PUBLICATION INTERNE
N° 1372

LEADER-BASED CONSENSUS

ACHOUR MOSTEFAOUI , MICHEL RAYNAL

ISSN 1166-8687

IRISA
CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 02 99 84 71 00 – Fax : (33) 02 99 84 71 71
http://www.irisa.fr

Leader-Based Consensus

Achour Mostefaoui , Michel Raynal

Thème 1 — Réseaux et systèmes
Projet Adp

Publication interne n ° 1372 — Décembre 2000 — 13 pages

Abstract: It is now well recognized that consensus is a fundamental problem one has to solve to implement reliable applications on top of unreliable asynchronous distributed systems prone to failures. It has been shown that this problem cannot be solved if the underlying asynchronous system does not satisfy additional assumptions. This paper presents a new consensus protocol based on a leader oracle (denoted Ω in the litterature). Although this protocol uses asynchronous rounds, it is not based on the rotating coordinator paradigm. As a consequence, it does not suffer from drawbacks inherent to $\diamond S$-based consensus protocols that explicitly use this paradigm. As Ω and $\diamond S$ are equivalent, the proposed protocol does not require assumptions stronger or weaker than the ones abstracted in $\diamond S$. Hence, it also requires $f < n/2$ (where n is the number of processes and f an upper bound on the number of processes that may crash).

From a design point of view, the proposed protocol is surprisingly simple. From an efficiency point of view, it allows the processes to agree in a single round when the oracle provides the processes with the same leader (a common case in practice).

It is also shown that the time and message costs of the protocol can be reduced when $f < n/3$. Moreover, when, in addition to the leader oracle, the system is equipped with a random oracle, the proposed protocol can be extended to provide a hybrid consensus protocol at no additional message cost.

Key-words: Asynchronous Distributed System, Consensus, Crash Failure, Fault-Tolerance, Leader Oracle.

(Résumé : tsvp)

email: {mostefaoui,raynal}@irisa.fr

Centre National de la Recherche Scientifique
(UPRESSA 6074) Université de Rennes 1 – Insa de Rennes

Institut National de Recherche en Informatique
et en Automatique – unité de recherche de Rennes

Un protocole de consensus fondé sur un leader

Résumé : Cet article présente un protocole de consensus fondé sur un oracle qui fournit le nom d'un leader. Cet oracle dénoté Ω dans la littérature est équivalent (en terme de classes de problèmes qu'ils peuvent résoudre) au détecteur de défaillances $\diamond S$. En conséquence la contrainte $f < n/2$ reste de rigueur (n représente le nombre de processus du système et f le nombre maximum de défaillances possibles).

Ce nouveau protocole procède par itérations successives comme les protocoles fondés sur un détecteur de défaillances $\diamond S$ à la différence que celui-ci ne considère pas une coordination tournante. Par rapport aux protocoles utilisant une coordination tournante ce protocole nécessite une phase supplémentaire pour chaque pas d'itération. Si le leader considéré par les différents processus est d'emblée le même, la convergence est obtenue au bout d'un seul pas d'itération. De plus un leader élu par les modules de l'oracle peut servir pour plusieurs exécutions du consensus.

Mots clés : Communication par messages, consensus, panne franche, système réparti asynchrone, tolérance aux défaillances, oracle.

1 Introduction

The *Consensus* problem lies at the heart of a lot of distributed computing problems one has to solve when designing reliable applications on top of unreliable distributed asynchronous systems. Those problems define the family of *agreement* problems, and consensus can be seen as their "greatest common subproblem". The consensus problem can be informally stated as follows. Each process proposes a value, and has to decide on a value (termination) such that (1) there is a single decided value (agreement), and (2) the decided value is a proposed value (validity). This apparently simple problem has actually no solution in an asynchronous systems as soon as (even only) one process may crash. This impossibility result, due to Fisher, Lynch and Paterson [9] (and known under the name FLP) is one of the most famous impossibility result in the domain of distributed computing.

