A Consensus Protocol Based on a Weak Failure Detector and a Sliding Round Window*

Michel HURFIN† Raimundo MACÊDO‡ Achour MOSTEFAOUI† Michel RAYNAL†

† IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France
‡ LaSiD-CPD-UFBA, Campus de Ondina, CEP 40170-110 Bahia, Brazil

Abstract

This paper revisits the "sliding window" notion commonly encountered in communication protocols and applies it to the round numbers of round-based asynchronous protocols. This approach is novel. To illustrate its benefits, the paper presents an original weak failure detector-based consensus protocol that allows each process to be simultaneously involved in several rounds. The rounds in which a process is simultaneously involved define its "sliding round window".

The proposed approach has several advantages. It fits better to the uncertainty created by the asynchrony and failures, and consequently permits to design efficient round-based asynchronous protocols. Maybe more important, it also provides a better understanding of the global synchronization that manages the protocol progress from round to round. This appears clearly in the proposed failure detector-based consensus protocol, where the "sliding round window" allows to dynamically define the message exchange pattern for each round separately.

1 Introduction

Context. The *Consensus* problem lies at the heart of an important family of distributed computing problems, namely, the family of agreement problems. It actually constitutes a basic building block on top of which solutions to practical agreement problems can be designed. A typical agreement problem is *Atomic Broadcast* where processes have to agree on a common message delivery order.

The consensus problem can be informally defined in the following way. Each process proposes a value, and all non-crashed processes have to agree on a common value which has to be one of the proposed values. Solving this problem in asynchronous distributed systems where processes can crash is far from being a trivial task. More precisely, it has been shown by Fischer, Lynch and Paterson [3] that there is no (deterministic) solution to this problem as soon as processes (even only one) may crash. A noteworthy advance proposed to circumvent this impossibility result lies in the *Unreliable Failure Detector* concept, proposed and investigated by Chandra, Hadzilacos and Toueg [1, 2]. A failure detector is basically defined by two properties: a *Completeness* property that is on the actual detection of failures, and an *Accuracy* property that limits the mistakes a failure detector can make. Chandra and Toueg have defined several completeness and accuracy properties that allowed them to define eight classes of failure detectors. In this paper, we are interested in asynchronous distributed systems equipped with a failure detector of the class $\diamond\mathcal{S}$. This class includes all the failure detectors that satisfy *strong completeness* ("Eventually, every crashed process is suspected by every correct process"), and *eventual weak accuracy* ("There is a time after which there is a correct process that is never suspected").

Several consensus protocols have been designed for asynchronous distributed systems equipped with a failure detector of the class $\diamond\mathcal{S}$ [1, 4, 8, 9]. They all are based on the same iterative control structure: processes proceed in asynchronous rounds whose aim is to make them eventually converge to the same value (and then decide on it). Each round r is managed by a predetermined coordinator that tries to impose its current estimate of the decision value as the final decision value.

In all these protocols, each of the n processes executes rounds sequentially, until it decides (or crashes). One of their main differences lies in the message exchange pattern used during each round. More precisely:

• Each round of the Chandra-Toueg's protocol [1] is based on a centralized message exchange pattern. The messages are from/to the round coordinator. A process p_i progresses from a round r to the next round ($r+1$) when it has sent a positive acknowledgment to the coordinator of r, or when it suspects it (it then sends it a negative acknowledgment).

It is important to notice that, when the process that co-

*This work is partially supported by a France Telecom grant 98 1B 123 and by CNPq/ProTeM-CC (Brazil), ARGO Project.

ordinates a round r suspects all the other processes, it can "directly" progress from r to the next round it will have to coordinate. This jump does not require intermediate synchronization. Only when a process p_i becomes the coordinator of a new round r', it has to synchronize with the other processes: it then waits for messages from the other processes; those messages ensure that if a value v has been decided by other processes, p_i cannot start r' with an estimate of the decision value different from v (agreement property).

• The protocols described in [4, 8, 9] use a decentralized message exchange pattern. The end of each round is managed by a synchronization barrier that prevents a process to proceed to the next round with a value different from the decided value if a value has been decided by some processes. This barrier requires a process to receive "enough" messages from the other processes before being allowed to progress to the next round.

Despite these important design differences, the previous protocols share an important design feature, namely, once involved in a round r, a process executes statements related to only this round r, and does no longer participate in any round $r' < r$.

Content of the paper. The "*Sliding Window*" notion has been introduced very early in communication protocols. Its aim was to allow the design of efficient communication protocols despite asynchrony and message losses. The *sliding window* notion applies to message sequence numbers. This paper revisits this notion and applies it to the round numbers of round-based asynchronous protocols, hence the notion of "*Sliding Round Window*". When considering a round-based protocol, the idea is to allow a process to simultaneously participate in several consecutive rounds[1]. To illustrate this approach, the paper presents a $\diamond\mathcal{S}$-based consensus protocol where a process that is executing a round r, continues executing rounds $r' < r$: each process has a *sliding round window* that defines the consecutive rounds it is currently involved in. In that way, a process has more means to cope with the asynchrony of the execution. As far as $\diamond\mathcal{S}$-based consensus protocols are concerned, all the protocols we know allow a process to participate in only a single round at a time. To our knowledge, this is the first time the *sliding window* notion is applied to the rounds of a failure detector-based consensus protocol.

When applied to the design of a $\diamond\mathcal{S}$-based consensus protocol, the proposed *sliding round window* approach has both conceptual and practical advantages. One lies in the better understanding it provides on the type of synchronization processes have to follow in order the consensus agreement property be never violated. Actually, the constraint imposed by this synchronization shows that it is sufficient for each process p_i to strongly synchronize with the other processes only every n rounds (namely, at rounds $i + kn$, $\forall k \geq 0$) without being prevented from asynchronously participating in n consecutive rounds.

