $\#_\perp(V_j) = t - d$ has $V_i = V_j$, so it deterministically decides $F(V_i)$ (line 7).

---

[7]It is shown in [36] that, for some conditions, there are very efficient ways to compute the predicate $P()$. As an example, for the $(t - d)$-legal condition $C1$ (defined in Section 4.2), we have $P(V_i) \equiv \#_{\max(J)}(V_i) > (t - d) - \#_\perp(V_i)$.

In the third case, $\#_\perp(V_i) < t - d$ and p_i writes $\top$ in $D[i]$ to indicate it cannot decide from its local view V_i (so, $D[j] = \perp$ means that p_j has not yet finished executing its protocol or has crashed). Then, as it cannot decide by itself, p_i starts the "best effort termination" part of the protocol (lines 9- 15). It enters a loop (line 10), during which it looks for a decided value ($\exists j : D_i[j] \neq \perp, \top$) and decides it if there is one (line 11), or a configuration where $\#_\perp(D_i) < k$ (this is the only place where k is used in the protocol). If the condition ($\nexists j : D_i[j] \neq \perp, \top$) $\wedge$ ($\#_\perp(D_i) < k$) is satisfied, p_i builds a local view of the input vector corresponding to the processes that have executed at least until line 9. As we will see in the proof, if several such views (Y_i, Y_j, etc.) are computed, due to the invocations of snapshot(D) at line 10 that precede their construction the associated containment property implies these views (Y_i, Y_j, etc.) are also ordered by containment. The process p_i then decides the value $F(Y_i)$. Let us notice that, as $\#_\perp(D_i) < k$, the vector Y_i has at most $k - 1$ entries equal to $\perp$. It follows that at most k different values can be decided at line 14. Let α be the number of values decided at line 4, 7, 11 and 18, and β be the number of values decided at lines 14. The proof will show that $\alpha + \beta \leq k$.

Correctness proof

Theorem 4 *When instantiated with a $(t - d)$-legal condition C, the protocol described in Figure 2 implements a condition-set agreement object with $k = 1 + \max(0, d - y)$.*

Proof Validity property (a decided value is a value proposed in the input view of a process). This property follows directly from the fact that both the function $h()$ and the function $F()$ extracts a non-$\perp$ value from the vector they are applied to.

Agreement property (at most $k = 1 + \max(0, d - y)$ different values are decided). The processes that decide, do it at line 4, 7, 11, 14, or 18. We determine the maximum number of values that are decided by the processes at each of these lines of the protocol.

- Consider the processes that decide at line 4.
  These processes p_i are such that ($\#_\perp(V_i) \leq t - d$) and $P(V_i)$ is satisfied. We show that a single value can be decided at line 4.

  Let p_i and p_j be two processes that decide at line 4. Due to the use of a snapshot operation, we have either $V_i \leq V_j$ or $V_j \leq V_i$. Let us consider that $V_i \leq V_j$.
  We then have (1) $\#_\perp(V_i) \leq t - d$ and $\#_\perp(V_j) \leq t - d$, (2) both $P(V_i)$ and $P(V_j)$ are satisfied, i.e., $\exists I1 \in C$ such that $V_i \leq I1$, and $\exists I2 \in C$ such that $V_i \leq V_j \leq I2$, (3) and the condition is $(t - d)$-legal. It follows from these three items and Lemma 1 that $h(V_i) = h(V_j) = h(I_1) = h(I_2)$. Consequently, no more than one value can be decided by the processes executing line 4.

- Consider the processes that decide at line 7.
  These processes p_i are such that ($\#_\perp(V_i) = t - d$) and $P(V_i)$ is not satisfied. In this case at most one value is decided at line 7, because due to the snapshot containment property, all processes that execute this line have exactly the same view V_i. Moreover, if a process executes this line, no process executes line 4. This is because any process p_j that executes line 4 has a view V_j such that ($\#_\perp(V_j) \leq t - d = \#_\perp(V_i)$) and as we have either $V_i < V_j$ or $V_j < V_i$ we conclude that $V_i < V_j$. Consequently, if p_i executes line 7, $P(V_i)$ is false, and hence $P(V_j)$ is also false as $V_i \leq V_j$, by definition of the predicate $P()$.

- Consider the processes that decide at line 18.
  These processes p_i are such that ($t - y \geq \#_\perp(V_i) > t - d$). We show that these processes decide at most $\max(0, d - y)$ different values.

  Due to the containment property on the vectors provided by the snapshot operation, any pair of processes p_i and p_j that execute line 18 are such that $V_i \leq V_j$ (or $V_j \leq V_i$). We conclude from that observation that the processes that execute line 18 have at most $\max(0, (t - y) - (t - d))$=$\max(0, d - y)$ different vectors. As F is deterministic, at most $\max(0, d - y)$ different values can be decided by the processes that decide at line 18.

  It follows that, when we consider the processes that decide at line 4, 7 or 18, at most $k = 1 + \max(0, t - d)$ different values can be decided.

- Consider the processes that decide at line 11.
  A process p_i that decides at line 11, decides a value (that it retrieves in $D[j]$) that has been decided by another process p_j (p_j has deposited that value in $D[j]$ at line 4, 7, 14 or 18). Consequently, no additional value can be decided at line 11.

- Finally, consider the processes that decide at line 14.
  Let β be the number of different values decided by the processes that execute line 14. Let α be the number of values decided by the processes that execute line 4, 7 or 18. We claim that $\alpha + \beta \leq k = 1 + \max(0, t - d)$.

It follows from this case analysis that at most $k = 1 + \max(0, t - d)$ different values can be decided, which proves the theorem.

Proof of the claim. Let us consider two time instants t_0 and t_1 defined as follows:
- t_0 = first time instant where $\#_\perp(D) = k - 1$ (or $+\infty$ if it never happens),
- t_1 = first time instant where $\exists\, D[j] \notin \{\top, \perp\}$ (or $+\infty$ if it never happens)[8].

Let us first consider $t_1 \leq t_0$. Let us notice that a process p_i stops the repeat loop of line 10 as soon as $(\exists\, D_i[j] \notin \{\top, \perp\} \ \vee\ \#_\perp(D_i) < k)$. As the test that (at line 11) immediately follows the exit of the repeat loop privileges the case $(\exists\, D_i[j] \notin \{\top, \perp\})$ with respect to the case $(\#_\perp(D_i) < k)$ when both are satisfied, it follows that p_i immediately executes the return () statement at line 11. Consequently, when $t_1 \leq t_0$, any process p_i that enters the loop of line 10 and then decides, decides at line 11. We then have $\beta = 0$ (no process decides at line 14).