To circumvent this impossibility, two main approaches have been investigated. One of them consists in abandoning the determinism requirement of the protocol, and allowing the processes to query an oracle providing them with random values [3, 21]. Another approach consists in enriching the system with synchrony assumptions until they allow the problem to be solved [7]. This approach has been abstracted in the notion of unreliable failure detectors [4, 5]. A failure detector can be seen as a distributed oracle that gives (possibly incorrect) hints about which processes have crashed so far. Of course to be useful, a failure detector has to satisfy some properties. An *hybrid* approach, that consists in combining the use of a failure detector oracle with a random oracle has also been investigated in [1, 20].

Chandra and Toueg have proposed eight classes of failure detectors, each being defined by two properties, namely a completeness property (that is on the actual detection of crashes) and an accuracy property (that limits the mistakes the corresponding failure detector can make). Among these classes, the class denoted $\diamond S$ is particularly interesting as it has been shown to be the weakest class of failure detectors that allows to solve the consensus problem [5] (when a majority of processes do not crash).

The class $\diamond S$ is defined by the following two properties. Strong Completeness: every crashed processes is eventually suspected by every correct process (a correct process is a process that does not crash). Eventual Weak Accuracy: there is a time after which there is a correct process that is never suspected by the correct processes. Several consensus protocols designed for asynchronous distributed systems equipped with a failure detector of the class $\diamond S$ have been designed [4, 12, 18, 24]. All these protocols are based on the rotating coordinator paradigm. Processes proceed in asynchronous rounds and each round is managed by a predetermined process. Basically, during a round, the corresponding coordinator tries to impose its estimate of the decision value as the decided value. The completeness property is used to prevent processes from indefinitely waiting a message from a crashed coordinator. The accuracy property is used to ensure that there is eventually a round whose coordinator will not be suspected, thereby ensuring termination. The main difficulty these protocols have to solve is to guarantee that the consensus agreement is never violated despite asynchrony, process crashes and erroneous suspicions.

As observed in [15, 16], the previous $\diamond S$-based consensus protocols have an intrinsic inconvenience due to the fact that they are based on the rotating coordinator paradigm. More precisely, in order to decide, the processes can be forced to proceed until the round r whose coordinator is not suspected, even if this process is not suspected from the very beginning. Hence the idea [16] to design a consensus protocol that is not coordinator-based but leader-based. More precisely, a leader oracle provides each process with the name of a leader. Such a leader election capability can be used as follows: the processes can still proceed in consecutive asynchronous rounds, but during each round they trust the current leader instead of a round coordinator. The crucial difference lies in the fact that a round coordinator is defined according to the round number, while a leader is defined by the oracle independently of the round in which the oracle is invoked. Hence, all the consensus executions that occur in a "good" period (i.e., when the leader oracle provides the processes with the same correct leader) allow the processes to decide during their very first round.

A leader oracle-based consensus protocol is described in [16]. It is based on a new failure detector denoted $\diamond C$ defined by two properties (a completeness property and an eventual consistent accuracy property). This protocol can be seen as an adaptation of the $\diamond S$-based consensus protocol designed by Chandra and Toueg [4] to the $\diamond C$ leader oracle. It is shown in [15] that, from a computational point of view, the classes $\diamond S$ and $\diamond C$ are equivalent. It seems important to note that, although not expressed in the failure detector-based formalism, to our knowledge, the first leader-based consensus protocol that has been proposed is Lamport's Part-Time Parliament protocol ($\Pi\alpha\xi o\varsigma$) [13].

This paper presents an oracle based consensus protocol that does not use the rotating coordinator paradigm, and consequently does not suffer from the associated inherent drawback. The oracle it uses is the failure detector Ω introduced in [5] (where it is also shown that Ω and $\diamond S$ are equivalent). Basically, Ω stipulates that there is a time after which all the correct processes trust the same correct process. This is actually a leader oracle that can be used by the processes, independently of the round number they are currently executing. In addition to the very early decision it allows (when the leader oracle behaves correctly) the resulting protocol has interesting properties. The consensus executions that occur when the leader oracle behaves correctly cost a single round each (a round costs three sequential phases). Moreover, the protocol has a particularly simple design.

