Distrib. Comput. (1999) 12: 209–223

DISTRIBUTED COMPUTING

A simple and fast asynchronous consensus protocol based on a weak failure detector

Michel Hurfin, Michel Raynal

IRISA, Campus de Beaulieu, F-35042 Rennes Cedex, France (e-mail: {hurfin, raynal}@irisa.fr)

Received: August 1997 / Accepted: March 1999

Summary. The Consensus problem is a fundamental paradigm for fault-tolerant asynchronous systems. It abstracts a family of problems known as Agreement (or Coordination) problems. Any solution to consensus can serve as a basic building block for solving such problems (*e.g.*, atomic commitment or atomic broadcast). Solving consensus in an asynchronous system is not a trivial task: it has been proven (1985) by Fischer, Lynch and Paterson that there is no deterministic solution in asynchronous systems which are subject to even a single crash failure. To circumvent this impossibility result, Chandra and Toueg have introduced the concept of unreliable failure detectors (1991), and have studied how these failure detectors can be used to solve consensus in asynchronous systems with crash failures. This paper presents a new consensus protocol that uses a failure detector of the class $\Diamond\mathscr{S}$. Like previous protocols, it is based on the rotating coordinator paradigm and proceeds in asynchronous rounds. Simplicity and efficiency are the main characteristics of this protocol. From a performance point of view, the protocol is particularly efficient when, whether failures occur or not, the underlying failure detector makes no mistake (a common case in practice). From a design point of view, the protocol is based on the combination of three simple mechanisms: a voting mechanism, a small finite state automaton which manages the behavior of each process, and the possibility for a process to change its mind during a round.

Key words: Asynchronous distributed systems – Consensus problem – Crash failures – Fault-tolerance – Unreliable failure detectors

1 Introduction

It is now well recognized that agreement problems are fundamental when designing fault-tolerant distributed systems. Among those, atomic commitment and ordered reliable broadcast (or atomic broadcast) are the most often encountered. In both cases, correct processes have to take a consistent decision. In the first case, processes are data managers and they have to agree in order to commit or to abort effects of a transaction [10]. In the second case, processes have to agree on a single delivery order for a given set of messages [2].

All these practical agreement problems have been abstracted into a basic problem, namely the *Consensus* problem. More precisely, each process is endowed with a value that it proposes to the others, and all correct processes must agree on a common decision value, which has to be one of the proposed values. When the only failures considered are process crashes, this problem has relatively simple solutions in synchronous distributed systems. Unfortunately this is not the case in asynchronous distributed systems. In these systems the most famous result is a negative one. The so-called FLP (Fischer-Lynch-Paterson) result [9] states that it is impossible to design a deterministic algorithm solving the consensus problem in an asynchronous distributed system which is subject to even a single process *crash* failure. Intuitively, this is because (in an asynchronous setting) it is impossible to safely distinguish a very slow process (or a process with which communications are very slow) from a crashed process. This impossibility result has challenged and motivated researchers to find a set of minimal properties that, when satisfied in an asynchronous distributed system, make the consensus problem solvable (with a deterministic algorithm). Minimal synchronism [6], partial synchrony [8] and unreliable failure detectors [4] constitute answers to this challenge.

In this paper, we are interested in solving the consensus problem, in the presence of process crash failures, in an asynchronous model of computation augmented with unreliable failure detectors. Each process is equipped with a failure detector module that provides it with a list of processes it currently suspects to have crashed. A failure detector module can make mistakes by not suspecting a crashed process or by erroneously suspecting a correct one. In a seminal paper [4], Chandra and Toueg have introduced and studied classes of failure detectors. They have defined *Completeness* and *Accuracy* properties of failures detectors. The completeness property is a property on the actual detection of process crashes. The aim of an accuracy property is to restrict the mistakes a failure detector can make. Defining two completeness properties and four accuracy properties, Chandra

and Toueg have proposed eight classes of failure detectors. Among these classes, one of them, denoted $\Diamond\mathcal{W}$ (*Eventually Weak*), is particularly important as it has been proven to be the weakest one that makes the consensus problem solvable [5]. Chandra and Toueg have also shown [4] that, using an appropriate reduction algorithm, these eight failure detectors classes can be reduced to four distinct classes. Actually, using this reduction algorithm, the class $\Diamond\mathcal{W}$ and the class denoted $\Diamond\mathcal{S}$ (*Eventually Strong*) are equivalent.

Several protocols have been proposed which solve the consensus problem when the asynchronous distributed system is augmented with failure detectors of the class $\Diamond\mathcal{S}$ and when a majority of processes are correct. To benefit from the accuracy property of $\Diamond\mathcal{S}$, these protocols are based on the *rotating coordinator* paradigm: they proceed in asynchronous consecutive rounds, each round being managed by a predetermined process (the round coordinator). The first of these protocols (CT) has been proposed by Chandra and Toueg [4]. Another protocol (SC) has been proposed by Schiper [14]. It is important to note that it is the possibility of erroneous suspicions which makes failure detector-based consensus protocols anything but trivial. So, for a given run, the real behavior of a protocol depends on the actual *quality of service* offered by the underlying failure detector.

Among the reasons that motivate the search for time efficient consensus protocols, the following one is particularly important. As previously indicated, consensus is a basic building block that can be intensively used to provide higher level services (such as atomic broadcast or atomic commitment). Consequently, this crucial part of a fault-tolerant service is worth being made as time efficient as possible: this helps the service to offer a good response time to upper layer applications, in all circumstances. Here, finding the "holy grail" would be to design an efficient consensus protocol whose response time would not be affected by process crashes! With a more modest and more realistic goal, in this paper we are interested in designing a consensus protocol that favors a graceful degradation in presence of failures.

An important point that concerns the design of failure detector-based consensus protocols is related to the quality of service offered by the underlying failure detector. As pointed out in [1]: "In many systems, failures are rare, and failure detectors can be tuned to seldom make mistakes (*i.e.*, erroneous suspicions)". So, a crucial issue lies in the design of consensus protocols that are time efficient in runs where the underlying failure detector provides a good quality of service (*i.e.*, when it makes no mistake). This aim has been attained by the SC protocol for *failure-free* runs. In those runs the decision is obtained in 2 communication steps (which seems to be optimal [7, 12]). In this paper we are interested in designing a failure detector-based consensus protocol that is time efficient in runs where the underlying failure detector makes no mistakes, *whether there are failures or not*. In such runs, if the current coordinator has not crashed, the proposed protocol is as good as SC. If the current coordinator has crashed, the protocol requires a single communication step to proceed to the next round (thereby allowing processes to benefit from a new round coordinator). This favors a graceful degradation in presence of process crashes. Efficiency in no erroneous suspicion runs has been obtained by using a decentralized message exchange pattern, and by investigating a very simple idea, namely, allow a process to query the failure detector only during limited periods, and force it to trust the information it obtains.

When designing this protocol, we aimed to produce a protocol which is not only efficient but which also has a relatively simple structure. As noted in [3] "Reliability need not compromise elegance and performance in distributed computing". When solving a problem, we think that the design of a protocol which is *both simple and efficient* can provide a deeper insight into the problem in question. The design principles of the proposed protocol rely on the use of simple mechanisms: a voting mechanism, a small finite state automaton that manages the behavior of each process during a round, and allowing a process to change its mind during a round. This design simplicity is an important issue that should help demystify the consensus problem, and make it attractive as a basic building block for solutions to distributed agreement problems.

The paper is organized as follows. It is composed of five main sections. Section 2 presents the system model, the class $\Diamond\mathcal{S}$ of failure detectors, and the consensus problem. Then, Sect. 3 presents the consensus protocol. Section 4 gives a correctness proof. Section 5 discusses the protocol, analyses its cost and compares it with CT and SC. Finally, Sect. 6 concludes the paper and summarizes its contribution.

2 Asynchronous systems and consensus

This section introduces the system model, failure detectors and the consensus problem. The system model is patterned after the one described in [4, 9, 14]. A formal introduction to failure detectors is provided in [4, 5].

2.1 Asynchronous systems

We consider a system consisting of $n > 1$ processes $\Pi = \{p_1, p_2, \ldots, p_n\}$. A process can fail by *crashing*, *i.e.*, by prematurely halting; it behaves correctly (*i.e.*, according to its specification) until it (possibly) crashes. By definition, a *correct* process is a process that does not crash, otherwise it is *faulty*. Let f denote the maximum number of processes that can crash. Processes communicate and synchronize by sending and receiving messages through channels. Every pair of processes is connected by a *reliable* channel, *i.e.*, a message sent by a process p_i to a process p_j is eventually received by p_j, if p_j is correct. It is the multiplicity of processes and the communication by message-passing that make the system *distributed*.

There is no assumption about the relative speed of processes or the message transfer delays. This absence of timing assumptions makes the distributed system *asynchronous*.

2.2 Failure detectors

Informally, a failure detector consists of a set of modules, each one attached to a process: the module attached to p_i

maintains a set (named $suspected_i$) of processes it currently suspects to have crashed. Any failure detector module is inherently unreliable: it can make mistakes by not suspecting a crashed process or by erroneously suspecting a correct one. Moreover, suspicions are not necessarily stable: a process p_j can be added to and removed from a set $suspected_i$ according to whether p_i's failure detector module currently suspects p_j or not. As in [4], we say "process p_i suspects process p_j" at some time t, if at time t we have $p_j \in suspected_i$.