Let us now consider the case $t_0 < t_1$. Let us first observe that, since the function $F()$ is deterministic and each Y_i computed at lines 12-13 contains at most $(k - 1)$ entries equal to $\perp$, it follows that the β values decided at line 14 correspond to (at least) β different Y_i vectors, which means (due to line 12) at least β different D_i vectors.

Due to the containment property of the invocations of the snapshot (D) invocations at line 10, the previous β D_i vectors are totally ordered (see the definition of "<" in Section 4.1), e.g., $D_{i1} < D_{i2} < \cdots < D_{i\beta} < \cdots$ and contain only $\perp$ and $\top$ entries. Moreover, for any pair of such vectors there is at least one entry which is equal to $\perp$ in one vector and to $\top$ in the other. As D is initialized to $[\perp, \ldots, \perp]$, and there are at least β different D_{ix}, we conclude that at least $(\beta - 1)$ values $\top$ have been written into D after t_0 (because, due to the snapshot (D) operations, we have $\#_\perp(D_{i1}) \leq k - 1$ at time t_0, $\#_\perp(D_{i2}) \leq k - 2$ at time t'_0, $t'_0 > t_0$, etc.).

Before being decided, the α different values decided at lines 4, 7 and 18 have been written into the array D (they are decided after t_1). Due to the definition of t_1, they have been written into D at or after t_1, i.e. (from the case assumption), after t_0.

Hence, after t_0, α entries of D have been set to proposed values by lines 4, 7 and 18, and $(\beta - 1)$ entries have been set to $\top$. As, at t_0, the number of entries of D that were equal to $\perp$ was equal to $(k - 1)$, it follows that $\alpha + (\beta - 1) \leq (k - 1)$, i.e., $\alpha + \beta \leq k$, which proves the claim when $t_0 < t_1$. *End of the proof of the claim.*

Termination property (let J be the actual input vector. Every correct process decides if (1) $J \in \mathcal{C}_t$ or $\#_\perp(J) \geq t - d$ -C is d-satisfied for J- or (2.a) a process decides, or (2.b) more than $(n - k)$ correct processes invoke cond_algo()).

If the input vector V_i, $V_i \leq J$, is such that $\#_\perp(V_i) > t - d$, the process p_i trivially decides at line 18. When $\#_\perp(V_i) = t - d$, the test on line 5 leads to termination. On another side, if $\#_\perp(V_i) \leq t - d$ and $J \in \mathcal{C}_t$, then $P(V_i)$ is satisfied and p_i decides at line 4. So, the case (1) is done.

Let us consider case (2.a). Before deciding a value at line 4, 7, 14, or 18, a process deposits that value in the array D. It follows that, after a process has decided, the repeat loop of line 10 always terminates, and any process that executes line 11 decides, which proves the case.

Let us finally consider case (2.b). Let us assume that more than $(n - k)$ correct processes invoke the object and no one decides. This means that none of them executes line 4, 7, or 18. They all consequently enter the repeat loop at line 10, from which we conclude that eventually the predicate $\#_\perp(D_i) < k$ becomes true. It follows that the correct processes exit the repeat loop and decide. $\qquad\square_{Theorem\ 4}$

---

[8]If both t_0 and t_1 are equal to $+\infty$, no process decides and the claim is trivially true.

6 Discussion

6.1 Initial Crashes: No Condition is Needed

The theorem that follows considers a particular case, namely the case where the faulty processes crash before the protocol starts its execution.

Theorem 5 *Consider an execution of the protocol described in Figure 1 instantiated with a failure detector of the class ϕ_t^y. Let us assume that more than $(t - y)$ processes have crashed before the protocol starts. The protocol then solves the consensus problem (whatever the $(t - d)$-legal condition it is instantiated with).*

Proof If more than $(t - y)$ processes have initially crashed, due to the property of the QUERY$_y$ () invocations at line 3, we have $(\#_\perp(V_i) > t - y)$ for any process p_i. It follows that, for any process p_i, we have $prop_i = CONS$ (line 4). Due to the obligation property of the adopt-commit-abort object, every process p_i obtains $(commit, CONS)$. Consequently, all the processes invoke the underlying consensus object, which proves the theorem. $\square_{Theorem\ 5}$

Remark The previous theorem considers the case where the faulty processes have crashed before the protocol starts. It is interesting to observe that a similar result appears in [16], where a consensus protocol is presented for asynchronous systems where a majority of processes are correct, and the faulty processes crash before the protocol starts its execution.

6.2 An Always Terminating Version of the Protocol

It is possible to trade safety for liveness by providing a version of the protocol where every correct process always decides. This can be obtained at the price of an enlarged set of possibly decided values. More precisely, let I be an input vector, and C the $(t - d)$-legal condition the protocol is instantiated with. When $I \in C$, at most $k = 1 + \max(0, d - y)$ values are decided; when $I \notin C$, up to $k' = t + 1 - y$ values can be decided. Interestingly, this always terminating version of the protocol provides a new insight into the way the parameters t, y (power of the failure detection) and d (power of the condition) are related.

   In the protocol described in Figure 2, the statement that can prevent a correct process p_i from terminating is the repeat loop at line 10. This occurs when p_i enters the lines 9-15, V_i being such that $\#_\perp(V_i) \leq \min(t - y, t - d)$ (assumption on the input parameter and line 1), while $P(V_i)$ is equal to *false* (line 2). The modification to get an always terminating cond_algo() protocol is very simple: it consists in replacing the lines 9-15 in Figure 2 by a weakened statement that always terminates, namely:

   [9-15]'   **if** $(\exists j : D[j] \neq \perp)$ **then** return $(D[j]$ such that $D[j] \neq \perp)$
                                  else $w_i \leftarrow F(V_i)$; $D[i] \leftarrow w_i$; return (w_i)
             end if

Theorem 6 *Let us consider the protocol depicted in Figure 2 instantiated with a $(t - d)$-legal condition C, where the lines 9-15 are replaced by the statement [9-15]'. Every correct process decides. Let I be an input vector. If $I \in C$, at most $k = 1 + \max(0, d - y)$ values are decided. If $I \notin C$, at most $k' = t + 1 - y$ values can be decided.*

Proof Every correct process trivially terminates, and a decided value comes from a proposed vector (same proof as in Theorem 4).