Another advantage lies in the possibility for the protocol to determine, for each round separately, the message exchange pattern of that round (and consequently, its number of messages). All the $\diamond\mathcal{S}$-based consensus protocols proposed so far divide, at each round, the processes into two classes: the ones that can test a decision predicate ("deciding" class), and the ones that cannot [6]. Hence, if a value is decided during r, the processes of the first class can "directly" decide it. (A process of the other class can only proceed to the next round, or be informed of a decision by a process of the first class.) The definition of the deciding class associated with a round has an important consequence on the pattern of messages exchanged during that round (because a process has to receive messages to decide). The definition of the deciding class is implicit in the previously proposed $\diamond\mathcal{S}$-based consensus protocols. In [1], the deciding class associated with a round r includes only the coordinator of r, and consequently the number of messages exchanged during a round is $O(n)$. In the other protocols, whatever the round, the deciding class is statically defined and includes all the processes; the number of messages exchanged during a round is then $O(n^2)$. The proposed protocol has the noteworthy and original feature allowing the processes to dynamically define, for each round separately, which of them belong to the deciding class, i.e., the processes define the message exchange pattern of each round. The only constraint that has to be satisfied is that, given a round r, the coordinators of r and $r + 1$ have to belong to the deciding class of r (the first is required to allow the protocol to terminate, the second to ensure that the agreement property will never be violated). From an operational point of view, each process p_i defines for each round r the set of processes to which it sends messages allowing to test the decision predicate. Hence, for each round, the number of messages exchanged can vary from $O(n)$ (centralized pattern) to $O(n^2)$ (fully distributed pattern). This shows that (1) each round can follow its own message exchange pattern, and (2) the centralized message exchange pattern and the fully distributed message exchange pattern are not antagonistic but merely two instantiations of a more generic message exchange pattern. Furthermore, this versatility can be exploited by the protocol to adapt the number of messages according to the its current view of the network load.

To summarize, the proposed sliding round window $\diamond\mathcal{S}$-based consensus protocol enjoys the following properties. First, it allows each process to participate in several consecutive rounds, while synchronizing with the other pro-

[1]Or equivalently, the processes can simultaneously be involved in several consecutive *ballots*, according to Paxos terminology [7].

cesses only every n rounds. In addition to a reduction in the number of synchronizations, it (1) provides a better understanding of the synchronization requirement of rotating coordinator-based consensus protocols; (2) defines waiting conditions weaker than previous protocols; and (3) allows, for each round, the processes to define the message exchange pattern of that round (hence, the protocol can adapt its behavior to the network load).

The paper is made up of five sections. Section 2 defines the context of the study: the computation model, the class of eventual weak failure detectors, and the consensus problem. Section 3 presents the $\diamond\mathcal{S}$-based consensus protocol based on the window notion. Section 4 discusses the protocol. The correctness proof of the uniform agreement property is given in Section 5. The proof of the other properties is omitted for lack of space, it can be found in [5].

2 Computation Model and Consensus

Asynchronous System with Crash Failures. The system model is patterned after the one described in [1, 3]. We consider a system consisting of a finite set Π of $n > 1$ processes, namely, $\Pi = \{p_1, \dots, p_n\}$. 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. Let f be the maximum number of processes that can crash. It is assumed that $f < n/2$.

Processes communicate and synchronize by sending and receiving messages through channels. Every pair of processes is connected by a channel. Channels are not required to be FIFO. They are only assumed to be reliable in the following sense: they do not create, duplicate, alter or lose messages. This means that a message sent by a process p_i to a process p_j is assumed to be eventually received by p_j, if p_j is correct.

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

The Class $\diamond\mathcal{S}$ of Unreliable Failure Detectors. A formal introduction to failure detectors is provided in [1]. 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 the identities of the 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 or removed from a set $suspected_i$ according to whether p_i's failure detector module currently suspects p_j or not. As in papers devoted to failure detectors, we say "process p_i suspects process p_j" at some time, if at that time we have $j \in suspected_i$.

As indicated in the Introduction, a failure detector class is formally defined by two abstract properties, namely a *Completeness* property and an *Accuracy* property. In this paper we consider the class of *eventual weak* failure detectors [1]. This class, denoted $\diamond\mathcal{S}$, includes all the failure detectors that satisfy the following properties:

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

It is important to note that any failure detector that belongs to $\diamond\mathcal{S}$ can make an arbitrary number of mistakes.

The Consensus Problem. In the consensus problem, every correct process p_i *proposes* a value v_i and all correct processes have to *decide* on some value v, in relation with the set of proposed values. More precisely, the *Consensus* problem is defined by the three following properties [1, 3]:
- Termination: Every correct process eventually decides on some value.
- Validity: If a process decides v, then v was proposed by some process.
- Uniform Agreement: No two processes (correct or not) decide differently.

Let us consider an asynchronous distributed system equipped with a failure detector of the class $\diamond\mathcal{S}$. The following results are attached to the consensus problem when one tries to solve it in such a context.
- (1) The consensus can be solved in such a system [1, 4, 8, 9].
- (2) $f < n/2$ is a necessary requirement [1].
- (3) $\diamond\mathcal{S}$ is the weakest class of failure detectors that allows to solve the consensus problem [2].

3 The Consensus Protocol

3.1 Rotating Coordinator and Sliding Window

As in the other $\diamond\mathcal{S}$-based consensus protocols, the processes of the proposed protocol proceed in asynchronous rounds. The rounds are started sequentially. A round r is coordinated by a predetermined process, namely p_{cc} (the identity cc is computed from r using the mod function). This is the *Rotating Coordinator* paradigm [1]. Moreover, in addition to p_{cc}, the proposed protocol requires another process to play a particular role during each round r, namely the coordinator of the next round $(r+1)$, denoted p_{nc}.

As announced in the Introduction, a process p_i may be simultaneously involved in up to n consecutive rounds, namely, the rounds numbered r (the last round it has entered), $r-1, \ldots$, until possibly $r-(n-1)$. Those round numbers define the current *Round Window* of p_i. This window (of maximal size n) slides according to the progression of p_i. A process p_i has consequently local variables related to the management of its last n rounds.

When compared to previous $\diamond\mathcal{S}$-based consensus protocols, the fact that each process can participate simultaneously in several rounds creates a new difficulty to guarantee the agreement property. The protocol copes with it by using appropriate data structures and a weak synchronization.