As indicated in the Introduction, failure detector classes have been defined by Chandra and Toueg [4] in terms of two abstract properties, namely *Completeness* and *Accuracy*. We are interested here in the class of failure detectors, called $\Diamond\mathscr{S}$ (*Eventually Strong*), characterized by the two following properties:

- Strong Completeness: Eventually, every crashed process is permanently suspected by every correct process.
- Eventual Weak Accuracy: There is a time after which some correct process is never suspected by any correct process.

It is important to note that failure detectors of the class $\Diamond\mathscr{S}$ can make an arbitrary number of mistakes (*e.g.*, a correct process can loop suspecting and not suspecting a correct process). Moreover, it is also interesting to notice that the Eventual Weak Accuracy property suggests to base protocols using a failure detector of the class $\Diamond\mathscr{S}$, on the rotating coordinator paradigm: in that case, there is a time after which, a correct coordinator will not be suspected by any correct process.

2.3 The consensus problem

The problem. In this problem, every correct process p_i *proposes* a value v_i and all correct processes have to *decide* on some value v, in relation to the set of proposed values. More precisely, the *Consensus problem* is defined by the three following properties [4, 9]:

- Termination: Every correct process eventually decides on some value.
- Validity: If a process decides v, then v was proposed by some process.
- Agreement: No two correct processes decide differently.

The agreement property applies only to correct processes. So, it is possible that a process decides on a distinct value just before crashing. *Uniform Consensus* prevents such a possibility. It has the same Termination and Validity properties plus the following agreement property:

- Uniform Agreement: No two processes (correct or not) decide differently.

In the following we are interested in the uniform consensus problem.

Solving consensus with $\Diamond\mathscr{S}$. Two important results are attached to protocols that solve the consensus problem, using failure detectors of the class $\Diamond\mathscr{S}$:

- Chandra and Toueg have shown that any such protocol requires at least a majority of processes to be correct (*i.e.*, $f < n/2$) for the problem to be solvable [4].
- Guerraoui has shown that any protocol solving the consensus problem using such a failure detector, also solves the uniform consensus problem [11].

3 The consensus protocol

3.1 Underlying principles

3.1.1 Brief survey of related works

The first consensus protocol designed to work with a failure detector belonging the class $\Diamond\mathscr{S}$ was proposed by Chandra and Toueg [4]. Other protocols have since been proposed (*e.g.*, [1, 13, 14]). All these protocols share the following design principles:

- Each protocol is based on the *rotating coordinator* paradigm and proceeds in consecutive asynchronous rounds. Each round is coordinated by a process. The coordinator of round r, is a predetermined process, namely, p_c with $c = (r \bmod n) + 1$. As in [4], when considering a round r, the associated process p_c will be called the "current" coordinator. The rotating coordinator paradigm is used in the following way.
  During a round r, the coordinator tries to impose a value as the decision value. To this end processes have to cooperate. Different (centralized/decentralized) protocols can be designed, depending on how the processes cooperate. Crashes can be dealt with by moving to the next coordinator. It is possible that not all processes decide in the same round, depending on the pattern of failures and on the pattern of failure suspicions that occur during a given execution. One important point which differentiates between the protocols is the way they solve this issue, while ensuring there is a single decision value (*i.e.*, the Agreement property).
- In every protocol, each process p_i manages a local variable est_i that represents its current estimate of the decision value (initially, est_i is the initial value v_i proposed by p_i). This value is updated as the protocol progresses and converges to the decision value.

Chandra-Toueg's protocol and Schiper's protocol are now briefly described. (These protocols will be compared with the proposed protocol in Sect. 5.3).

Chandra-Toueg's protocol. In this protocol [4], during a round, the cooperation between processes to establish a decision value is centralized. More precisely, each round is composed of four phases. In round r, the current coordinator p_c tries to establish a consensus value in the following way:

- During phase 1, each process p_i sends its estimate est_i to the current coordinator p_c.
- During phase 2, p_c gathers values from a majority of processes, defines a new estimate from the values it received and broadcasts this new estimate to all processes.

– During phase 3, each process p_i waits for the receipt of an estimate value from p_c. If it receives one, it sends back a positive acknowledgment to p_c. If it suspects p_c to have crashed, it sends back a negative acknowledgment to p_c.
– During phase 4, the coordinator p_c waits for a majority of acknowledgments. If it receives a majority of positive acknowledgments, it informs all the processes that the value it has previously disseminated has to be considered as the decision value.

It is important to note that Chandra-Toueg's protocol presents the following property: when a majority of processes in a round have adopted (by sending a positive acknowledgment) an estimate value proposed by the round coordinator, this value becomes "locked" in the sense that no other decision value is possible. This property is essential: it ensures Agreement will be satisfied despite the fact that not all processes decide in the same round.

Schiper's protocol. While Chandra-Toueg's protocol is based on a *Master-Slave* scheme[1], no such "asymmetry" exists in Schiper's protocol [14]. In this protocol, the cooperation between processes to establish a decision value during a round is decentralized. More precisely, during a round r, the process p_c that is the current coordinator initiates this round by broadcasting its own estimate est_c. Then all processes behave similarly.

– As indicated, when a round starts, the current coordinator p_c broadcasts its estimate est_c to all the processes. A process p_i that receives est_c, forwards it to all processes. If a process receives est_c from a majority of processes, it decides on this value.
– At any time during a round r, a process p_i may suspect that the current coordinator p_c has crashed. In this case, p_i broadcasts a SUSPICION message. As soon as a process has received a majority of SUSPICION messages, it can no longer decide during this round. It then executes an "estimate-locking" protocol whose aim is to ensure that processes starting the next round will start with "consistent" estimate values[2]. This "estimate-locking" protocol requires an additional exchange of messages.

During a round, Chandra-Toueg's protocol uses a centralized scheme (all messages are to/from the coordinator), while Schiper's protocol uses a decentralized scheme (after the coordinator has sent its estimate, each process sends messages to all other processes). Actually, these two schemes have been investigated by Skeen [15] to solve a particular agreement problem (namely, the non-blocking atomic commitment problem) in the context of distributed systems equipped with reliable failure detectors. More precisely, if we consider that there are neither failures, nor false suspicions, the decision is obtained in the first round, and, from a syntactic point of view, the message exchange pattern generated by Chandra-Toueg's protocol is similar to the one generated by Skeen's centralized commit protocol [15]. In the same context (no failure, no suspicion), the message exchange pattern generated by Schiper's protocol is syntactically similar to the one generated by Skeen's decentralized commit protocol. What makes Chandra-Toueg's and Schiper's protocols in no way trivial lies in the fact that they do not require reliable failure detectors.

[1] During each round r, the coordinator of r acts a master role.

[2] "Consistent" means that if a process has decided est_c during round r, then any process p_i proceeding to round $r+1$, will start with $est_i = est_c$. Using a method different from that used in Chandra-Toueg's protocol, this ensures that est_c is "locked". See Sect. 5.3.

3.1.2 Underlying principles of the proposed protocol

As indicated in the Introduction, when designing this protocol, our aim was to produce a protocol which is both simple and efficient whenever the underlying failure detector provides a good quality of service (*i.e.*, when it makes no mistakes, whether failures occur or not). From a structural point of view, the protocol is based on a simple combination of well-known mechanisms: asynchronous rounds, voting, finite state automaton and allowing a process to change its mind. From a behavioral point of view, the proposed protocol is based on a decentralized message exchange pattern (similar to [14, 15]): during a round, the cooperation between processes to establish a decision value is decentralized. Assuming that the underlying failure detector makes no mistakes, let us consider a round r.

– The decision is obtained in two communication steps if the current coordinator p_c has not crashed[3]. More generally, whatever is the quality of service offered by the failure detector, as long as the current coordinator is not suspected, the protocol tries to impose its estimate value as the decision value.
– Progress to the next round (to benefit from another coordinator) takes only one communication step if the current coordinator has crashed. More generally, as soon as the current coordinator is suspected, the protocol tries to make all correct processes progress, as quickly as possible, to the next round.

These two items show the guideline that has governed the design of the protocol: processes trust the failure detector a priori. This intuitively explains why the protocol is efficient when the failure detector is reliable.

To attain this goal, at round r, each process p_i has to vote. It votes either to proceed to the next round (NEXT vote) or to decide during the current round (CURRENT vote). Moreover, a CURRENT/NEXT vote carries the current estimate (of the decision value) of the process issuing the vote. Voting is implemented by messages. In order to prevent blocking (see below) a process may vote CURRENT and later change its mind and vote NEXT. On the other hand, in each round, a NEXT vote is definitive. A process p_i decides at the current round as soon as it has received a majority of CURRENT votes. It proceeds to the next round if it receives a majority of NEXT votes. If a process p_i decides in round r and another process p_j decides in round $r' > r$, the protocol ensures that p_j starts round $r+1$ with the value decided by p_i. Moreover (and independently of the fact that a value has been decided

[3] In this ideal scenario (no failures, no erroneous suspicions), Schiper's protocol also decides in two communication steps.