   As far as the number of values that are decided is concerned, let us first consider the case where the input vector belongs to the condition. In that case, when a process p_i executes line 2, $P(V_i)$ is trivially satisfied. It follows that the new line [9-15]' is never executed. Consequently, Theorem 4 remains valid when $I \in C$, and at most $k = 1 + \max(0, d - y)$ values are then decided.

   Considering now the case where the input vector does not belong to the condition, let us first observe that if a process p_i decides at line [9-15]' a value $D[j]$ such that $D[j] \neq \perp$, it does not decide a new value as $D[j]$ is counted as a decided value at line 4, 7, 18 or in the **else** part of the new **if** statement. So, let us count the number of values that can be decided by the processes executing line 4 or the **else** part of the new line [9-15]'. For each such process p_i we have $\#_\perp(V_i) \leq t - \max(y, d + 1)$. Moreover (due to the containment property on the vectors V_i provided by

the cond_algo() invocations we have $V_i \leq V_j$ (or $V_j \leq V_i$) for two processes executing line 4 or the **else** part of line [9-15]'. It then follows that there are at most $k1 = t - \max(y, d+1) + 1$ different vectors V_i for the processes that execute line 4 or the **else** part of line [9-15]'. Let us observe that if a process p_j decides at line 4, the same vector V_j will not be used to decide another value at line [9-15]'. Finally, due to that observation and the fact that $F()$ is deterministic, at most $k1$ different values can be decided by the processes executing line 4 or the **else** part of line [9-15]'. On the other side, the processes that execute line 18 decide at most $k2 = \max(0, d-y)$ different values (the proof is the same as the corresponding proof in Theorem 4). Recall that all the processes that execute line 7 decide the same value. Finally, summing up, we get $k' = k1 + k2 + 1$, i.e., $k' = \big(t - \max(y, d+1) + 1\big) + \big(\max(0, d-y)\big) + 1)$, which can be simplified to provide $k' = t + 1 - y$. $\square_{Theorem\ 6}$

Let us notice that when the input vector does not belong to the condition, the maximal number of values that can be decided, namely, $k' = t + 1 - y$, does not depend on d. If the information on failure is maximal ($y = t$), the protocol solves consensus. At the other extreme, if there is no information on failures ($y = 0$), and there is no power provided by the condition, the protocol solves the trivial version of the set agreement problem, namely $k' = t + 1$.

7 A Lower Bound

This section presents a lower bound matching Theorem 3.

Theorem 7 *When instantiated with a failure detector of the class ϕ_t^y and a $(t - d)$-legal condition, no protocol solves the condition-based k-set agreement problem for $k \leq \max(0, d - y)$.*

Proof Assume for contradiction that a protocol solves the k-set agreement problem for $k \leq \max(0, d - y)$. Hence, $d > y$ and $\max(0, d - y) = d - y$. Partition the processes in two groups, the *main* processes $p_1, \ldots, p_{n-t+d}$ and the *secondary* processes, $p_{n-t+d+1}, \ldots, p_n$. Consider the executions where the secondary processes crash before taking any steps. These are executions with at least $t - d$ failures. By Definition 5, all correct process must decide whatever the input vector. Now, consider the subset of these executions with at most $d - y$ additional failures. The total number of failures is at most $t - y$ failures. Recall that any relevant query is invoked with a set the size of which is greater than $(t - y)$. So, all relevant invocations $\text{QUERY}_y()$ issued by the main processes will include at least one correct process and thus will return *false* and all other invocations return the trivial output. Thus, in these executions, the failure detector gives no information, and therefore, the main processes have to solve the standard set agreement problem (i.e. terminate for every input vector), tolerating $d - y$ failures. The results of [6, 28, 29, 49] (more specifically, Corollary 5.5 in [28]) imply that in one of these executions at least $d - y + 1$ different values are decided, a contradiction. $\square_{Theorem\ 3}$

The following corollaries are direct consequences of the previous theorem. They consider the extreme cases where there is either no failure detector (i.e., $y = 0$), or no condition (i.e., $d = t$). The first corollary answers an open problem stated in [3, 39]. The second corollary shows the optimality[9] of ϕ_t^y.

Corollary 1 *Let C be a $(t - d)$-legal condition. There is no condition-based k-set agreement protocol for C when $k \leq d$.*

Corollary 2 *When considering the family $(\phi_t^y)_{0 \leq y \leq t}$ of failure detector classes, ϕ_t^y with $y = t - k + 1$, is the weakest that allows solving the k-set agreement problem.*

8 Comparing ϕ_t^y with the Chandra and Toueg's Failure Detector Classes

8.1 Chandra and Toueg's Failure Detector Classes

This section presents the failure detectors introduced by Chandra and Toueg, used in this paper. These classes are defined from the following completeness and accuracy properties [8]:

---

[9]This result complements another k-set agreement minimality result [27] which shows that, among the family $(\mathcal{S}_x)_{1 \leq x \leq t+1}$ of perpetual failure detectors (introduced in [43, 51]), $\mathcal{S}_x$ is the weakest to solve the k-set agreement problem for $k > t - x + 1$.

- *Strong (Weak) completeness.* Eventually, every process that crashes is permanently suspected by every (some) correct process.
- *Perpetual strong accuracy.* No process is suspected before it crashes.
- *Eventual strong accuracy.* There is a time after which no correct process is suspected.
- *Perpetual weak accuracy.* Some correct process is never suspected.
- *Eventual weak accuracy.* There is a time after which some correct process is never suspected.

The classes we are interested in are the following [8]. They are collectively called "Chandra and Toueg's failure detector classes" in the rest of the paper.

- $\mathcal{P}$: The class of *perfect* failure detectors. It includes all the failure detectors satisfying strong completeness and perpetual strong accuracy.

- $\mathcal{S}$: The class of *strong* failure detectors. It includes all the failure detectors satisfying strong completeness and perpetual weak accuracy. We have $\mathcal{P} \subseteq \mathcal{S}$.

- $\Diamond\mathcal{P}$: The class of *eventually perfect* failure detectors. It includes all the failure detectors satisfying strong completeness and eventual strong accuracy. We have $\mathcal{P} \subseteq \Diamond\mathcal{P}$.

- $\Diamond\mathcal{S}$: The class of *eventually strong* failure detectors. It includes all the failure detectors satisfying strong completeness and eventual weak accuracy. We have $\Diamond\mathcal{P} \subseteq \Diamond\mathcal{S}$, and $\mathcal{S} \subseteq \Diamond\mathcal{S}$.