3.2 Data Structures

Each process p_i manages a set of local variables that define its local state. These variables can be classified according to the rounds they are on. We have the following sets of local variables.

Variables whose scope is the whole computation. There are three such variables.

• r_i: number of the last round joined by p_i.

• est_i: current estimate of the decision value. (Initialized to v_i, the value proposed by p_i.)

• ts_i: local scalar clock (whose value is a round number) used to timestamp est_i; it actually measures the up-to-dateness of est_i.

The processes exchange their estimate values in order to converge to a single decided value (see Section 3.3). To this end, they send EST messages that always carry the three previous values. The behavior of ts_i obeys the following rules (close to the rules that manage Lamport scalar clocks):

- (1) When p_i enters a new round r_i of which it is the coordinator, it increases ts_i to r_i,
- (2) When p_i receives an $\text{EST}(r, est, ts)$ message, it updates ts_i to $\max(ts, ts_i)$.

It follows from these rules that $\forall\ i$: r_i and ts_i never decrease; moreover, $ts_i \leq r_i$.

Variable whose scope is the last round. There are two such variables.

• pos_ack_i: boolean flag indicating whether p_i has sent an $\text{EST}(r_i, est_i, ts_i)$ message with $ts_i = r_i$. This message is interpreted as a positive acknowledgment. (similar "positive/negative" interpretation is given to messages that carry estimates in [1].)

• neg_ack_i: boolean flag indicating whether p_i has sent an $\text{EST}(r_i, est_i, ts_i)$ message with $ts_i \neq r_i$. This message is interpreted as a negative acknowledgment.

Variables related to the management of the window. These variables are:

• $prevc_i$: number of the last round coordinated by p_i. (Initial value: 0).

• $bnextc_i$: number of the round that precedes the next round that p_i will coordinate. (Initially $bnextc_i = i-1$. Then, $bnextc_i = prevc_i + n - 1$). (Note that, when $bnextc_i \neq r_i$, this pointer falls outside the current window.)

These two values could be computed from the pair (i, r_i), each time they are needed. They are managed as two variables to ease the presentation of the protocol.

As announced previously, during each round, processes are required to send $\text{EST}(r, est, ts)$ messages. As indicated, such a message with $r = ts$ is interpreted as a "positive acknowledgment" [1, 4] with respect to r; its aim is to favor the decision during that round. The following three variables are related to the number of EST messages received by p_i. The two first variables count positive and negative acknowledgments (related to particular rounds), while the last variable counts only positive acknowledgments.

• tot_prevc_i: total number of $\text{EST}(r, est, ts)$ received with $r = prevc_i$. (Initial value: n.)

• tot_bnextc_i: total number of $\text{EST}(r, est, ts)$ received with $r = bnextc_i$. (Initial value: 0.)

• $pos_i[r_i - (n-1) : r_i]$: sliding array of integers $\in \{0, \ldots, (f+1)\}$. The entry $pos_i[r]$ contains the number of $\text{EST}(r, est, ts)$ received with $r = ts$. (Number of positive acknowledgments received with respect to the round r. When p_i starts r: $pos_i[r] = 0$.)

3.3 The Protocol

The protocol executed by a process p_i is formally described in Figure 1. A process p_i starts a consensus execution by invoking $\mathsf{Consensus}(v_i)$. It terminates it when it executes the statement $\mathsf{return}(est)$ which provides it with the decided value, namely est. The termination at line 17 is "direct" while the one at line 24 is "indirect".

For each process p_i, the protocol is made up of three tasks: $T1$, $T2$ and $T3$. The task $T1$ manages the sequential launching of rounds, while $T2$ and $T3$ are devoted to the processing of received messages. $T1$ and $T2$ constitute the core of the protocol.

The primitive *send* $\text{EST}()$ *to* X, where X is a process set is a shortened form for $\forall p_x \in X$ **do** *send* $\text{EST}()$ *to* p_x **enddo**. Moreover, *broadcast*$\text{EST}() \equiv$ *send* $\text{EST}()$ *to* Π.

Task $T1$. This task repeatedly launches rounds until a decision is reached. The launching of a new round does not mean that p_i stops participating in previous rounds. Let us

Function Consensus(v_i)

(1) $r_i \leftarrow 0; est_i \leftarrow v_i; ts_i \leftarrow 0; nc \leftarrow 1; prevc_i \leftarrow 0; tot\_prevc_i \leftarrow n; bnextc_i \leftarrow (i-1); tot\_bnextc_i \leftarrow 0;$

Task $T1$:
(2) **while** $true$ **do**
(3) $r_i \leftarrow r_i + 1; cc \leftarrow nc; nc \leftarrow (r_i \bmod n) + 1; pos\_ack_i \leftarrow false; neg\_ack_i \leftarrow false; pos_i[r_i] \leftarrow 0;$
(4) **if** $(i = cc)$
(5) **then** $prevc_i \leftarrow r_i; tot\_prevc_i \leftarrow 0; bnextc_i \leftarrow r_i + n - 1; tot\_bnextc_i \leftarrow 0;$
(6) $ts_i \leftarrow r_i;$
(7) *broadcast* $\text{EST}(r_i, est_i, ts_i); pos\_ack_i \leftarrow true$
(8) **else** **wait until** $((cc \in suspected_i) \vee pos\_ack_i);$
(9) **if** $(\neg pos\_ack_i)$ **then** *send* $\text{EST}(r_i, est_i, ts_i)$ *to* $\{p_{cc}, p_{nc}\}$; $neg\_ack_i \leftarrow true$ **endif**;
(10) **if** $(i = nc)$ **then**
(11) **wait until** $((tot\_prevc_i \geq n - f) \wedge ((pos\_ack_i) \vee (tot\_bnextc_i \geq n - f)))$ **endif**;
(12) **endif**