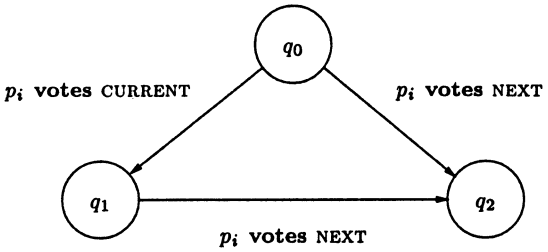

Fig. 1. Finite state automaton of a process p_i

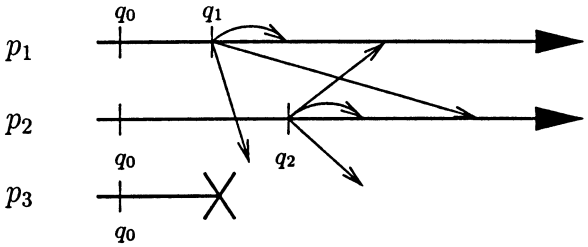

Fig. 2. A deadlock situation

by some processes in round r), it is important to note that the progression of processes to round $r + 1$ is synchronized by the exchange of NEXT votes.

During each round, the behavior of each process p_i is determined by a finite state automaton (as in [15]). This automaton, illustrated in Fig. 1, is composed of 3 states: an initial state q_0, and two other states q_1 and q_2. The local variable $state_i$ will denote the automaton state in which p_i currently is. During a round, the current state of a process p_i is directly related to the votes it issues and not to the ones it has received. When p_i decides, it can be in any state of the automaton; but, when it proceeds to the next round, p_i is in state q_2. During a round, the state of the automaton has the following meaning:

- $state_i = q_0$: p_i has not yet voted.
- $state_i = q_1$: p_i has voted CURRENT and has not changed its mind.
- $state_i = q_2$: p_i has voted NEXT.

The protocol manages the progression of each process p_i within its automaton, according to the following rules. At the beginning of round r, $state_i = q_0$. Then, during r, the transitions are:

- *Transition* $q_0 \rightarrow q_1$ (p_i *first votes* CURRENT). This transition occurs when p_i, while in the initial state q_0, receives a CURRENT vote. This means that p_i has not previously suspected the round coordinator. Moreover, when p_i moves to q_1, it broadcasts a CURRENT vote.
- *Transition* $q_0 \rightarrow q_2$ (p_i *first votes* NEXT). This transition occurs when p_i, while in the initial state q_0, suspects the current coordinator. This means that p_i has not previously received a CURRENT vote. Moreover, when p_i moves to q_2, it broadcasts a NEXT vote.
- *Transition* $q_1 \rightarrow q_2$ (p_i *changes its mind*). This transition is used to prevent a possible deadlock. Let us consider Fig. 2 where $n = 3$. Processes p_1, p_2 and p_3 have entered round r (coordinated by p_1), and p_3 has crashed just after entering this round. Let us examine the following scenario:
  - Process p_1 has sent a CURRENT vote, and has moved to state q_1.
  - While it was in state q_0, process p_2 has suspected the current coordinator p_1. Consequently, it has issued a NEXT vote and has moved to state q_2.

  Both p_1 and p_2 have received votes from a majority of processes, but they have received neither a majority of CURRENT votes (so they cannot decide), nor a majority of NEXT votes (so they cannot progress to the next round). Moreover, as p_3 has crashed, it will not vote. This is a typical deadlock situation. To prevent such a deadlock, a process that has issued a CURRENT vote is allowed to change is mind; in the example, p_1 is authorized to issue a NEXT vote. After this vote has been received, each correct process (p_1 and p_2) has a majority of NEXT votes and consequently may proceed to the next round.

Summary. To summarize, the principles underlying the protocol are:

1. The use of the *rotating coordinator* paradigm. The protocol proceeds in consecutive asynchronous rounds, each round being coordinated by a predetermined process. Combined with the properties of $\Diamond\mathscr{S}$, this ensures that there will eventually be a round in which the coordinator will not be suspected.
2. The use of the *voting* paradigm. Used at each round, it consists of:
   - Two votes.
     A CURRENT (resp. NEXT) vote means that the issuing process is in favor of deciding in the current round (resp. proceeding to the next round).
   - A majority rule.
     A process decides at the current round (resp. progresses to the next round) as soon as it has received a majority of CURRENT (resp. NEXT) votes.
   - A simple automaton.
     This automaton defines the state of a process with respect to the votes it has issued.
   - The possibility for a process to change its mind.
     A process that voted CURRENT can later issue a NEXT vote. This prevents processes from blocking forever.

3.2 Description of the protocol

The consensus protocol is described in Fig. 3. To propose its value v_i, each correct process p_i calls the function consensus(v_i). It decides when it executes the statement **return**(v) at line 3 or at line 14 (where the decided value is v).

Local variables. In addition to the local variables previously introduced (est_i and $state_i$), process p_i manages the following four local variables:

- r_i defines the current round number.

– $nb\_current_i$ (resp. nb_next_i) counts the number of CURRENT (resp. NEXT) votes received by p_i during the current round.
– rec_from_i is a set composed of the process identities from which p_i has received a (CURRENT or NEXT) vote during the current round.

Finally, $suspected_i$ is a set managed by the associated failure detector module (cf. Sect. 2.2); p_i can only read this set.

Let us remark that, except for r_i, the domain of every variable is bounded.

View of a round by a process. If we consider a process p_i at round r, p_i has the following view of the global state concerning round r:

– $state_i$ defines its current state, with respect to r.
– $nb\_current_i$, nb_next_i and rec_from_i describe its perception of the whole set of processes, with respect to their progress within their automaton during round r. More precisely:
  – $nb\_current_i$ is the number of processes that p_i perceives as having moved from q_0 to q_1 in round r (p_i has received a CURRENT vote from them). It is possible that some of those processes are now in q_2.
  – nb_next_i is the number of processes p_i perceives as being presently in state q_2 in round r (p_i has received a NEXT vote from them).

Protocol description. Function consensus() adopts the structure used in [4, 14], namely, it consists of two concurrent tasks. The first task handles the reception of a DECIDE message (lines 2-3); it ensures that if a process p_i decides (line 3 or line 14), then all correct processes will also receive a DECIDE message. The second task (lines 4-27) describes a round: it consists of a loop that constitutes the core of the protocol.

Messages. In addition to other values, each (CURRENT or NEXT) vote carries the identity of its sender and its round number[4]. The notation "**send** MSG() **to** X", where $X \subseteq \Pi$ means "$\forall p_j \in X$ **do send** MSG() **to** p_j **enddo**".

A CURRENT vote carries the estimate value of the current round coordinator. A NEXT vote carries the estimate value of its sender and a boolean flag indicating whether the sender moved to q_2 directly (transition: $q_0 \rightarrow q_2$) or indirectly (transitions: $q_0 \rightarrow q_1 \rightarrow q_2$).
- In the first case, the flag value is $suspicion$ (this occurs at line 16 or at line 25). The estimate value carried by the message is then not necessarily equal to est_c.
- In the second case, the flag value is $deadlock\_prevention$ (this occurs at line 22 or at line 26). The estimate value of the sender is then necessarily equal to est_c.

[4] Notation. At line 9 and at line 18, when a (CURRENT or NEXT) vote is received, the second message field is underlined. This notation means that this field of the received vote must contain a value equal to the value of the corresponding local variable, namely r_i, for the message to be received. In other words, in any round r, only votes related to round r can be received.

Protocol statements. During a round r, a process executes the following actions:

– At the beginning of r, each process initializes its local variables (line 5). Moreover, the current coordinator p_c proposes its estimate est_c to become the decision value by broadcasting a CURRENT vote carrying this value, lines 6 and 12.
  The sending of a message by the current coordinator p_c to itself at line 6, is a "fictional" sending, used only to get a description of the **while** loop (as far as possible) independent of process identities. Process p_c instantaneously receives this message and executes lines 9-14. In the **while** loop the identity of p_c appears only at line 15 and concerns the suspicion of p_c.
– Each time a process p_i receives a (CURRENT or NEXT) vote, it updates the corresponding counter and the set rec_from_i (lines 11 and 19). When, in the **while** loop, process p_i broadcasts a vote, it does not send it to itself but simulates its reception by updating these control variables (lines 13, 17 and 23).
  Note that only nb_next_i is updated at line 23. Updating rec_from_i is needless because, in this case, p_i's identity is already in the set rec_from_i (as, when p_i moved to state q_1, it executed line 13).
– When a process receives a CURRENT vote for the first time, namely, CURRENT(p_k,r,est_k), it adopts est_k as its current estimate (line 10). If, in addition, it is in state q_0, it moves to state q_1, and also votes CURRENT to push the decision on this estimate during the current round (line 12).
– A process p_i decides on the estimate proposed by the current coordinator as soon as it has received a majority of CURRENT votes, *i.e.*, a majority of votes that agree to conclude on this value during the current round (line 14).
– A process p_i takes into account the fact that the current coordinator p_c is suspected only when $state_i = q_0$ (lines 15-17). If it suspects p_c, then p_i votes to proceed to the next round (by broadcasting a NEXT vote) and updates $state_i$ to q_2 accordingly (see Fig. 1).
– When p_i receives a NEXT($p_k, \underline{r_i}, est_k, flag_k$) vote (*i.e.*, a vote to proceed to the next round, line 18), it updates its control variables (line 19). Moreover, if p_i has not yet received a CURRENT vote carrying the estimate of p_c (*i.e.*, if $nb\_current_i = 0$), and if p_k can provide it with this estimate (*i.e.*, if $flag_k = deadlock\_prevention$), then p_i adopts it (line 20).
– The set of statements at lines 21-23 aims to prevent the blocking of processes (see the discussion at the end of Sect. 3.1, just before the "Summary" paragraph). Processes can block in the current round if not enough CURRENT votes have been sent (*i.e.*, $nb\_current_i < n/2$), and not enough NEXT votes have been sent (*i.e.*, $nb\_next_i < n/2$). If p_i executes line 21, it has not yet decided. So, if (1) p_i has only voted CURRENT (*i.e.*, it was for deciding at the current round, and $state_i = q_1$), if (2) additionally p_i has received a (CURRENT or NEXT) vote from a majority of processes (*i.e.*, $\mid rec\_from_i \mid > n/2$), and if also (3), from its perception of the current round, p_i will not receive any new information (*i.e.*, $\forall k : p_k \in rec\_from_i \cup suspected_i$), then p_i changes its

```
function consensus(v_i)

(1) r_i ← 0; est_i ← v_i;
    cobegin
(2)   || upon reception of DECIDE(p_k, r_k, est_k)
(3)      send DECIDE(p_i, r_k, est_k) to Π − {p_i, p_k}; return(est_k)

(4)   || loop % on a sequence of asynchronous rounds %
(5)      c ← (r_i mod n) + 1; r_i ← r_i + 1; state_i ← q_0; rec_from_i ← ∅; nb_next_i ← 0;
(6)      if (i = c) then send CURRENT(p_i, r_i, est_i) to itself; nb_current_i ← −1
(7)                 else nb_current_i ← 0 endif;

(8)      while (nb_next_i ≤ n/2) do % wait until a branch can be selected, and then execute it %
(9)        upon reception of CURRENT(p_k, r_i, est_k)
(10)            if (nb_current_i = 0) then est_i ← est_k endif;
(11)            nb_current_i ← nb_current_i + 1; rec_from_i ← rec_from_i ∪ {p_k};
(12)            if (state_i = q_0) then state_i ← q_1; send CURRENT(p_i, r_i, est_i) to Π − {p_i};
(13)                                   nb_current_i ← nb_current_i + 1; rec_from_i ← rec_from_i ∪ {p_i} endif;
(14)            if (nb_current_i > n/2) then send DECIDE(p_i, r_i, est_i) to Π − {p_i}; return(est_i) endif

(15)       upon (p_c ∈ suspected_i)
(16)            if (state_i = q_0) then state_i ← q_2; send NEXT(p_i, r_i, est_i, suspicion) to Π − {p_i};
(17)                                   nb_next_i ← nb_next_i + 1; rec_from_i ← rec_from_i ∪ {p_i} endif

(18)       upon reception of NEXT(p_k, r_i, est_k, flag_k)
(19)            nb_next_i ← nb_next_i + 1; rec_from_i ← rec_from_i ∪ {p_k};
(20)            if ((nb_current_i = 0) ∧ (flag_k = deadlock_prevention)) then est_i ← est_k endif

(21)       upon ((state_i = q_1) ∧ (| rec_from_i |> n/2) ∧ (∀p_k: p_k ∈ rec_from_i ∪ suspected_i))
(22)            state_i ← q_2; send NEXT(p_i, r_i, est_i, deadlock_prevention) to Π − {p_i};
(23)            nb_next_i ← nb_next_i + 1; % the variable rec_from_i has already been updated %
(24)     endwhile;

(25)     if (state_i = q_0) then state_i ← q_2; send NEXT(p_i, r_i, est_i, suspicion) to Π − {p_i} endif;
(26)     if (state_i = q_1) then state_i ← q_2; send NEXT(p_i, r_i, est_i, deadlock_prevention) to Π − {p_i} endif
(27)   endloop
    coend
```

Fig. 3. Fast consensus protocol based on $\Diamond\mathscr{S}$

mind: it broadcasts a NEXT vote to favor the transition to the next round (line 22) and, accordingly, it moves to state q_2. So, this NEXT vote carries a flag whose value is $deadlock\_prevention$, and consequently, the estimate value it carries is equal to est_c. As we will see in the proof (assuming a majority of correct processes), this strategy will ensure that if a correct process p_i can not decide at a given round, then it will eventually have $nb\_next_i > n/2$, and consequently, it will progress to the next round (line 4).

– Finally, the aim of lines 25-26 is to ensure that, when a process progresses from round r to round $r + 1$, it has issued a NEXT vote during round r. These NEXT votes are used to prevent other processes from remaining blocked in round r.

Remark. Message processing order. Note that, if several choices are possible when a process p_i is in the **while** loop (lines 8-24), choosing to process CURRENT votes first can allow p_i to decide sooner.

4 Correctness proof

This section proves that the previous protocol satisfies the Termination, Validity and Uniform Agreement properties stated in Sect. 2.3. This proof assumes that:

– **H1**: There is a majority of processes that are correct (*i.e.*, $f < n/2$).
– **H2**: The underlying failure detector belongs to the class $\Diamond\mathscr{S}$, *i.e.*, it satisfies:
H2.1: Strong Completeness (Eventually, every crashed process is permanently suspected by every correct process). And,
H2.2: Eventual Weak Accuracy (There is a time after which some correct process is never suspected by any correct process).
– **H3**: Communication channels are reliable.

4.1 Validity

Theorem 1 *If a process p_i decides v, then v was proposed by some process.*

Proof. We show that the value v returned by the statement **return**(v) is a value proposed by some process. The only

lines at which **return** is executed are the lines 3 and 14. In both cases, the returned value is an estimate (est_k or est_i). For any process p_i, we have:

- Initially: $est_i = v_i$ (line 1).
- Then: est_i can be modified at line 10 or at line 20. Due to assumption H3 (no message alteration, no spurious message), in both cases the new value of est_i is the value of another estimate.

It follows, by induction, that est_i is a value proposed by some process. $\square_{Theorem\ 1}$

4.2 Termination

Two preliminary Lemmas are first proved. The first lemma shows that no blocking can occur at a given round. Then, the theorem will show that the number of rounds is finite.

Lemma 1 *If no process decides during any round $r' \leq r$, then all correct processes start round r + 1.*

Proof. The proof is by contradiction. Suppose no process has decided in any round $r' \leq r$, where r is the smallest round number in which some correct process blocks forever in the **while** loop (lines 8-24). Let us note that no correct process has received a DECIDE message (otherwise, it would execute lines 2-3 and decide).

1. Note first that any process starting round r does so in state q_0 (line 5). Firstly, it is shown that a correct process p_i can not remain in state q_0 during round r. This follows from:
   - (i) Either p_i suspects p_c and moves to q_2 (lines 15-17). This is due either to a mistake of the underlying failure detector, or to a crash of p_c.
   - (ii) Or p_i never suspects p_c. Due to assumption H2.1 (Strong Completeness), this means that p_c is correct: eventually it broadcasts a CURRENT vote. Due to assumption H3, p_i receives at least one CURRENT vote (from p_c or from another process) and moves to q_1 (line 12).
2. Next, it is shown, that a correct process can not remain in state q_1.
   From the previous point, any correct process either sends a CURRENT vote and moves to q_1 (Case ii), or sends a NEXT vote and moves to q_2 (Case i). As there is a majority of correct processes, it follows that for any correct process p_i, we will have $|rec\_from_i| > n/2$. Moreover, during round r, for any (correct or not) process p_k, any correct process p_i either eventually receives a CURRENT/NEXT vote from p_k, or (due to assumption H2.1: Strong Completeness) eventually suspects p_k (by definition, if p_k is not correct, it eventually crashes). Hence, the condition stated at line 21 eventually becomes true: p_i issues a NEXT vote and moves to q_2.
3. It follows from the two previous points that all correct processes move to q_2 and send a NEXT vote. Consequently, any correct process p_i receives a majority of NEXT votes, and as $nb\_next_i > n/2$, proceed to round r + 1. A contradiction.

$\square_{Lemma\ 1}$

Lemma 2 *During a round r, if no process moves from q_0 directly to q_2 within the* **while** *loop, then no process sends a* NEXT *vote (or equivalently, no process moves to q_2).*

Proof. Note first that any process starts round r in state q_0 (line 5). Moreover, during any round r, a process can receive several NEXT votes but can send at most one. Let a *r-prime* process be a process that, during round r, has sent a NEXT vote before receiving a NEXT vote. Suppose that, during round r, a process sends a NEXT vote. At least one process p_i is *r-prime*. We will show there is a contradiction.

Following the Lemma assumption, no process executes line 16. Consequently, p_i sends the NEXT vote at line 22 (Case 1) or at line 25-26 (Case 2).

- *Case 1.* Process p_i has executed line 22.
  To change its mind, p_i had $| rec\_from_i | > n/2$, *i.e.*, it had received a (CURRENT or NEXT) vote from a majority of processes (line 21).
  - If all these votes are CURRENT votes, then p_i has decided at line 14 when it received the last vote making true the condition $nb\_current_i > n/2$. Consequently, as p_i has executed a **return** statement it will never execute line 22. This contradicts the Case 1 assumption.
  - If one of these votes is of type NEXT, then some process p_k sent a NEXT vote that has been received by p_i. This contradicts the fact that p_i is *r-prime*.
- *Case 2.* Process p_i has executed line 25 or 26.
  To send a NEXT vote at line 25 or 26, p_i has terminated its **while** loop. So, it has $nb\_next_i > n/2$. It follows that p_i has received NEXT votes. This contradicts the fact that p_i is *r-prime*.