The class $\Diamond\mathcal{S}$ is the weakest that allows solving the consensus problem, and is equivalent to the class $\Diamond\mathcal{W}$ in shared memory systems and in message passing systems with reliable channels [8, 7]. It has also been shown that $\Diamond\mathcal{S}$ and the class of *leader* failure detectors, denoted Ω, are equivalent in systems where each process initially knows all the process identities [7, 11, 41].

Figure 3 summarizes Chandra and Toueg's failure detector classes. Following Definition 1, an arrow from A to B means that $A \geq B$ (any failure detector of the class A can be used to build a failure detector of the class B). The absence of a path from A to B means that it is not the case $A \geq B$ (given any failure detector of the class A, it is not possible to build a failure detector of the class B). Finally, $A \equiv B$ if $A \leq B$ and $B \leq A$. The figure follows from [7, 8].
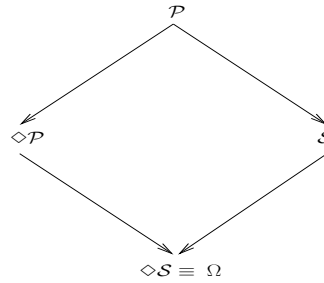

Figure 3: Relations among Chandra-Toueg's failure detector classes

8.2 ϕ_t^y with Respect to Chandra and Toueg's Failure Detector Classes

This section studies the relation between ϕ_t^y and the classic failure detectors introduced by Chandra and Toueg. We show that ϕ_t^t allows building a perfect failure detector, namely, $\mathcal{P} \leq \phi_t^t$ (Theorem 8). Therefore, $\mathcal{P}$ is equivalent to ϕ_t^t as $\phi_t^t \leq \mathcal{P}$ (from their definitions).

Theorem 8 (1) $\mathcal{P} \equiv \phi_t^t$. *Let f denote the actual number of process crashes in a run.* (2) *If $f \leq t-y$, ϕ_t^y, $0 \leq y \leq t-1$ does not allow building a failure detector of any of Chandra and Toueg's failure detector classes (e.g. $\Diamond\mathcal{S}$, $\Diamond\mathcal{P}$, Ω).* (3) *If $f > t - y$, ϕ_t^y, $0 \leq y \leq t - 1$ allows building a failure detector of the class $\mathcal{P}$.*

Proof Let us first consider item (1). The construction described in Figure 4 constructs a perfect failure detector from a failure detector of the class ϕ_t^t. This construction works as follows. A process that queries the perfect failure detector obtains the current value of the set $suspected_i$. As $y = t$, $\text{QUERY}_t(S)$ -where S is made up of a single process p- eventually returns $true$ if and only p has crashed. The strong completeness and strong accuracy properties defining the class $\mathcal{P}$ follow. Moreover, $\phi_t^t \leq \mathcal{P}$ follows directly from their definitions. Therefore, $\mathcal{P}$ is equivalent to ϕ_t^t.

---

init: $suspected_i \leftarrow \emptyset$

repeat forever for all j such that $p_j \in (\{p_1, \ldots, p_n\} \setminus suspected_i)$ **do**
    if $\text{QUERY}_t(\{p_j\})$ **then** $suspected_i \leftarrow suspected_i \cup \{p_j\}$ **end if**
end repeat

Figure 4: From ϕ_t^t to $\mathcal{P}$ (algorithm for p_i)

For proving item (2), let us first observe that an implementation that systematically suspects all the processes trivially satisfies the completeness property of any of Chandra and Toueg's failure detector classes, but prevents its accuracy property from being satisfied. So, assuming that ϕ_t^y ($0 \leq y \leq t - 1$) allows implementing the accuracy property of any of Chandra and Toueg's failure detector classes, we show that it does not allow implementing the associated (weak or strong) completeness property.

Let us consider any run during which no more than $x = t - y$ ($1 \leq x \leq t$) processes crash. Due to the definition of ϕ_t^y, we have the following:

- Any $\text{QUERY}_t(S)$ where $|S| \leq t - y = x$ always returns $true$ whatever the x (≥ 1) processes composing S. This follows from the triviality property of ϕ_t^y, $|S| \leq t - y = x$.

- Any $\text{QUERY}_t(S)$ where $|S| > x$ always returns $false$ whatever the processes composing S. This follows from the safety property of ϕ_t^y, as at least one process among these processes has not crashed.

These observations show that, when no more than $x = t - y$ ($1 \leq x \leq t$) processes crash, the boolean value returned by a query depends only on the number of processes defining S (it depends neither on which processes are in S, nor on the failure pattern). It follows that, when no more than $x = t - y$ ($1 \leq x \leq t$) processes crash, there is no way for a process to know if a given process has crashed or not, thereby making impossible to implement the (weak or strong) completeness property of any of Chandra and Toueg's failure detector classes.

The proof of item (3) consists in designing an algorithm that, in runs where $f > t - y$, builds a failure detector of the class $\mathcal{P}$ from a failure detector of the class ϕ_t^y. Let us first observe that, as $f > t - y$, there is a set S such as $|S| = t - y + 1$ and, after some finite time, $\text{QUERY}_y(S)$ returns $true$ forever. The algorithm is the following.

- Each set $suspected_i$ is initialized to $\emptyset$. Initially, each process p_i issues $\text{QUERY}_y(X)$ for all the possible sets X of size $|X| = t - y + 1$ until such a query returns $true$. Due to the fact that all the queries are relevant ($t - y < |X| \leq t$), and the previous observation, this eventually happens. When it occurs, p_i considers the corresponding set (say S) and executes $suspected_i \leftarrow S$.
- Then, for each $p_j \notin suspected_i$, p_i regularly executes $\text{QUERY}_y(S \cup \{p_j\})$. If the query returns $true$, p_i can conclude from the property of ϕ_t^y that p_j has crashed. It consequently adds p_j to $suspected_i$. Otherwise p_i keeps on issuing $\text{QUERY}_y(S \cup \{p_j\})$.