Task $T2$:
(13) **upon** *receipt* **of** $\text{EST}(r, est, ts)$ **such that** $r_i - n < r \leq r_i$:
% If $r \leq r_i - n$: the message is discarded. If $r_i < r$: the message is delayed %
(14) **if** $(ts > ts_i)$ **then** $ts_i \leftarrow ts; est_i \leftarrow est$ **endif**;
(15) **if** $(r = ts)$ **then**
(16) $pos_i[r] \leftarrow pos_i[r] + 1;$
(17) **if** $(pos_i[r] \geq f + 1)$ **then** $\forall j \neq i$: *send* $\text{DECISION}(est)$ *to* p_j; return(est) **endif**;
(18) **if** $((r = r_i) \wedge \neg pos\_ack_i \wedge \neg neg\_ack_i)$ **then**
(19) **let** X_i **be** a locally defined set including $\{p_{cc}, p_{nc}\}$;
(20) *send* $\text{EST}(r, est, r)$ *to* X_i; $pos\_ack_i \leftarrow true$ **endif**;
(21) **endif**;
(22) **if** $(r = bnextc_i)$ **then** $tot\_bnextc_i \leftarrow tot\_bnextc_i + 1$ **endif**;
(23) **if** $(r = prevc_i)$ **then** $tot\_prevc_i \leftarrow tot\_prevc_i + 1$ **endif**

Task $T3$:
(24) **upon** *receipt* **of** $\text{DECISION}(est)$ **from** p_k: $\forall j \neq i, k$: *send* $\text{DECISION}(est)$ *to* p_j; return(est)

Figure 1. The *Sliding Round Window* $\diamond\mathcal{S}$**-Based Consensus Protocol**

consider a round $r_i = r$. The behavior of $T1$ depends on the process that executes it:

• If p_i is the coordinator of r (then, $i = cc$), it resets its window-related local variables (line 5), increases ts_i to r (line 6), and "announces" it started this round by broadcasting the message $\text{EST}(r, est_i, r)$ (line 7).

• If p_i is not the coordinator of r, it waits (line 8) until it suspects p_{cc} or it has got the estimate value sent by p_{cc}.

A process decides, as a direct consequence of its participation to a round r, when it has received enough positive acknowledgments concerning the estimate value broadcast by the coordinator of r. When $r_i = r$, p_i is allowed to start participating in the round $r + 1$ as soon as it has sent an acknowledgment concerning the estimate value sent by p_{cc}.

The sending of positive acknowledgments (by $p_i \neq p_{cc}$) is the job of the task $T2$ which sets pos_ack_i to $true$. The task $T1$ manages only the sending of positive acknowledgments by the round coordinator ($p_i = p_{cc}$) and the sending of negative acknowledgments by all processes (lines 8-9). Such an acknowledgment is sent by p_i when it suspects p_{cc} and has not previously sent a positive acknowledgment (test of line 8). Moreover, as the reception of a negative acknowledgment does not allow a process to decide, it is sufficient to send such a negative ack only to p_{cc} and p_{nc} to prevent them from possibly blocking forever.

• If, additionally, while $r_i = r$, p_i is such that $i = nc$ (i.e., p_i will be the coordinator of $r + 1$), the protocol forces p_i to wait until the following two conditions are satisfied.
- (1) The first condition is used to prevent p_i from indefinitely progressing from round to round. The occurrence of such an indefinite progression would violate the termination property. This prevention is done by synchronizing p_i with respect to the last round it has coordinated. More precisely, a process p_i $(= p_{nc})$ is allowed to progress from the round r to the round $r + 1$ it will coordinate, only if it is no longer possible for it to decide as far as the last round it has previously coordinated (namely, the round $prevc_i$) is concerned. This is expressed by the predicate $(tot\_prevc_i \geq n - f)$ (line 11)[2].
- (2) It is possible that a value v be decided by one or several processes while p_i (=p_{nc}) is not aware of it and progresses from the round r (=$bnextc_i$) to the round $r+1$. In that case,

[2]If $r + 1$ is the first round coordinated by p_i, the initialization of tot_prevc_i (line 1) makes true this predicate.

in order the agreement property not be violated, p_i has to start $r+1$ with its estimate equal to v. This is ensured with another synchronization that requires p_i (=p_{nc}) to wait for the predicate $(pos\_ack_i \vee (tot\_bnextc_i \geq n-f))$ (line 11). (For an extended discussion on these two synchronization conditions, see Section 4.3.)

Task $T2$**.** This task manages the reception of $\text{EST}(r, est, ts)$ messages received by p_i. First of all, the messages that lie outside the window are either definitely discarded ($r \leq r_i - n$) or momentarily delayed ($r > r_i$).

First (line 14), if the $\text{EST}(r, est, ts)$ message carries a more recent estimate value (this is known by comparing ts_i and ts), the estimate est_i and its timestamp ts_i are updated. In that way, the pair (est_i, ts_i) defines the most recent view that p_i has on the estimate of the decision value.

Then, if $r = ts$, p_i concludes that the EST message carries a positive acknowledgment concerning the value initially broadcast by the coordinator of r. Hence, $pos_i[r]$ is incremented (line 16). Additionally, if p_i has received enough positive acknowledgments (i.e., $f+1$) related to r, it decides on the value broadcast by that coordinator (line 17).

Moreover, if r is the last round in which p_i is engaged ($r = r_i$), then it participates in this round by acknowledging positively the value broadcast by the round coordinator (line 20). This participation consists in forwarding $\text{EST}(r, est, ts)$ to a set X_i of processes including at least p_{cc} and p_{nc} (cc and nc refer to the round $r_i = r$). This allows p_{cc} (resp. p_{nc}) to update its local variable tot_prevc_i (resp. tot_bnextc_i) and hence prevents it (resp. p_{nc}) from blocking at line 11 when it executes the round $r+n-1$ (resp. r).

Finally, if r is equal to $bnextc_i$ (resp. $prevc_i$), the counter tot_bnextc_i (resp. tot_prevc_i) is appropriately incremented.

For a round r, it is important to note the difference in the role played by a positive acknowledgment (sent at line 20) and a negative acknowledgment (sent at line 9). The aim of the latter is only to prevent p_{cc} and p_{nc} from blocking forever at the rounds indicated above. In addition to this prevention, the aim of the former is to allow processes to decide "directly" during r (a process decides directly during r, when it has received $(f+1)$ positive acknowledgments related to r, line 17). Hence, from the point of view of p_i, the processes of X_i are the ones that p_i potentially allows to test the decision predicate during r (line 17).