$\square_{Lemma\ 2}$

Theorem 2 *Every correct process eventually decides some value.*

Proof. Let us consider the two following cases.

1. A process p_j decides. It does so at line 3 or at line 14. As shown by these lines, p_j has broadcast a DECIDE message before deciding. As communication is reliable (H3), any correct process p_i will receive this DECIDE message, and will decide accordingly.
2. No process decides. We will show there is a contradiction. In that case, there is a time t after which: (1) there are no more crashes[5], and (due to assumption H2.2, Eventual Weak Accuracy) (2) there is a correct process that is no longer suspected (let p_j be this process). Let r be the first round that occurs after t and which is coordinated by p_j (due to Lemma 1, such a round does exist since no process decides).
   During round r, let us examine the set of processes after they entered the **while** loop (lines 8-24). Note that, due to the assumption on r, from now on, all active processes are correct.
   - The coordinator p_j sends a CURRENT vote to all processes (lines 6 and 9-12).
   - As, by assumption, the current coordinator p_j is not

[5] So, after t we have only to consider correct processes.

suspected, no process p_i executes lines 15-17. Consequently, no process executes line 16. More precisely, no process moves directly from q_0 to q_2 within the **while** loop.
- As no process moves directly from q_0 to q_2 within the **while** loop, from Lemma 2, we conclude that no process sends a NEXT vote, *i.e.*, no process moves to q_2.
- As each process p_i enters the **while** loop and does not move to q_2 within this loop, when it receives a CURRENT vote for the first time, p_i is necessarily in state q_0. According to lines 9 and line 12, it moves to q_1 and also votes CURRENT (sending this vote to all processes).
- As there is a majority of correct processes (H1), and as communication is reliable (H3), it follows that each process p_i will receive a majority of CURRENT votes. As no process sends a NEXT vote, it follows that any correct process p_i will necessarily decide at line 14. This contradicts the assumption that no process decides.

$\square_{Theorem\ 2}$

4.3 Uniform agreement

First, three preliminary lemmas are proved. Then, they are used in the proof of the Uniform Agreement theorem.

Lemma 3 *Let us consider a process p_i. During any round r, we have $(nb\_current_i \neq 0) \Rightarrow (est_i = est_c)$, where p_c is the coordinator of round r.*

Proof. Case $i = c$. First, let us note that est_c is not updated. When p_c executes line 10, due to line 6, $nb\_current_c = -1$. Then, $nb\_current_c$ is increased to 1 (lines 11 and 13). So, if it is executed, line 20 does not update est_c (because $nb\_current_c > 0$).

Case $i \neq c$. The estimate est_i can be updated during round r at line 10 or at line 20. During round r, due to the fact that (1) $nb\_current_i$ ($i \neq c$) is initialized to 0 (line 7), (2) $nb\_current_i$ is now different from 0 (Lemma assumption), (3) $nb\_current_i$ can only be increased at line 11, it follows that, the last update of est_i has necessarily occurred at line 10 (if any, the previous ones have been done at line 20). So, est_i has been updated at line 10 with est_k, when p_i received a vote CURRENT(p_k,r,est_k) for the first time (see the test at line 10).

1. Case $p_k = p_c$. The Lemma trivially follows.
2. Case $p_k \neq p_c$.
   When examining the sequence of CURRENT messages that entail the update of est_i, we can make the following observations:
   - During a given round r, the update of the estimate at line 10 and the sending of a CURRENT vote (at line 12) are executed at most once by a process.
   - During a given round r, any process p_j (distinct from p_c) that issues a CURRENT(p_j, r, est_j) vote (at line 12), has previously received a CURRENT($-, r, est$) vote (at line 9). Moreover, due to the update of est_j done at line 10, $est_j = est$.
   - A single process, namely p_c, can initiate such a sequence of CURRENT votes.

   It follows from these observations that the sequence of CURRENT votes entailing the update of est_i (1) is finite, (2) is initiated by p_c, and (3) piggybacks the value of est_c. Hence, $est_i = est_k = \cdots = est_j = \cdots = est_c$.

$\square_{Lemma\ 3}$

Lemma 4 *Any process p_i that sends a* DECIDE *message labeled with the round number r, decides value est_c (where p_c is the coordinator of round r).*

Proof. Let us consider the two possible cases.

1. Process p_i decides during round r at line 14: so, it has $nb\_current_i \neq 0$. From Lemma 3, it decides est_c. Moreover, let us observe that all processes that decide at line 14 during round r, sent DECIDE messages carrying the same value, namely, est_c.
2. If a process p_i decides at line 3, it decides on the value carried by a DECIDE message sent with the round label r. All those DECIDE messages carry est_c. This follows from the fact that any DECIDE message labeled with r:
   - Either is (initially) sent at line 14: due to the previous observation, it carries est_c.
   - Or, is sent at line 3: in that case, it only forwards a value sent initially. (Due to asynchrony, it is possible that this value, launched from a process deciding during round r at line 14, went through several processes, and has therefore been carried by several "consecutive" DECIDE messages before arriving at p_i).

$\square_{Lemma\ 4}$

Lemma 5 *If process p_i decides v and sends a* DECIDE *message labeled with the round number r, then all processes p_j that start round $r + 1$ do so with $est_j = v$.*

Proof. Let us consider round r (coordinated by p_c). We first establish a relation (R7) that is then used to prove the lemma by contradiction.

1. As p_i sends a DECIDE message labeled with the round number r, some process p_k (possibly $p_k = p_i$) has executed line 14 during round r, consequently, $nb\_current_k > n/2$ (R1). Furthermore, by Lemma 4, the decided value v is equal to est_c.
2. Let us consider the three following sets of processes (related to round r):
   – X_1^r = { processes that moved from q_0 to q_1 and did not move to q_2 }
   – X_2^r = { processes that moved from q_0 directly to q_2 }
   – X_3^r = { processes that moved from q_0 to q_1 and then to q_2 }
   Note that these sets are disjoint. They include processes that have possibly crashed after moving from q_0 to another state[6]. Moreover, we have $|X_1^r| + |X_2^r| + |X_3^r| \leq n$ (R2).
3. Let $sent\_to\_pk$ be the number of processes that sent a CURRENT vote to p_k (at line 12). As the number of CURRENT votes sent to a p_k is greater or equal to the number of CURRENT votes received by p_k, we have $sent\_to\_pk \geq nb\_current_k$ (R3).

[6] They may have crashed during the execution of the send statement associated with a state transition.

4. All processes belonging to X_3^r have sent a CURRENT vote to p_k at line 12 (when they moved from q_0 to q_1). Moreover, all processes belonging to X_1^r and which executed all of line 12 have sent a CURRENT vote to p_k. Some processes belonging to X_1^r and which have partially executed line 12 have also sent a CURRENT vote to p_k. Finally, processes in X_2^r have not sent a CURRENT vote. It follows that $sent\_to\_pk \leq |X_1^r| + |X_3^r|$ (R4).
5. From $nb\_current_k > n/2$ (R1) and $sent\_to\_pk \geq nb\_current_k$ (R3), we conclude $sent\_to\_pk > n/2$ (R5).
6. From $sent\_to\_pk > n/2$ (R5) and $sent\_to\_pk \leq |X_1^r| + |X_3^r|$ (R4), we conclude $|X_1^r| + |X_3^r| > n/2$ (R6).
7. From $|X_1^r|+|X_3^r| > n/2$ (R6) and $|X_1^r|+|X_2^r|+|X_3^r| \leq n$ (R2), we conclude $|X_2^r| < n/2$ (R7).

The proof is now by contradiction. Suppose that p_i decides (and consequently R7 holds), and that it exists a process p_j which enters round $r + 1$ with $est_j \neq v$ (*i.e.*, $est_j \neq est_c$). Let us consider the value $nb\_current_j$ just before p_j leaves round r (lines 4-27) and enters round $r + 1$ (line 5). There are two cases.

1. $nb\_current_j \neq 0$. In this case, according to Lemma 3, we have $est_j = est_c$ at the end of round r. A contradiction.
2. $nb\_current_j = 0$. In this case, since, according to the lemma assumption, p_j proceeds to the next round, it has received a majority of NEXT votes, *i.e.*, we have $nb\_next_j > n/2$ (R8), at the end of round r. Combining (R7) and (R8), we get $nb\_next_j > n/2 > |X_2^r|$ (R9).
   Since NEXT votes are only sent by processes belonging to $X_2^r \bigcup X_3^r$, and as (by definition) $X_2^r \bigcap X_3^r = \emptyset$, from (R9) (namely, $nb\_next_j > n/2 > |X_2^r|$) we conclude that p_j received at least one NEXT vote from a process $p_l \in X_3^r$.
   As p_l belongs to X_3^r (see Fig. 1):
   – p_l first passed through q_1. So, it executed lines 9-14, from which we conclude $nb\_current_l \neq 0$. Moreover, as $nb\_current_l \neq 0$, from Lemma 3, we get $est_l = est_c$.
   – p_l then moved from q_1 to q_2. So, it necessarily sent this NEXT vote at line 22 or 26. Hence, this vote has the form NEXT($p_l, r, est_c, deadlock\_prevention$).
   When p_j receives NEXT($p_l, r, est_c, deadlock\_prevention$) (lines 18-20) the condition at line 20 ($nb\_current_j = 0 \wedge flag_k = deadlock\_prevention$) is satisfied, and consequently, p_j has updated est_j to est_c. A contradiction.