It follows from the definition of S and the safety and liveness properties of ϕ_t^y that the sets $suspected_i$ of the correct processes eventually include all the crashed processes, and never includes a "not yet" crashed process, i.e., they satisfy the properties that define the class $\mathcal{P}$ of perfect failure detectors [8]. $\square_{Theorem\ 8}$

8.3   From Chandra and Toueg's Failure Detectors to ϕ_t^y

Figure 5 presents a simple protocol transforming any failure detector of the class $\mathcal{P}$ into a failure detector of the class ϕ_t^y. The underlying set $suspected_i$ satisfies (by assumption) the properties defining the class $\mathcal{P}$. In contrast, we show that there is no protocol transforming any failure detector of the class ϕ_t^y, for $y < t$, into a failure detector of the class $\mathcal{P}$.

```
when QUERY_y (S) is invoked by p_i:
    case (|S| ≤ t − y)      then return (true)
         (|S| > t)          then return (false)
         (t − y < |S| ≤ t)  then return (S ⊆ suspected_i)
    end case
```

Figure 5: From $\mathcal{P}$ to ϕ_t^y (algorithm for p_i)

Theorem 9 *The protocol of Figure 5 transforms any failure detector of the class $\mathcal{P}$ into a failure detector of the class ϕ_t^y for $0 \le y \le t$.*

Proof The triviality property of ϕ_t^y is ensured by the first two case statements. The safety property follows from the fact that, due to the perpetual strong accuracy of the underlying failure detector, $suspected_i$ contains only crashed processes. Finally, the liveness property of ϕ_t^y follows from the fact that, due to completeness of the underlying failure detector, the set $suspected_i$ eventually contains all crashed processes. $\square_{Theorem\ 9}$

The next theorem states that there is no protocol transforming a failure detector of the class $\mathcal{S}$, $\diamondsuit\mathcal{P}$, $\diamondsuit\mathcal{S}$ or $\diamondsuit\mathcal{W}$ into a failure detector of the class ϕ_t^y for $0 < y$. It is surprising that these failure detectors are not strong enough to implement ϕ_t^y even when $y < t$, as in this case ϕ_t^y cannot solve consensus (Corollary 2), while these failure detectors can solve consensus. (In the case of $y = t$ both ϕ_t^t and those failure detectors can solve consensus.)

Theorem 10 *For $1 \le y \le t$, $\phi_t^y \not\preceq \mathcal{S}$, $\phi_t^y \not\preceq \diamondsuit\mathcal{P}$, $\phi_t^y \not\preceq \diamondsuit\mathcal{S}$, and $\phi_t^y \not\preceq \diamondsuit\mathcal{W}$.*

Proof The impossibility comes from the fact that nothing prevents the sets $suspected_i$ from containing correct processes for an unbounded amount of time. As $\diamondsuit\mathcal{S} < \diamondsuit\mathcal{P}$ and $\diamondsuit\mathcal{W} < \diamondsuit\mathcal{P}$, it is sufficient to prove it for $\diamondsuit\mathcal{P}$, as far as $\diamondsuit\mathcal{S}$, $\diamondsuit\mathcal{W}$ and $\diamondsuit\mathcal{P}$ are concerned. The proof for $\mathcal{S}$ is verbatim the same as the one for $\diamondsuit\mathcal{P}$ (replacing only $\diamondsuit\mathcal{P}$ by $\mathcal{S}$).

The proof consists in assuming (for contradiction) that there is a protocol transforming a failure detector of the class $\diamondsuit\mathcal{P}$ into a failure detector of the class ϕ_t^y. Let us consider a run where an infinite sequence of relevant queries is issued, all of the form QUERY$_y(S)$, for the same S, $t - y < |S| \le t$, and suppose that all processes in S are initially crashed. The answers returned by the protocol define then a sequence consisting of a finite prefix of *false* answers followed by an infinite suffix of *true* answers (by the safety and liveness property of ϕ_t^y). Let τ be a time instant after which all the invocations of QUERY$_y(S)$ return *true*.

However, it could be that no process ever crashes, and no process in S takes a step until after $\tau + \delta$ (where $\delta > 0$ is an arbitrary finite period), with $\diamondsuit\mathcal{P}$ suspecting each process exactly as in the previous fault-prone run from the very beginning until $\tau + \delta$.

As $\diamondsuit\mathcal{P}$ provides each process with the same outputs in both runs until time $\tau + \delta$, it follows that the queries QUERY$_y(S)$ issued between τ and $\tau + \delta$ returns *true* in both runs. This contradicts the safety property of ϕ_t^y in the failure-free run. $\square_{Theorem\ 10}$

9   Conclusion

This paper focused on the combination of two approaches to solve the *k-set agreement* problem, namely failure detectors and conditions. It has proposed novel failure detectors for solving the k-set agreement problem, that, when combined with a condition, establish a new bridge among asynchronous, synchronous and partially synchronous systems with respect to agreement problems.

The paper has presented three main contributions. The first is the new class of failure detectors denoted ϕ_t^y, $0 \le y \le t$. The processes can invoke a primitive QUERY$_y(S)$ with any set S of process ids. Roughly speaking, QUERY$_y(S)$ returns *true* only when all processes in S have crashed, provided $t - y < |S| \le t$. These failure detectors seem interesting in their own right. They have been thoroughly investigated, and compared to the classical failure detectors introduced by Chandra and Toueg.

The second contribution of the paper is a condition-based protocol that solves the k-set agreement problem, with $k = 1 + \max(0, t - (x + y))$, for a condition C of power x and a failure detector of power y, with termination guaranteed for inputs in C. By "power" we mean: C is x-*legal* if and only if it can be used to solve x-fault tolerant

asynchronous consensus, and the failure detector is in the class ϕ_t^y, $0 \leq y \leq t$. Several noteworthy properties and variants of this protocol (that provides a new way to solve asynchronous set agreement and in particular consensus) have been studied.

The third contribution is a corresponding lower bound, showing that there is no ϕ_t^y-based k-set agreement protocol for $(t-d)$-legal conditions with $k \leq \max(0, d-y)$. It follows from this lower bound that there is no condition-based k-set agreement protocol such that $k \leq d$ for any $(t-d)$-legal condition.

Acknowledgments

References

[1] Afek Y., Attiya H., Dolev D., Gafni E., Merritt M. and Shavit N., Atomic Snapshots of Shared Memory. *Journal of the ACM*, 40(4):873-890, 1993.

[2] Anceaume E., Fernandez A., Mostefaoui A., Neiger G. and Raynal M., Necessary and Sufficient Conditions for Transforming Limited Accuracy Failure Detectors. *Journal of Computer and System Sciences*, 68:123-133, 2004.