It is also important to note that for a same round r, two processes p_i and p_j can have different sets X_i and X_j provided that $\{p_{cc}, p_{nc}\} \subseteq (X_i \cap X_j)$. As noted in the Introduction, this versatility allows the processes to dynamically adapt the number of messages they send according to their current perception of the network load (see Section 4).

Task $T3$**.** To prevent a process from blocking forever (i.e., waiting for a value from a process that has already decided), a process that decides (line 17) uses a reliable broadcast to disseminate its decision value (message DECISION). The task $T3$ implements this reliable broadcast. (This feature is not related to the management of round numbers. It is shared by all the previously proposed $\diamond\mathcal{S}$-based consensus protocols).

Structure of a Round. Similarly to other $\diamond\mathcal{S}$-based consensus protocols, each round of the proposed protocol is made up of two phases. During the first phase of a round r, its coordinator broadcasts an $\text{EST}(r, est, r)$ message, and each other process p_i waits until it suspects it or receives the $\text{EST}(r, est, r)$ message. Then, during the second phase, each process p_i sends an acknowledgment. If the acknowledgment is positive, p_i sends it to a dynamically defined set of processes X_i (see Section 4.1). If it is negative, it sends it only to the coordinators of r and $r+1$. Then, a process that receives positive acknowledgments counts them, and decides directly (i.e., at line 17) if it has received enough of them.

The originality of the protocol comes from the fact that it allows processes to be simultaneously involved in several rounds. To our knowledge, it is the first protocol to provide this feature. This allows processes to be less constrained by synchronization than in other protocols.

4 Discussion

4.1 Definition of the X_i Sets

As we have seen (line 19, Figure 1), at each round r, each process p_i defines a set X_i including the processes it allows to test the decision predicate (test of line 17): this is the set of processes to which p_i sends an $\text{EST}(r, -, r)$ message (positive ack). As required by the proof, the definition of such a set X_i has to satisfy a single constraint, namely $\{p_{cc}, p_{nc}\} \subseteq X_i$ (where p_{cc} and p_{nc} are the coordinators of r and $r+1$, respectively). The constraint $p_{cc} \in X_i$ is required to ensure the *protocol termination*, while the constraint $p_{nc} \in X_i$ is required to ensure the *protocol agreement*. Hence, the protocol actually defines a family of protocols, each member of the family being characterized by a particular definition of the sets X_i. As noted in the Introduction, this versatility can be used by the processes to adapt the message load according to their current perception of the network load.

As previously indicated, the definition of the sets X_i for a round r actually defines the pattern of messages exchanged during r. Two definitions for the sets X_i are particularly interesting:

$i.\ \forall\, r:\ \forall\, i:\ X_i = \{p_{cc}, p_{nc}\}$. In that case each process ($\neq p_{cc}$) sends only two EST messages during each round. Hence, the number of EST messages exchanged during a round is $3n-4$ ($n-1$ sent by the coordinator, $2(n-2)$ by the processes not in X_i to the processes of X_i, and 1 from p_{nc} to p_{cc}). This case corresponds to a centralized message exchange pattern.

$ii.\ \forall\, r:\ \forall\, i:\ X_i = \Pi$ (all processes). In that case, no process is a priori prevented from testing the decision predicate (line 17). The number of EST messages exchanged during a round is then $n(n-1)$. This case corresponds to a fully decentralized message exchange pattern.

Interestingly, when the instantiation (i) is used, the message exchange pattern of each round becomes close to the one of [1]. Differently, when the instantiation (ii) is used, the message exchange pattern of each round is close to the one of [8]. This shows that, as far the message exchange pattern of a round is concerned, the centralized pattern of [1] and the distributed pattern of [8] are not antagonistic, but are merely two instantiations of a more generic pattern ([6] provides an in-depth discussion on this point).

4.2 The Protocol wrt Other Protocols

This section briefly sketches a comparison of the proposed protocol (HMMR) with the protocols proposed in [1] (CT), [4] (HR), [8] (MR), and [9] (SC). This comparison does not pretend to be exhaustive.

Communication steps. For each protocol we count the minimal number of communication steps for a process to decide. More precisely, we count the minimal number of communication steps (sequence of messages) that occur before a process is allowed to test the decision predicate at line 17 (and decides accordingly). This means that the "indirect" decisions (due to DECISION messages) are not taken into account.

Let us consider a system of n processes: $p_1, \ldots, p_n$. By assumption $f < n/2$. We consider the worst case $f = \lfloor (n-1)/2 \rfloor$.

In order not to be bothered by the behavior of the underlying failure detector, we assume it offers a very good quality of service, i.e., it does not make mistakes (this is consistent with the fact that in many systems, failure detectors can be tuned to very seldom make mistakes). Similarly, not to be dependent on the particular timing of crash occurrences, we assume that a process that is not correct has crashed before becoming a round coordinator. We consider the following failure patterns:

FP_0: all processes are correct.
FP_1: all processes but p_1 (first coordinator) are correct.
FP_f: all processes but $p_1, p_2, \ldots$ and p_f (first f coordinators) are correct.

It is important to note that, although failures are rare in practice, they do occur. This means that the failure patterns most often encountered are the ones where the first coordinator is correct, and if the first coordinator has crashed, the failure patterns where the second coordinator is correct. When considering the previous failure patterns, this means that the most often encountered are FP_0 and FP_1.

For each failure pattern, Table 1 indicates the minimal number of communication steps a protocol requires for a process to (directly) decide. In all failure patterns, CT requires 3 communication steps. This is due to the fact that, when the underlying failure detector makes no mistake, CT requires (1) 3 steps for the round coordinator to directly decide during a round if it has not crashed, and (2) no communication step 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).

	FP_0	FP_1	...	FP_f
CT	3	3	...	3
HR/MR	2	4	...	$f+2$
SC	2	6	...	$2f+2$
HMMR	2	3	...	3

Table 1. #Steps with Reliable FD

If the current coordinator has not crashed (and is not suspected), each of HR, MR and SC requires 2 communication steps for any process to directly decide (two phases of a single round). But, if the current coordinator has crashed, SC requires 2 communication steps for processes to progress to the next round, while HR and MR require only one.