$\square_{Lemma\ 5}$

Theorem 3 *No two processes decide different values.*

Proof. Let us consider two processes p_i and p_j which decide. Let us consider a round r and let p_c be the coordinator of round r. As before, let est_c denote the estimate of p_c during round r. We consider the two following cases.

1. Both p_i and p_j send a DECIDE message labeled with r. In that case, due to Lemma 4, they decide the same value, namely, est_c.
2. Process p_i decides v and sends a DECIDE message labeled r, while p_j sends a DECIDE message labeled r' ($r' > r$).
   Due to Lemma 5, all processes p_k that start round r+1, do so with $est_k = v$. Moreover, due to Lemma 4, $v = est_c$. In other words, the only estimate value present in any process participating in any round $r' \geq r + 1$, is now $v = est_c$. So, whatever the coordinator of round r', due to Lemma 4, the value decided by p_j will be $v = est_c$.

$\square_{Theorem\ 3}$

5 Discussion

5.1 About synchronization

The case of FIFO channels. Firstly, if the (reliable) channels are also FIFO, then the votes are received according to their round number. Let us examine the behavior of two processes p_i and p_j during a round r; p_j is the sender and p_i is the receiver. The are two cases (Fig. 1).

1. Process p_j moves from q_0 to q_2 and sends a NEXT vote (at line 16 or at line 25). As we have seen, this vote is definitive: during this round p_j will not send another vote.
2. Process p_j first sends a CURRENT vote and moves from q_0 to q_1. We have seen that, in that case, it may later send a NEXT vote (and accordingly, move to q_2); this can occur at line 22 because p_i changes its mind, or at line 26 because p_i progresses to the next round. So, if p_j sends two votes during round r, they are ordered in the following way:
   a) CURRENT(p_j,r,est_j) at line 12.
   b) NEXT(p_j,r,$est_j, deadlock\_prevention$) at line 22 or at line 26.
   Let us consider the receiver p_i. If channels are FIFO, it first receives CURRENT(p_j,r,est_j) and executes lines 9-14. Hence, whatever the value of $nb\_current_i$ was before executing lines 9-14, this value is different from 0 after. So, when p_i later receives the NEXT vote from p_j (line 18), the test done at line 20 necessarily evaluates to false.

Consequently, if channels are FIFO, line 20 can be suppressed, and both the estimate field and the flag field can also be suppressed from NEXT votes, without altering the correctness of the protocol.

Proceeding to the next round. As we have seen in the proof of Lemma 1, lines 21-23 aim at preventing deadlock situations, *i.e.*, if no process has yet decided, then no correct process will be blocked in the current round. To unblock a (potentially) blocking situation, a process p_i changes its mind by voting NEXT after having voted CURRENT. This change of mind is conditioned by the following condition $C1$ (line 21):

$$C1 \equiv ((state_i = q_1) \wedge (\mid rec\_from_i \mid > n/2) \wedge (\forall p_k : p_k \in rec\_from_i \cup suspected_i))$$

Actually, a weaker condition $C2$ (independent of the underlying failure detector) can be used by a process to change

its mind, namely[7]:

$$C2 \equiv ((state_i = q_1) \wedge (\mid rec\_from_i \mid > n/2))$$

If processes use $C2$ instead of $C1$, then, during a round r, only the current coordinator p_c can be suspected (line 15). Moreover, a process has to consult its failure detector module (line 16) only during a limited period, namely, when $state_i = q_0$. Let us note that any process p_i can use $C1$ in some rounds, and $C2$ in other rounds. Actually, as far as progress to the next round is concerned, these two conditions implement two different strategies:

- $C2$ implements an *eager* strategy. A process sends a NEXT$(-,-,-,deadlock_prevention)$ vote as soon as it suspects the possibility of a deadlock. This strategy can make processes proceed quicker to the next round, but this does not mean that a "true" deadlock has been prevented.
- $C1$ implements a *conservative* strategy. To send a NEXT $(-,-,-,deadlock\_prevention)$ vote, a process waits until it has obtained information about the state of each process.

More generally, the synchronization used to proceed to the next round can be tuned to fit particular needs. It is also possible to add a user-defined condition to $C1$ or to $C2$ (of course, in that case, termination of the consensus protocol will also depend on this condition). For example, if the underlying asynchronous system has additional behavioral properties (*e.g.*, related to real-time), such user-defined conditions could take into account some of these properties.

5.2 Cost analysis

Message complexity. It is easy to see that, per round, the number of messages of the proposed protocol is $O(n^2)$ when the underlying network is a point-to-point communication network, and $O(n)$ when the underlying network is a broadcast communication network.

Time complexity. For analyzing time complexity, we consider, on the one hand, that the duration of local processings is negligible and consequently takes no time, and on the other hand, that every message transfer takes one "logical time unit" (this constitutes assumption A1). This assumption frees us from uncertainties due to the time-freeness of asynchrony. It allows evaluation of the cost of a round-based consensus protocol by counting the minimal number of communication steps (*i.e.*, the lenght of the sequence of messages) required to reach a decision in well identified scenarios (when considering failure-free scenarios, this number is the *latency* notion introduced in [14]). Actually, this number also depends on the actual failure pattern and on the behavior of the underlying failure detector. (In the worst case, if the underlying failure detector never satisfies properties defined by $\Diamond\mathscr{S}$, it is possible that no consensus will ever be reached). So, to evaluate the intrinsic time complexity of the proposed consensus protocol, we also assume that the underlying failure detector makes no mistakes (this constitutes assumption A2). This assumption frees us from arbitrary behaviors of the failure detector[8].

Assuming A1 and A2, we evaluate the time complexity by counting the number of communication steps involved in a round r in the following failure scenarios (p_c denotes the coordinator of round r):

- Pattern (1): At the end of round r, the current coordinator p_c has not crashed.
  In that case, there are 2 communication steps. During the first step, p_c sends a vote CURRENT(p_c, r, est_c) and all processes receive it (line 9)[9]. The second step starts when each correct process propagates this vote (line 12), and terminates when they have received a majority of CURRENT votes allowing them to decide (line 14). Thus, during the current round, a decision is obtained after 2 communication steps.
- Pattern (2): The current coordinator p_c has crashed before round r.
  In that case, at the beginning of round r, all correct processes suspect p_c (line 15) and send a NEXT vote to proceed to the next round (line 16). As there is a majority of correct processes, each of them receives a majority of NEXT votes after one time unit. So, in that scenario, the proposed protocol requires only one communication step in order to proceed to round $r + 1$.

5.3 A comparison with other protocols

Section 3.1 briefly sketched Chandra-Toueg's [4] (CT) and Schiper's [14] (SC) consensus protocols. Assuming that the reader is familiar with the CT and the SC protocols, this section compares these protocols with the one proposed in this paper (HR protocol). They all are based on the rotating coordinator paradigm and proceed in asynchronous rounds. We first compare how each protocol ensures safety (Validity and Agreement) and liveness (Termination). Then, we compare these protocols in some failure patterns.

How is the safety property ensured? The Validity property is easy to ensure. As indicated before, the Agreement property is the difficult part because, due to process crashes and possibly erroneous failure suspicions, it is possible that distinct processes decide during distinct rounds. As with other consensus protocols (*e.g.*, [8]), we use the *value locking* notion to explain how Agreement is guaranteed by CT, SC and HR: a value gets *locked* as soon as it has been adopted by a protocol thus becoming the decision value (independently from the fact processes know it).

In CT, the locking mechanism works in the following way. During a round, a value proposed by the current coordinator gets locked as soon as it has been positively acknowledged by a majority of processes. When a process positively

[7] $C2$ is weaker than $C1$ because $C1 \Rightarrow C2$. The reader can easily check that Case 2, in the proof of Lemma 1, remains correct when $C2$ is used instead of $C1$.

[8] In other words, to analyze time complexity, we assume that the underlying network behaves as a synchronous network, and that the underlying failure detector behaves as a perfect failure detector [4] (*i.e.*, a failure detector that makes no mistakes).

[9] Of course, we suppose that, at line 6, the sending by p_c of a message to itself takes no time.

acknowledges a value, it considers it as its new estimate. A timestamping mechanism associated with estimates, ensures that if several estimates are locked during distinct rounds, they are necessarily equal to a same value initially proposed by a process.