[3] Attiya H. and Avidor Z., Wait-Free n-Set Consensus when Inputs are Restricted. *Proc. 16th Int. Symposium on Distributed Computing (DISC'02)*, Springer Verlag LNCS #2508, pp. 326-338, 2002.

[4] Attiya H. and Rachman O., Atomic Snapshots in O($n \log n$) Operations. *SIAM Journal of Computing,* 27(2):319-340, 1998.

[5] Attiya H. and Welch J., Distributed Computing, Fundamentals, Simulation and Advanced Topics (Second edition). *Wiley Series on Parallel and Distributed Computing*, 414 pages, 2004.

[6] Borowsky E. and Gafni E., Generalized FLP Impossibility Results for t-Resilient Asynchronous Computations. *Proc. 25th ACM Symposium on the Theory of Computing (STOC'93)*, ACM Press, pp. 91-100, 1993.

[7] Chandra T.D., Hadzilacos V. and Toueg S., The Weakest Failure Detector for Solving Consensus. *Journal of the ACM*, 43(4):685-722, 1996.

[8] Chandra T.D. and Toueg S., Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225-267, 1996.

[9] Chaudhuri S., More *Choices* Allow More *Faults:* Set Consensus Problems in Totally Asynchronous Systems. *Information and Computation,* 105:132-158, 1993.

[10] Chaudhuri S., Herlihy M., Lynch N. and Tuttle M., Tight Bounds for k-Set Agreement. *Journal of the ACM*, 47(5):912-943, 2000.

[11] Chu F., Reducing Ω to $\Diamond \mathcal{W}$. *Information Processing Letters*, 76(6):293-298, 1998.

[12] Delporte-Gallet C., Fauconnier H., Guerraoui R., Hadzilacos V., Kouznetsov P. and Toueg S., The Weakest Failure Detectors to Solve Certain Fundamental Problems in Distributed Computing. *Proc. 23th Int. ACM Symposium on Principles of Distributed Computing (PODC'04)*, ACM Press, pp. 338-346, July 2004.

[13] De Prisco R., Malkhi D. and Reiter M., On k-Set Consensus problems in Asynchronous Systems. *IEEE Transactions on Parallel and Distributed Systems,* 12(1):7-21, 2001.

[14] Dwork C., Lynch N. and Stockmeyer L., Consensus in the Presence of Partial Synchrony. *Journal of the ACM*, 35(2):288-323, 1988.

[15] Fischer M.J. and Lynch N., A Lower Bound for the Time to Assure Interactive Consistency. *Information Processing Letters*, 71:183-186, 1982.

[16] Fischer M.J., Lynch N.A. and Paterson M.S., Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374-382, 1985.

[17] Friedman R., Mostefaoui A., Rajsbaum S. and Raynal M., Distributed Agreement Problems and their Connection with Error-correcting Codes. *IEEE Transactions on Computers*, 56(7):865-875, 2007.

[18] Friedman R., Mostefaoui A., and Raynal M., The Notion of Veto Number for Distributed Agreement Problems. Keynote speech, *Proc. 6th Int. Workshop on Distributed Computing (IWDC'04)*, Springer Verlag LNCS, #3326, pp. 315-325, 2004.

[19] Gafni E., Round-by-round Fault Detectors: Unifying Synchrony and Asynchrony. *Proc. 17th ACM Symposium on Principles of Distributed Computing (PODC'00)*, ACM Press, pp. 143-152, 1998.

[20] Gafni E., Guerraoui R. and Pochon B., From a Static Impossibility to an Adaptive Lower Bound: The Complexity of Early Deciding Set Agreement. *Proc. 37th ACM Symposium on Theory of Computing (STOC'05)*, ACM Press, May 2005.

[21] Guerraoui, R., Indulgent Algorithms. *Proc. 19th Int. ACM Symposium on Principles of Distributed Computing (PODC'00)*, ACM Press, pp. 289-297, July 2000.

[22] Guerraoui R., Non-Blocking Atomic Commit in Asynchronous Systems with Failure Detectors. *Distributed Computing*, 15(1):17-25, 2002.

[23] Guerraoui R. and Raynal M., The Information Structure of Indulgent Consensus. *IEEE Transactions on Computers,* 53(4):453-466, 2004.

[24] Guerraoui R. and Raynal M., The Alpha of Asynchronous Consensus. *The Computer Journal*, 50(1):53-67, 2007.

[25] Guerraoui R. and Schiper A., Gamma-accurate Failure Detectors. *Proc. 10th Workshop on Distributed Algorithms (WDAG'96)*, Springer Verlag LNCS #1151, pp. 269-286, 1996.

[26] Herlihy M.P., Wait-Free Synchronization. *ACM Transactions on programming Languages and Systems*, 11(1):124-149, 1991.

[27] Herlihy M.P. and Penso L. D., Tight Bounds for k-Set Agreement with Limited Scope Accuracy Failure Detectors. *Distributed Computing*, 18(2):157-166, 2005.

[28] Herlihy M.P and Rajsbaum S., Algebraic Spans. *Mathematical Structures in Computer Science*, 10(4):549-573, 2000.

[29] Herlihy M.P. and Shavit N., The Topological Structure of Asynchronous Computability. *Journal of the ACM*, 46(6):858-923, 1999.

[30] Herlihy M.P. and Wing J.L., Linearizability: a Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463-492, 1990.

[31] Hurfin M., Mostefaoui A. and Raynal M., A Versatile Family of Consensus Protocols Based on Chandra and Toueg's Unreliable Failure Detectors. *IEEE Transactions on Computers*, 51(4):395-408, 2002.

[32] Izumi T. and Masuzawa T., Condition Adaptation in Synchronous Consensus. *IEEE Transactions on Computers*, 55(7):843-853, 2006.

[33] Lamport L. and Fischer M., Byzantine Generals and Transaction Commit Protocols. *Unpublished manuscript*, 16 pages, April 1982.

[34] Lo W.-K. and Hadzilacos V., Using failure Detectors to solve Consensus in Asynchronous Shared Memory Systems. *Proc. 8th Int'l Workshop on Distributed Computing (WDAG'94)*, Springer Verlag LNCS #857, pp. 280-295, 1994.

[35] Mostefaoui A., Mourgaya E., and Raynal M., Asynchronous Implementation of Failure Detectors, *Proc. Int. IEEE Conference on Dependable Systems and Networks (DSN'03)*,IEEE Computer Press, pp. 351-360, 2003.