The last line of Table 1 shows that HMMR nicely combines the advantages of the other protocols. It requires 2 rounds in FP_0 (as HR, HR and SC) and 3 rounds in the other failure patterns (as CT).

Direct decision. Let us now consider the number (x) of processes that are a priori allowed to decide directly during a round (i.e., at line 17).

CT	HR	MR	SC	HMMR
$x=1$	$x=n$	$x=n$	$x=n$	$2 \le x \le n$
$y=1$	$y=1$	$y=1$	$y=1$	$y=n$

Table 2. Two Features of $\Diamond\mathcal{S}$-Based Protocols

CT allows a single process to decide during a round (namely, the round coordinator), hence $x = 1$. The other processes decides at least one communication step later (either "directly" at a higher round or "indirectly" when they receive a DECISION message). In HR, MR and SC no process is a priori prevented from testing the decision predicate, hence here x is always equal to n. Differently from CT, HR, MR and SC, the number of processes that are allowed to decide directly during a round is not fixed once for all in HMMR. This set of processes actually depends on the definition of the sets X_i (see the previous section). The first line of Table 2 summarizes this discussion. The fact that HMMR does not fix a priori the value of x makes it very versatile, and allows its users to instantiate it in a way suited to their context.

The second line of Table 2 indicates for each protocol the maximal number of rounds $(y \geq 1)$ in which a process can be simultaneously involved. The greater the number y, the less synchronized the protocol in the sense that a process that attained round r is not prevented to continue to participate in the previous rounds r' $(r > r' > r - y)$ and can consequently be allowed to decide due to its participation in any of those rounds. It follows from the second line of Table 2 that HMMR requires less synchronization than the other protocols and can fit better to an asynchronous context.

4.3 Size of the Window

Reducing hidden synchronization. The protocol given Figure 1 provides processes with a window of size n. More precisely, the current window of p_i is made up of the rounds $r_i - (n-1), \ldots, r_i$. Hence, a message EST(r, est, ts) that arrives at p_i while $r_i < r$ is delayed until $r_i = r$. This delay constitutes a hidden synchronization that can slow down the protocol. This drawback can be eliminated by forcing a process to process an EST(r, est, ts) message even when $r_i < r$. The round window management becomes then similar to the sequence number window management encountered in communication protocols.

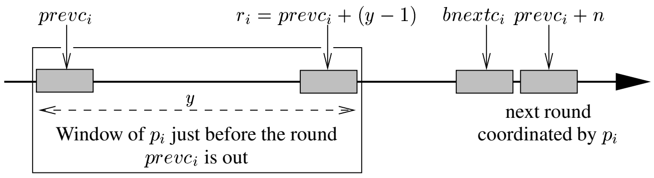

Figure 2. p_i**'s window When** $r_i = prevc_i + (y-1)$

Defining the size of the window. We show here a very slight modification of the protocol that allows to consider any value y $(1 \leq y \leq n)$ as the size of the window.

Let us first remind the motivation for the synchronization of line 11. A process p_i coordinates the rounds $i + kn$ for all $k \geq 0$. Let us consider a round r such that p_i will be the coordinator of $r+1$ (hence, $p_i = p_{nc}$ and $r = bnextc_i$). We have seen in Section 3.3 (and also in the proof of the protocol) that the synchronization of the line 11 concerns only p_{nc}. As indicated, this synchronization is twofold. It ensures that (1) the agreement property cannot be violated (role of the predicate $(pos\_ack_i \vee (tot\_bnextc_i \geq n - f))$, and (2) the termination property cannot be violated (by guaranteeing that a process does not indefinitely progress). This second guarantee is realized by forcing p_i $(= p_{nc})$ to wait until, from its point of view, there is no possibility to decide as far as the last round it has coordinated (namely, the round $prevc_i$) is concerned (role of the condition $(tot\_prevc_i \geq n - f)$).

A closer look at line 11 and the proof show that, for a process p_i, the second synchronization is actually done with respect the last round p_i has coordinated when this round is about to exit from the current window. So, let us consider that we have a window whose size is equal to y $(1 \leq y \leq n)$. In this context, the previous observation means that when p_i is in the round r_i such that $r_i = prevc_i + (y-1)$ (see Figure 2) it has to wait for the condition $(tot\_prevc_i \geq n - f)$ to become true. The protocol can accordingly be modified to consider a window size equal to y. We only have to replace the lines 10 and 11 with the following lines:

if $(r_i = prevc_i + (y-1))$ **then**
  wait until $(tot\_prevc_i \geq n - f))$ **endif**;
if $(i = nc)$ **then**
  wait until $(pos\_ack_i \vee (tot\_bnextc_i \geq n - f))$ **endif**.

Let us note that when $y = n$, the equality $r_i = prevc_i + (y-1)$ means that $r_i = bnextc_i$ and $p_i = p_{nc}$. This explains why in the original protocol the two waiting conditions are merged in a single wait statement. (Let us note that, when $y = 1$, the equality $r_i = prevc_i + (y-1)$ implies that p_i is the coordinator of r_i, hence is p_{cc}. If additionally, $X_i = \{p_{cc}, p_{nc}\}$, the behavior of the modified protocol becomes close to the $\diamond\mathcal{S}$-based Chandra-Toueg's protocol [1]. If $X_i = \Pi$, it becomes close to the $\diamond\mathcal{S}$-based Mostefaoui-Raynal's protocol [8].)

5 Proof of the Uniform Agreement Property

Notation. The symbol "$-$" in a field of an EST message means that the value of the corresponding field is irrelevant with respect to the lemma/theorem in which it appears.

Lemma 1 *For any process p_i, we have $ts_i \leq r_i$. Moreover for any* EST(r, est, ts) *message we have $ts \leq r$.*

Proof The proof is by induction on the round number r_i. It is left to the reader. $\square_{Lemma\ 1}$

Lemma 2 *If there is an* $\text{EST}(-, v, ts)$ *message, with* $ts \neq 0$*, sent/received by a process, then the coordinator of the round* ts *sent an* $\text{EST}(ts, v, ts)$ *message.*

Proof Due to the network reliability, any message that is received by a process has previously been sent. So, we only consider message sendings. Let $\text{EST}(r, v, ts)$ the EST message sent by a process p_i; it is sent at line 7, 9 or 20. We examine each case separately.