In SC and HR, a value gets locked as soon as it has been forwarded by a majority of processes, during a round. Both protocols ensure that if a value v has been locked during a round r, then any process entering a round $> r$ has v as the estimate value. Let $\mathscr{P}$ be this property. SC and HR differ in the way they guarantee $\mathscr{P}$. More precisely:

- In SC, during a round, a process can be in two states $phase_1$ or $phase_2$. In $phase_1$, p_i tries to establish a decision value. If a process p_i has received a majority of SUSPICION messages (indicating that a majority of processes suspect the current coordinator), it moves from $phase_1$ to $phase_2$: this means it will ineluctably progress to the next round $r+1$. But, while it is in $phase_2$ and before proceeding to $r+1$, p_i exchanges ESTIMATE2 messages (carrying estimate values) with other processes. These exchanges ensure that property $\mathscr{P}$ holds.
- In HR, ensuring property $\mathscr{P}$ requires neither an additional step, nor additional messages. A process guarantees this property when it sends a NEXT vote to prevent a possible deadlock (line 22). Such a vote carries the value proposed by the coordinator of round r (this value has possibly been decided by some processes during this round). To be more explicit, let us consider a process p_i that decides during round r and a process p_j that progresses to $r+1$. In that case, at the end of round r we have $nb\_current_i > n/2$ and $nb\_next_j > n/2$. At least one process p_k belongs to both majorities: so, it sent first a CURRENT vote and then a NEXT vote. The dissymmetric automaton governing the behavior of p_k, shows that p_k has first adopted the coordinator estimate est_c (when it moved from q_0 to q_1 and sent the CURRENT vote), and later has changed its mind (when it moved from q_1 to q_2 and sent a NEXT$(p_k, r, est_c, deadlock_prevention)$ vote to p_j). When p_j received this vote, it updated est_j to est_c (*i.e.*, the value proposed by the current coordinator and decided by p_i).

How is the liveness property ensured? In the three protocols Termination is based on the properties of the underlying failure detector and on the assumption that a majority of processes is correct. CT, SC and HR use these properties in the following way:

- Combined with the *rotating coordinator* paradigm, the Eventual Weak Accuracy property of the underlying failure detector is used by each protocol to ensure that it is possible to eventually reach a round r' during which the coordinator will not be suspected.
- Each protocol uses the Strong Completeness property of the underlying failure detector, and assumes a majority of correct processes, to make processes eventually progress to round $r+1$ if no value has been decided during round r. This prevents deadlock and consequently ensures that the round r' will actually be reached.

A subtle difference in the way each protocol ensures the Termination property is revealed by their Termination proofs. These proofs are done in one way for CT and HR, and in another way for SC. More precisely, the Termination proof of CT and HR explicitly considers the time t after which all faulty processes have crashed (for HR, see the proof of Theorem 2, footnote 5). The Termination proof of SC does not require considering such a time t. This difference comes from the fact that these protocols use very different mechanisms to prevent deadlocks. Consequently, this makes the protocols terminate in different ways according to failure and failure suspicion scenarios.

Number, type and size of messages in CT, SC and HR. Let us first consider the number of messages exchanged by each protocol during a round. As previously noted, CT uses Skeen's centralized message exchange pattern, and consequently a round requires $3(n-1)$ messages. When they decide during a round, both SC and HR require $n(n-1)$ messages. When it proceeds to the next round, HR does not require more messages, while SC does due its additional communication step (but this number is still $O(n^2)$). Clearly, when considering a round of coordinator-based $\Diamond\mathscr{S}$-based consensus protocols, there is a tradeoff between the time complexity (which favors SC and HR) and the message complexity (which favors CT).

CT, SC and HR use three types of messages: ESTIMATE, ACK/NACK and DECIDE messages in CT, ESTIMATE, SUSPICION and DECIDE messages in SC, and CURRENT, NEXT and DECIDE messages in HR. Let us consider the size of the biggest messages used by each protocol (ESTIMATE messages in CT and SC, and NEXT messages in HR). All those messages carry their identity, which is made of the pair (identity of the sender process, sequence number)[10], and an estimate of the decision value (*e.g.*, est_i in HR). Then, CT, SC and HR differ in the following way:

- In CT: ESTIMATE messages carry an additional potentially unbounded value, namely, a timestamp (the value of which is a round number).
- In SC: ESTIMATE messages carry two additional bounded values, namely:
  - A boolean value indicating the phase number (1 or 2) of the current round at which the messages is sent,
  - A process identity, namely, the identity of the process to which refers the current estimate value (for each process p_i, this value appears in a field of its current estimate).
- In HR: NEXT messages carry an additional boolean value (namely, its last field whose value is $suspicion$ or $deadlock\_prevention$).

So, with respect to CT, no additional timestamp is carried by messages in SC and HR. Moreover, when compared to ESTIMATE messages used in SC, a NEXT message used in HR has not to carry an additional process identity, thereby, saving $log_2(n)$ bits.

CT, SC and HR with reliable failure detectors. We first study the case where the underlying failure detector offers a very

[10] Here the sequence number is the corresponding round number.

Table 1. Number of steps with reliable failure detectors

	$FP0$	$FP1$	$FP2$	$FP3$
CT	4 ([11])	4	4	4
SC	2	4	6	8
HR	2	3	4	5

good quality of service, *i.e.*, when it makes no mistake. Remember that, in many systems, failure detectors can be tuned to make mistakes very infrequently.

Let us consider a system composed of 7 processes: $p_1, \ldots, p_7$, and let us examine the following four failure patterns. We also assume that processes that are not correct have crashed before the consensus is launched. Process p_1 is the first coordinator, p_2 is the second, etc.

- $FP0$: All processes are correct.
- $FP1$: All processes but p_1 are correct.
- $FP2$: All processes but p_1 and p_2 are correct.
- $FP3$: All processes but p_1, p_2 and p_3 are correct.

For each failure pattern, Table 1 defines the total number of communication steps required by a given protocol to decide (The "communication step" notion is the one introduced in the "Time complexity" paragraph, Sect. 5.2. Actually, the "number of communication steps" measure is close to *latency* notion [14] defined from failure-free runs).

In all these failure patterns, CT always requires 4 steps. This is due to the fact that, when the underlying failure detector makes no mistake, CT requires (1) 4 steps to decide during a round if the current coordinator has not crashed, and (2) no communication steps to proceed from a round with a crashed coordinator to the next round (a process that suspects the current coordinator immediately proceeds to the next round without waiting for messages). In the same context, if, during a round, the current coordinator has crashed SC requires 2 communication steps for processes to progress to the next round; HR requires only one.

It is important to note that, while failure are rare in practice, they do occur. This means that the failure patterns most often encountered are $FP0$ and $FP1$.

SC and HR with unreliable failure detectors. From a syntactical point of view, when there are neither failures nor suspicions, both SC and HR follow a message exchange pattern similar to the one used by Skeen in his decentralized commit protocol [15]. Their respective behaviors differ when there are failures or erroneous suspicions. Two of their basic differences lie (1) in the way they prevent deadlock, and (2) in the way they ensure the Agreement property. Each protocol uses a specific solution to address these problems. More precisely, they use distinct conditions to allow processes to progress to the next round, and distinct techniques for ensuring a single value is decided. As we have previously examined the latter point (in the Section titled "How is the safety property ensured?"), we focus here on deadlock prevention and on other differences between the protocols. The progress to the next round is related to SUSPICION messages in SC, and to NEXT votes in HR. But these messages/votes are not equivalent. Consequently, SC and HR can behave very differently when there are erroneous failure suspicions.

[11] Actually, this number can easily be reduced to 3 as shown in [1, 14].

Although HR compares favorably with SC when the underlying failure detector makes no mistake, this can no longer be true when erroneous failure suspicions occur. Intuitively, SC and HR present a fundamental difference in the way they behave with respect to failure suspicions: SC "does not trust" the failure detector while HR "trusts" it. The following particularities characterize each protocol.

- In SC, during a round, only the coordinator p_c can be suspected. A process broadcasts a SUSPICION message only when it suspects p_c. A process decides to proceed to the next round when it has received a majority of SUSPICION messages. So, when a process progresses to the next round, the coordinator is suspected by a majority of processes. This "majority of suspicions" condition is well-suited to resist erroneous failure suspicions: it favors a decision during the current round when the coordinator has not crashed and less than a majority of processes (falsely) suspects it.
- In HR, a process proceeds to the next round when it has received a majority of NEXT votes. But, as we have seen, two different conditions are associated with the sending of NEXT votes.
  - The first condition, used at lines 15-16, states that the issuing process p_i suspects the coordinator: this vote has the form NEXT$(-, -, -, suspicion)$. In that case, as soon as p_i has sent such a vote, differently from SC, it can no longer vote CURRENT. This means that if a process has sent NEXT$(-, -, -, suspicion)$, it can no longer favor a decision during the current round. (But, note that this does not prevent a decision to be taken during the current round.)
  - The second condition, used at line 21, is associated with deadlock prevention: the corresponding vote is NEXT$(-, -, -, deadlock_prevention)$. Sending this vote does not means that the issuing process p_i suspects the coordinator, it only means that p_i either suspects or has got a vote from any process. Note that, when p_i sends such a vote, it has previously sent a CURRENT vote, thereby favoring the decision during the current round.

So, in presence of erroneous failure suspicions, it is difficult to compare failure detector-based consensus protocols. Actually, an in-depth performance study of those protocols has to be based on a probabilistic model of the *quality of service* of the underlying failure detector (the highest quality being offered by a perfect failure detector). Such a performance characterization is beyond the study addressed in this paper. It is however important to note that both SC and HR have the following interesting property: in failure-free runs, erroneous suspicions do not force processes to progress to the next round, as long as a majority of failure detector modules do not make mistakes. Such a property is not guaranteed by CT, where there are runs in which it is possible that an erroneous suspicion made by a single failure detector module prevents the current coordinator from deciding during the current round.