[36] Mostefaoui A., Rajsbaum S. and Raynal M., Conditions on Input Vectors for Consensus Solvability in Asynchronous Distributed Systems. *Journal of the ACM,* 50(6):922-954, 2003.

[37] Mostefaoui A., Rajsbaum S. and Raynal M., Synchronous Condition-Based Consensus. *Distributed Computing*, 18(5):325-343, 2006.

[38] Mostefaoui A., Rajsbaum S., Raynal M. and Travers C., On the Computability Power and the Robustness of Set Agreement-oriented Failure Detector Classes. *Distributed Computing*. To appear: DOI 10.1007/s00446-008-0064-2, 2008.

[39] Mostefaoui A., Rajsbaum S., Raynal M. and Roy M., Condition-Based Protocols for Set Agreement Problems. *Proc. 16th Int. Symposium on Distributed Computing (DISC'02)*, Springer Verlag LNCS #2508, pp. 46-62, 2002.

[40] Mostefaoui A., Rajsbaum S., Raynal M. and Roy M., Condition-based Consensus Solvability: a Hierarchy of Conditions and Efficient Protocols. *Distributed Computing*, 17:1-20, 2004.

[41] Mostefaoui A., Rajsbaum S., Raynal M. and Travers C., From $\Diamond \mathcal{W}$ to Ω: a Simple Bounded Quiescent Reliable Broadcast-based Transformation. *Journal of Parallel and Distributed Computing*, 67(1):125-129, 2007.

[42] Mostéfaoui A. and Raynal M., Solving Consensus Using Chandra and Toueg's Unreliable Failure Detectors: a General Quorum-Based Approach. *Proc. 13th Int. Symposium on Distributed Computing (DISC'99)*, Springer Verlag LNCS #1693, pp. 49-63, 1999.

[43] Mostefaoui A. and Raynal M., k-Set Agreement with Limited Accuracy Failure Detectors. *Proc. 19th Int. ACM Symposium on Principles of Distributed Computing (PODC'00)*, ACM Press, pp. 143-152, July 2000.

[44] Mostefaoui A. and Raynal M., Randomized k-Set Agreement. *Proc. 13th Int. ACM Symposium on Parallel Algorithms and Architectures (SPAA'01)*, ACM Press, pp. 291-297, July 2001.

[45] Mostefaoui A. and Raynal M., Leader-Based Consensus. *Parallel Processing Letters*, 11(1):95-107, 2001.

[46] Neiger G., Failure Detectors and the Wait-Free Hierarchy. *Proc. 14th Int. ACM Symposium on Principles of Distributed Computing (PODC'95)*, ACM Press, pp. 100-109, August 1995.

[47] Raïpin Parvédy Ph., Raynal M. and Travers C., Strongly-terminating Early-Stopping k-set Agreement in Synchronous Systems with General Omission Failures. *Proc. 13th Colloquium on Structural Information and Communication Complexity (SIROCCO'06)*, Springer Verlag LNCS #4056, pp. 182-196, 2006.

[48] Raynal M., Consensus in Synchronous Systems: a Concise Guided Tour. *Proc. 9th IEEE Pacific Rim Int. Symposium on Dependable Computing (PRDC'02)*, Tsukuba (Japan), IEEE Computer Press, pp. 221-228, 2002.

[49] Saks M. and Zaharoglou F., Wait-Free k-Set Agreement is Impossible: The Topology of Public Knowledge. *SIAM Journal on Computing*, 29(5):1449-1483, 2000.

[50] Schiper A., Early Consensus in an Asynchronous System with a Weak Failure Detector. *Distributed Computing*, 10:149-157, 1997.

[51] Yang J., Neiger G. and Gafni E., Structured Derivations of Consensus Algorithms for Failure Detectors. *Proc. 17th Int. ACM Symposium on Principles of Distributed Computing (PODC'98)*, ACM Press, pp. 297-308, July 1998.

[52] Zibin Y., Condition-Based Consensus in Synchronous Systems. *Proc. 17th Int. Symposium on Distributed Computing*, Springer-Verlag LNCS #2848, pp. 239-248, 2003.

A   A $\Diamond \mathcal{P}$-based shared memory consensus protocol

A.1   Description of the Protocol

The protocol uses the following shared registers.

- $D[1..n]$ is an array of write-once single-writer multi-reader atomic registers. $D[i]$, initialized to $\perp$, can be read by all the processes but written only by p_i; p_i sets $D[i]$ to a non-$\perp$ value v to indicate that the value v can be decided.
- $PART[1..n]$ is an array of boolean write-once single-writer multi-reader atomic registers, initialized to $[false, \ldots, false]$. The process p_i sets $PART[i]$ to $true$ to indicate that it participates in the consensus.
- $AC[1..)$ is an unbounded array of adopt-commit-abort objects. Each adopt-commit-abort object is associated with a round.

The $\Diamond\mathcal{P}$-based consensus protocol is described in Figure 6 (this code is for a process p_i, where v_i is the value proposed by that process). First, p_i sets its estimate of the decision value to v_i, registers in the protocol, and sets its local round number r_i to 0 (line 1). The design principle of the protocol relies on the combined use of an eventual common leader (as the Ω-based consensus protocols, e.g., [7, 23, 44]) and adopt-commit-abort objects.

A process p_i executes a sequence of rounds until it decides. Before starting a new round, a process first looks if a value has been decided and decides it if any (lines 3-4). Otherwise, p_i considers as the current leader the process p_ℓ with the smallest identity, among the processes perceived as participating and not currently suspected ($\ell = \min\{j \mid PART_i[j] \wedge j \notin suspected_i\}$ at line 5). If $i = \ell$ (line 6) p_i considers it is the leader and starts the next round (line 7), during which it invokes the adopt-commit-abort object associated with that round, namely $AC[r]$. If this object returns *commit*, p_i decides the value it has been provided with by $AC[r]$. If $AC[r]$ returns *adopt* or *abort*, p_i enters again the repeat loop. Let us notice that, if $AC[r]$ returns *commit* or *adopt*, the local estimate value est_i has possibly been modified. This is a crucial point for the consensus agreement property.

```
Function cons_alg (vi)

(1)  esti ← vi; PART[i] ← true; ri ← 0;
(2)  repeat
(3)     if (∃j : D[j] ≠ ⊥)
(4)        then esti ← D[j]; return (esti);
(5)        else ℓ ← min{j | PART[j] ∧ j ∉ suspectedi});
(6)              if (i = ℓ)
(7)                 then ri ← ri + 1; (di, esti) ← AC[ri].adopt_commit(esti);
(8)                      if di = commit then D[i] ← esti; return (esti) end if
(9)                 end if
(10)    end if
(11) end repeat
```

Figure 6: A $\Diamond\mathcal{P}$-based shared memory consensus protocol

A.2   Proof of the Protocol

Lemma 2 *If a correct process p_i invokes* cons_alg()*, it decides a value.*

Proof Let CP be the set of correct processes that participate in the protocol. Let us assume by contradiction that no process of CP decides (it is easy to see that if one of them decides, the predicate $(\exists j : D[j] \neq \bot)$ is satisfied, and consequently the other correct processes also decide).