- Case 1: $\text{EST}(r, v, ts)$ is sent at line 7. In that case the message is sent by the coordinator of r. The lemma follows from line 6 which sets ts_i to r_i (hence, $ts = ts_i = r_i = r$).
- Case 2: $\text{EST}(r, v, ts)$ is sent at line 20. In that case, we have $ts = r$ (protocol text), and p_i received at line 13 the same $\text{EST}(r, v, r)$ message (due to line 15). Let p_j be its sender. If p_j is the coordinator of r, the lemma immediately follows.

  So, let us consider that p_j is not the coordinator of r. In that case, p_j received $\text{EST}(r, v, r)$ from some process p_k. Due to the fact that the number of processes is bounded and the fact that a process does not send several times an EST message with the same round number (see the booleans pos_ack_i and neg_ack_i), there is a process that sent the first $\text{EST}(r, v, r)$ message, and this message has necessarily been sent at line 7 or 9. We claim that a message $\text{EST}(r, v, r)$ cannot be sent at line 9. It follows that the first $\text{EST}(r, v, r)$ message has been sent at line 7. Hence, it has been sent by the coordinator of r.

  Proof of the claim. Let us consider a process p_i that executes line 9 while $r_i = r$. Note that pos_ack_i and neg_ack_i are false.

  Just before p_i entered r (line 2), we had (Lemma 1) $ts_i \leq r_i = r - 1$. Then p_i increased r_i (line 3) which became equal to r. There are two cases.

  - If, while waiting in the wait statement (line 8) before executing line 9, p_i is not required to execute the updates of line 14, ts_i is not modified, and consequently, we have $ts_i < r_i$ when p_i executes line 9.
  - Let us now consider the case where p_i executes the updates of line 14 while waiting at line 8. This means that p_i has received an $\text{EST}(r', v', ts')$ message. This message is such that $ts' \leq r'$ (Lemma 1), and $r' \leq r_i$ (reception test). Hence, $ts_i = ts' \leq r' \leq r_i$ after the update. We show that we have more, namely, $ts' \leq r' < r_i$.

    Let us first observe that p_i has not executed line 20 (otherwise, due to pos_ack_i it would be prevented to send a message at line 9). ¿From the values of the boolean flags (which are false), the fact that line 20 has not been executed, $r' \leq r_i$, and the test at line 18, we conclude that $r' < r_i$. Consequently, $ts_i < r_i$ when p_i executes line 9.

  It follows that any $\text{EST}(r, v, ts)$ message sent at line 9 is such that $ts < r$. *End of the proof of the claim.*
- Case 3: $\text{EST}(r, v, ts)$ is sent at line 9. Let us consider the update of ts_i which set it to ts. This is the last update of ts_i before the sending of $\text{EST}(r, v, ts)$ at line 9. It has been done at line 6 or 14.
  - Case 3.1: The update $ts_i \leftarrow ts$ has been done at line 6.

    In that case, this update has been done during the round ts which is coordinated by p_i. We conclude from line 6, that during this round, we had $ts = ts_i = r_i$. Moreover, est_i, which has now the value v (because, by assumption, p_i is now sending $\text{EST}(r, v, ts)$ at line 9) has not been modified since the round ts. This is because the only line where est_i can be modified is line 14 which modifies both est_i and ts_i; but precisely we are considering the case where the last update of ts_i has not been done at line 14. It follows that est_i was equal to v when p_i sent $\text{EST}(ts, est_i, ts)$ at line 7 (while it was coordinating the round ts), which proves the lemma.
  - Case 3.2: The update $ts_i \leftarrow ts$ has been done at line 14.

    In that case, p_i received an $\text{EST}(-, v, ts)$ message (line 13), and did the updates: $ts_i \leftarrow ts$; $est_i \leftarrow v$ (line 14). Let p_j be the sender of the $\text{EST}(-, v, ts)$ message. The sending of this message by p_j falls into one of the following cases: Case 1, Case 2, Case 3.1 or Case 3.2. If it falls into Case 1, Case 2 or Case 3.1, the lemma follows from the previous case analysis.

    Let us consider the last case: the sending of $\text{EST}(-, v, ts)$ by p_j falls into Case 3.2. Hence, p_j received an $\text{EST}(-, v, ts)$ message (line 13) from a process p_k and so on, defining a chain of processes p_i, p_j, p_k, etc. This chain of processes is finite because there is a bounded number of processes and for a value ts, a process executes the updates of line 14 at most once. Consequently, this process chain ends with a process that sent $\text{EST}(-, v, ts)$ in a situation corresponding to Case 1, Case 2 or Case 3.1. $\square_{Lemma\ 2}$

Lemma 3 *Let* $r \geq 1$ *be the first round during which* $(f + 1)$ *processes send* $\text{EST}(r, v, r)$*. Then*[3]*: (i): No process decides directly before r (i.e., at line 17). (ii):* $\forall\ r' \geq r$*, if the coordinator of* r' *sends an* $\text{EST}(r', -, -)$ *message, then this message is* $\text{EST}(r', v, r')$*.*

Proof Proof of (i). To decide at line 17 during a round r, a process has to receive EST messages carrying a timestamp equal to r from $(f + 1)$ processes. The impossibility for a process to decide at line 17 before r is a direct consequence of the very definition of r which is the "first round during which ...".

The proof of (ii) is by induction on the round number.

- Base case: $r' = r$. The proof of (ii) follows directly from Lemma 2 that states if a process sends $\text{EST}(r, v, r)$, then the coordinator of r sent the same message.
- Induction case: $r' > r$. Let us assume that the lemma holds for any round k such that $r \leq k \leq r'$. We show it is true for $r' + 1$.

[3] Using the traditional terminology, this lemma states the condition that entails the "locking" of a value (v). Once a value is locked, no other value can be decided.