6 Conclusion

The Consensus problem is a fundamental paradigm for fault-tolerant asynchronous systems. It abstracts a family of problems known as Agreement (or Coordination) problems: any solution to consensus can serve as a basic building block for solving agreement problems (such as, for example, atomic commitment or atomic broadcast). Solving consensus in an asynchronous system is not a trivial task: it has been proven (1985) by Fischer, Lynch and Paterson that there is no deterministic solution in asynchronous systems subject to even a single crash failure. To circumvent this impossibility result, Chandra and Toueg have introduced the concept of unreliable failure detectors (1991), and have studied how these failure detectors can be used to solve consensus in asynchronous systems with crash failures.

This paper has presented a new consensus protocol, based on a failure detector of the class $\Diamond\mathscr{S}$. Like previous protocols, the proposed protocol is based on the rotating coordinator paradigm and proceeds in asynchronous rounds, to benefit from the Eventual Weak Accuracy property of $\Diamond\mathscr{S}$. More generally, in addition to a new consensus protocol, the main contributions of this paper are the followings.

- A first contribution lies in the design simplicity of the proposed protocol. Its design relies on a simple and original combination of well-known mechanisms: a voting mechanism, a small finite state automaton that manages the behavior of each process, and allowing a process to change its mind during a round. This shows that simple principles can go a long way.
- A second contribution lies in the time efficiency of the protocol when the underlying failure detector makes no mistake, whether there are failures or not. In those cases, (1) if the current coordinator has not crashed, the decision is obtained in two communication steps (as SC [14]); (2) if the current coordinator has crashed, only one communication step is required to proceed to the next round in order to benefit from another coordinator. This efficiency in no erroneous suspicion runs has been obtained by using a simple idea, namely that the protocol trusts the underlying failure detector. Consequently, one of its practical interests is the graceful degradation it provides in presence of process crashes, when the underlying failure detector makes no mistakes. An important point of the protocol is the fact that its efficiency has not been sacrificed in order to obtain design simplicity. Another important point is that its time efficiency has not been obtained at the price of an increase in the number of messages, or in their size.
- Another, but not least, contribution of this paper concerns the demystification of the consensus problem. Although the simplicity issue in the design of a protocol is not directly related to its correctness or to its efficiency, we think that the simplicity of a solution greatly contributes to the demystification of a problem. As far as the consensus problem is concerned, we hope that the proposed protocol is a step in this direction. This should "*lead to consider consensus as it should be, i.e., as a basic building block for implementing fault-tolerant distributed systems, rather than an interesting problem for theoreticians*" [14].

Schiper's protocol [14] and the proposed protocol can be seen as two $\Diamond\mathscr{S}$-based consensus protocols of the same family in the following sense: in failure-free and erroneous suspicion-free runs, they use the same "Skeen's decentralized message exchange pattern" [15]. Consequently, in those runs, they have the same time and message complexities. This is no longer true in presence of crashes. The aim of Schiper was to propose a protocol that is time efficient in runs where there are neither failures, nor erroneous suspicions. This was a significant advance. The proposed protocol can be seen as a further step in this direction: it is time efficient when the underlying failure detector behaves reliably, whether there are failures or not. The design of consensus protocols that, in addition, would remain time efficient in "some" erroneous failure suspicion scenarios, would constitute further significant advances. But, when considering Schiper's protocol, Chandra-Toueg's protocol and the proposed protocol it is important to say (as indicated at the end of Sect. 5.3) that none of them definitely outperforms the other two in presence of erroneous suspicions.

To conclude, let us note that the following question (which was out of the scope of the paper) remains open. It concerns the optimality [7] of consensus protocols in erroneous suspicion-free runs. In such a context, when considering consensus protocols based on a failure detector of the class $\Diamond\mathscr{S}$, are 2 and 3 the minimal numbers of communication steps required to decide at the end of the first round, and to decide at the end of the second round, respectively?

Acknowledgments. The authors are grateful to Roberto Baldoni, Jean-Michel Hélary, Achour Mostéfaoui and Frédéric Tronel whose comments on drafts of this paper helped improve the presentation. The comments and suggestions of the anonymous referees were also instrumental in improving the paper. The authors are also grateful to a referee who has suggested a shorter and "more natural" proof for Lemma 1. They also thank Udo Fritzke and Philippe Ingels who have implemented the protocol on a network of workstations.

References

1. Aguilera M.K. and Toueg S. Randomization and Failure Detection: A Hybrid Approach to Solve Consensus. In Proc. of the 10th Int. Workshop on Distributed Algorithms, Springer-Verlag, LNCS 1151 (Ö. Babaoğlu and K. Marzullo Eds), pp. 29–39, Bologna, Italy, October 1996
2. Birman K.P. and Joseph T.A. Reliable Communication in the Presence of Failures. ACM Transactions on Computer Systems,5(1):47–76, February 1987
3. Birman K.P. Building Secure and Reliable Network Applications. Manning Publication Co., Greenwich, CT, 1996, 591 pages
4. Chandra T. and Toueg S. Unreliable Failure Detectors for Reliable Distributed Systems. Journal of the ACM, 34(1):225–267, March 1996 (A preliminary version appeared in Proc. of the 10th ACM Symposium on Principles of Distributed Computing, pp. 325–340, 1991)
5. Chandra T., Hadzilacos V. and Toueg S. The Weakest Failure Detector for Solving Consensus. Journal of the ACM, 43(4):685–722, July 1996 (A preliminary version appeared in Proc. of the 11th ACM Symposium on Principles of Distributed Computing, pp. 147–158, 1992)
6. Dolev D., Dwork C. and Stockmeyer L. On the Minimal Synchronism Needed for Distributed Consensus. Journal of the ACM, 34(1):77–97, January 1987

7. Dolev D., Reischuk R. and Strong R., Early Stopping in Byzantine Agreement. Journal of the ACM, 37(4):720–741, April 1990
8. Dwork C., Lynch N. and Stockmeyer L. Consensus in the Presence of Partial Synchrony. Journal of the ACM, 35(2):288–323, April 1988
9. Fischer M.J., Lynch N. and Paterson M.S. Impossibility of Distributed Consensus with One Faulty Process. Journal of the ACM, 32(2):374–382, April 1985
10. Gray J.N. Notes on Database Operating Systems. In Operating Systems: An Advanced Course, Springer-Verlag, LNCS 60 (R. Bayer, R.M. Graham and G. Seegmuller Eds), pp. 393–481, 1978
11. Guerraoui R. Revisiting the Relationship Between Non-Blocking Atomic Commitment and Consensus. In Proc. of the 9th Int. Workshop on Distributed Algorithms, Springer-Verlag, LNCS 972 (J-M. Hélary and M. Raynal Eds), pp. 87–100, Le Mont-Saint-Michel, France, September 1995
12. Guerraoui R. and Schiper A. Consensus: the Big Misunderstanding. In Proc. of the 6th IEEE Workshop on Future Tends of Distributed Computing Systems, IEEE Computer Society, pp. 183–188, Tunis, Tunisia, October 1997
13. Malkhi D. and Reiter M. Unreliable Intrusion Detection in Distributed Computations. In Proc. of the 10th IEEE Computer Security Foundations Workshop, pp. 116–124, Rockport, MA, June 1997
14. Schiper A. Early Consensus in an Asynchronous System with a Weak Failure Detector. Distributed Computing, 10:149–157, 1997
15. Skeen D. Non-Blocking Commit Protocols. Proc. Int. ACM-SIGMOD Conference on Management of Data, pp. 133–142, 1981

Michel Hurfin received the PhD degree in Computer Science from the University of Rennes, France, in 1993. His dissertation topic addressed execution replay and property detection in distributed applications. In 1994, he spent one post-doctoral year at Kansas State University, Manhattan, in the research group of Professor M. Mizuno. Dr. Hurfin is currently a researcher at the INRIA unit of Rennes. His research interests include distributed systems, software engineering and middleware for distributed operating systems. Recently, he has initiated research on distributed fault-tolerant middleware.

Michel Raynal has been a professor of Computer Science at the University of Rennes, France, since 1984. At IRISA (CNRS-INRIA-University joint computing laboratory located in Rennes) he is the leader of the ADP (Distributed Algorithms and Protocols) research group that he created in 1986. He has served as program co-chair of WDAG (now DISC, the Symposium on Distributed Computing) in 1989 and 1995. He has also served several times as vice-chair for the “Distributed Algorithms” track of the IEEE Int. Conference on Distributed Computing Systems. Furthermore, he has served as a PC member in a lot of international conferences. Michel Raynal has written seven books (2 published by Wiley & Sons, and 2 by the MIT Press). He has published more 50 papers in journals and 100 in conferences. Together with other european leaders, he is currently a member of the ESPRIT Basic Research Network of excellence in Distributed Computing Architectures (CABERNET) currently headed by B. Randell. On the theoretical side, Michel Raynal’s research interests include distributed algorithms, distributed systems, distributed computing and fault-tolerance. His main interest lies in the fundamental concepts, principles and mechanisms that underly the design and the construction of distributed systems. Among them, he is currently interested in the study of the Causality concept and in the Consensus problem. On the practical side, Michel Raynal is interested in the implementation of reliable communication primitives, the consistency of distributed data, the design and the use of checkpointing protocols and the set of problems that can be solved on top of a consensus “building block”.