Due to the failure model (crash), and the properties of the failure detector class $\Diamond\mathcal{P}$, there is a time τ after which (1) only the processes of CP execute the protocol, and (2) their sets $suspected_i$ contains all the crash processes and only them.

After τ, the array $PART[1..n]$ is no longer modified, and the sets $suspected_i$ are all equal. It follows that there is a single process p_ℓ such that, for any process $p_i \in CP$, we have $\ell = \min\{j \mid PART[j] \wedge j \notin suspected_i\}$. Consequently, there is a finite round r such that only p_ℓ invokes $AC[r]$.adopt_commit(est_i) (lines 6-7). Due to the properties of the $AC[r]$ commit-object, p_ℓ obtains ($commit, v$), and consequently deposits v in $D[\ell]$ and decides. A contradiction with the assumption which proves the lemma. $\Box_{Lemma\ 2}$

Lemma 3 *If a correct process p_i invokes* cons_alg()*, it decides a value.*

Proof It follows from the text of the protocol (line 8 -"direct" decision-, and line 3 -"indirect" decision-) that, to be decided, a value v has first to be returned by an adopt-commit-abort object together with the tag *commit*. So, observing that the processes that invoke the adopt-commit-abort objects, do it in the same sequential order, let us define r as the smallest round number such that $AC[r]$ returns ($commit, v$) to a process p_i. We show that only v can be decided.

Let us first notice that, as p_i obtains ($commit, v$) from $AC[r]$, any other process p_j can obtain only ($commit, v$) or ($adopt, v$) from $AC[r]$ (agreement property of $AC[r]$). If p_j obtains ($commit, v$), it can decide only v. If p_j obtains ($adopt, v$), its local estimate est_j is set to v (line 7).

It follows that all the processes that execute the round $r + 1$ are such that their estimates values are equal to v, which means that only v can be proposed to the object $AC[r + 1]$. It then follows from the obligation property of the adopt-commit-abort object $AC[r + 1]$ that the only pair that can be returned by that object is $(commit, v)$. $\square_{Lemma\ 3}$

Theorem 11 *The protocol described in Figure 6 solves the consensus problem for all the participating processes.*

Proof The proof of the termination and agreement properties of the consensus problem follows from Lemma 2 and Lemma 3, respectively. The proof of the validity property (the decided value is a proposed value) follows directly from the validity property of the adopt-commit-abort objects (they do not create new values). $\square_{Theorem\ 11}$

B   An adopt-commit-abort object implementation

As announced in the paper, this appendix describes an implementation of an adopt-commit-abort protocol. The implementation described in Figure 7 is a merge of the one described in [51] (designed for an asynchronous shared memory system) and the one described in [42] designed for an asynchronous message-passing system). It uses two arrays of one-writer multi-reader atomic registers denoted $PHASE1[1..n]$ and $PHASE2[1..n]$, both initialized to $[\bot, \dots, \bot]$. Then, an entry of such an array contains a pair or remains equal to $\bot$.

```
Function adopt_commit (v_i)

(1)   PHASE1[i] ← v_i;
(2)   set1_i ← {v | PHASE1[j] = v  ∧  v ≠ ⊥  ∧  1 ≤ j ≤ n};
(3)   if (set1_i = {v_i}) then PHASE2[i] ← (single,  v_i)
(4)                       else  PHASE2[i] ← (several, v_i)
(5)   end if;
(6)   set2_i ← {(x,v) | PHASE2[j] ≠ ⊥  ∧  PHASE2[j] = (x,v)  ∧  1 ≤ j ≤ n};
(7)   case set2_i = {(single,v)}                          then return (commit, v)
(8)        set2_i = {(single,v),(several,v'),…}  then return (adopt,   v)
(9)        (single,v) ∉ set2_i                          then return (abort,  v_i)
(10) end case.
```

Figure 7: A shared memory adopt-commit protocol

The behavior of a process p_i can be decomposed into three phases.
- Phase 1 (lines 1-2). A process p_i first deposits its input value v_i in $PHASE1[i]$ to make public the fact that v_i has been proposed to the adopt-commit-abort object. Then, it reads (asynchronously) the whole array $PHASE1[1..n]$ to know if other values have been proposed. The local set $set1_i$ is used to keep these values.

- Phase 2 (lines 3-6). During the second phase, if (from its point of view) no value different from its value v_i has been proposed, p_i sets $PHASE2[i]$ to the pair $(single, v_i)$, otherwise it sets $PHASE2[i]$ to the pair $(several, v_i)$. Then, p_i determines how many pairs (x, v) have been deposited in $PHASE2[1..n]$. (Let us recall that we have $PHASE2[k] = \bot$ until p_k deposits a pair in $PHASE2[k]$.) These non-$\bot$ values (pairs) are collected in the set $set2_i$.

- Phase 3 (lines 7-10). Finally, p_i computes the final value it will return as the result of its invocation.
  - If $set2_i$ contains only the pair $(single, v)$, p_i returns $(commit, v)$: it "commits" the value v.
  - If $set2_i$ contains several pairs, and one of them is $(single, v)$, then p_i "adopts" that value v by returning $(adopt, v)$.
  - Finally, when $set2_i$ does not contain $(single, v)$, p_i has seen no value to be adopted or committed. It consequently "aborts", returning the value v_i it has initially proposed.

The proof of the termination, validity and obligation properties of the adopt-commit-abort object are trivial. A proof of the agreement property for the shared memory model can be found in [51]. A proof for a message-passing model can be found in [42] (that proof assumes a majority of correct processes). That proof consists in showing that,

for any pair of processes p_i and p_j that execute line 6, we have $set2_i = \{(single, v)\} \Rightarrow (single, v) \in set2_j$ (i.e., line 7 and line 9 are "mutually exclusive").