Due to the induction assumption, we have: for all k such that $r \le k \le r'$: if the coordinator of k sent $\text{EST}(k,-,-)$ then this message is $\text{EST}(k,v,k)$.

Let us consider the round $r'+1$. This round is coordinated by some process p_j. Let us consider the last update $ts_j \leftarrow ts$ that p_j executed before entering $r'+1$. We consider p_j when it enters $r'+1$ and show that $est_j = v$ at that time. Let us examine the following three cases.

- $r \le ts_j = ts \le r'$ and the last update of ts_j ($ts_j \leftarrow ts$) has been done at line 14.

  In that case, p_j received $\text{EST}(-,w,ts)$, and did the updates $ts_j \leftarrow ts$ and $est_j \leftarrow w$ at line 14. Due to Lemma 2, the coordinator of the round ts sent $\text{EST}(ts,w,ts)$. As $r \le ts \le r'$, due to the induction assumption, we have $w = v$.

- $r \le ts_j = ts \le r'$ and the last update of ts_j ($ts_j \leftarrow ts$) has been done at line 6.

  In that case, p_j was the coordinator of the round ts. As $r \le ts \le r'$ we conclude from the induction assumption that p_j sent $\text{EST}(ts,v,ts)$. Hence, we had $est_j = v$ when p_j started the round ts. As the last update of ts_j has not been done at line 14, it follows that p_j has not modified est_j since it sent $\text{EST}(ts,v,ts)$. Hence, we still have $est_j = v$ when p_j enters r'.

- $ts_j = ts < r$. We show that this case is impossible.

  Before entering $r_j = r'+1$, p_j executed line 11 while $r_j = r'$. Hence the predicate $((pos\_ack_j) \vee (tot\_bnextc_j \ge n-f))$ was true just before p_j updated r_j from r' to $r'+1$. We consider two cases.
  (1) If pos_ack_j was true at the end of $r_j = r'$, then p_j sent $\text{EST}(r',w,r')$ at line 20 during r'. This means that p_j received the same $\text{EST}(r',w,r')$ message at line 13. From $r' > r$ and the case assumption we get $r' > r > ts = ts_j$.

  As $r' > ts = ts_j$, p_j updated ts_j to r' (line 14) after it received $\text{EST}(r',w,r')$. A contradiction, as in that case the last update to ts_j has been done with the timestamp r' which is $> ts$.
  (2) Let us now assume that pos_ack_j was false just before p_j increased r_j from r' to $r'+1$. As p_j is the coordinator of $r'+1$, let us first observe that $bnextc_j = r'$. As pos_ack_j was false, the condition $(tot\_bnextc_j \ge n-f)$ was necessarily true when p_j progressed to $r'+1$. Hence, $tot\_bnextc_j + (f+1) > n$, from which we conclude that there is a process p_x such that:
  - p_x belongs to the set of $(f+1)$ processes that sent $\text{EST}(r,v,r)$ during r (lemma assumption),
  - p_x belongs to the set made up of the tot_bnextc_j processes that sent a $\text{EST}(r',-,-)$ message received by p_j during r' $(> r)$. Let us remind that, as $bnextc_j = r'$, p_j counted these messages at line 22. Let $\text{EST}(r',-,ts')$ be the message actually sent by p_x. As p_x sent $\text{EST}(r,v,r)$ and then $\text{EST}(r',-,ts')$, we have $r \le ts'$ (timestamps do not decrease). As now (when p_j enters $r'+1$) we have $ts_j = ts$, we conclude that $ts_j \le ts$ when p_j received $\text{EST}(r',-,ts')$. From $ts_j \le ts$ and $ts < r \le ts'$ we conclude that p_j updated ts_j to ts' (line 14) after it received $\text{EST}(r',-,ts')$. A contradiction, as in that case the last update to ts_j has been done with the timestamp ts' which is $> ts$.

$\square_{Lemma\ 3}$

Theorem 1 *No two processes decide differently.*

Proof If a process decides a value at line 24, then this value has been decided by another process at line 17. So, we only consider values decided at line 17.

Let p_i and p_j be two processes that decide $v1$ and $v2$ during the rounds $r1$ and $r2$, respectively. As p_i (resp. p_j) decides during $r1$ (resp. $r2$), it has received $\text{EST}(r1,v1,r1)$ (resp. $\text{EST}(r2,v2,r2)$) during $r1$ (resp. $r2$). Due to Lemma 2, the coordinator of $r1$ (resp. $r2$) has sent $\text{EST}(r1,v1,r1)$ (resp. $\text{PROP}(r2,v2,r2)$). Let us assume $r1 \le r2$.

Let r be the round characterized in Lemma 3. We conclude from Lemma 3 that the messages $\text{EST}(r,v,r)$, $\text{EST}(r1,v1,r1)$ and $\text{EST}(r2,v2,r2)$ are such that $r \le r1 \le r2$ and $v = v1 = v2$. $\square_{Theorem\ 1}$

References

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

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

[3] 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.

[4] Hurfin M. and Raynal M., A Simple and Fast Asynchronous Consensus Protocol Based on a Weak Failure Detector. *Distributed Computing*, 12(4):209-223, 1999.

[5] Hurfin M., Macêdo R., Mostefaoui A. and Raynal M., A Sliding Round Window $\diamond\mathcal{S}$-Based Consensus Protocol. *RR #1304*, IRISA, Université de Rennes, February 2000. http://www.irisa.fr/EXTERNE/bibli/pi/1304/1304.html.

[6] Hurfin M., Mostefaoui A. and Raynal M., A Comprehensive Approach for Failure Detector-Based Consensus Protocols. *RR #1303*, IRISA, Université de Rennes, February 2000. http://www.irisa.fr/EXTERNE/bibli/pi/1303/1303.html.

[7] Lamport L., The Part-Time Parliament. *ACM Transactions on Computer Systems*, 16(2):133-169, 1998.

[8] Mostefaoui A. and Raynal M., Solving Consensus Using Chandra-Toueg's Unreliable Failure Detectors: a General Quorum-Based Approach. *13th Int. Symp. on Distributed Computing (DISC'99)*, LNCS 1693:49-63, Bratislava, 1999.

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