The paper is composed of six sections. Section 2 presents the computation model and the leader oracle Ω. Then, Section 3 presents the consensus protocol. Section 4 proves it is correct. Section 5 discusses some features of the protocol. It is shown that the number of phases per round can be reduced from 3 to 2, when $f < n/3$. Considering that the underlying system provides a random oracle, it is also shown that the proposed leader-based protocol can be very easily made hybrid, thereby benefiting from the best of both worlds (leader oracle or random oracle). Finally, Section 6 concludes the paper.

2 Distributed Computation Model and the Consensus Problem

2.1 Asynchronous Distributed Systems with Crash Failures

The computation model follows the one described in [4, 9, 17]. We consider a system consisting of a finite set Π of $n > 1$ processes, namely, $\Pi = \{p_1, \ldots, p_n\}$. 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. A *faulty* process is one that is not correct. Let f denote the maximum number of processes that may crash. We assume $f < n/2$, i.e., a majority of processes is correct (see Section 2.3). Processes communicate and synchronize by sending and receiving messages through channels. Every pair of processes is connected by a channel. Channels are assumed to be reliable (Section 5 considers weaker assumptions). There is no assumption about the relative speed of processes nor on message transfer delays: the system is *asynchronous*.

2.2 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 the same value v, that has to be one of the proposed values. More precisely, the *Consensus problem* is defined by three safety properties (Validity, Integrity and Uniform Agreement) and a Termination Property [4, 9]:

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

2.3 Leader Oracle

As the consensus problem cannot be solved in an asynchronous distributed system [9], we consider that the system is equipped with an appropriate oracle. As announced in the Introduction, we consider here the class Ω of failure detector oracles introduced in [5]. We call *leader oracles* its failure detectors. This class is defined as follows. The processes can invoke a function that we call leader. This function outputs a process identity and satisfies the following property:

- Eventual Leadership: There is a time t and a correct process p such that, after t, every invocation of leader by a correct process returns p.

A failure detector of this class actually provides the processes with an eventual leader election capability. But, let us notice that there is no knowledge of when the leader is elected. This means that several leaders can coexist during an arbitrarily long period of time, and there is no way for the processes to learn this "confusing" period is over [13].

From a computational point of view, the class Ω of leader oracles, the class $\diamond S$ of unreliable failure detectors introduced in [4], and the class $\diamond C$ introduced in [15, 16] have been shown to be equivalent [5, 6, 15]. It follows that the consensus problem can be solved in any asynchronous distributed system equipped with any of them, provided that $f < n/2$.

3 The Leader Oracle-Based Consensus Protocol

3.1 Underlying Principle

The principle that underlies the protocol is rather simple. The protocol requires that each process p_i manages an estimate est_i of the decision value (est_i is initialized to v_i, the value p_i proposes). Observing that, if all the processes have the same estimate value v, it is easy to make them agree on it (without the help of an oracle), the protocol strives to provide the processes with the same estimate.

To this end, it uses the leader oracle in the following way: each process waits for an estimate value from the process it considers as the leader. If the processes consider the same leader, they get the same estimate v from it and decide v. Due to eventual leadership property, this will occur if, repeatedly, the processes invoke the leader oracle. The problem that has to be solved is that the consensus agreement property be not violated if, while processes have different leaders, some of them decide. Hence, the main issue the protocol has to solve is to guarantee the safety property will never be violated. The way this is realized is explained in the next section.

3.2 Description of the Protocol

The protocol is described in Figure 1. It assumes $f < n/2$. A process p_i starts a consensus execution by invoking $\mathsf{Consensus}(v_i)$. The function $\mathsf{Consensus}()$ is made up of two concurrent tasks, $T1$ (the main task) and $T2$. The statement **stop** terminates the task that invokes it. The statement *return*(v) terminates the consensus execution (as far as p_i is concerned) and returns the decided value v to p_i.

For the processes to eventually have the same estimate value, they proceed in consecutive asynchronous rounds[1]. Each process p_i has a local variable r_i defining the round it is currently involved in. Each round is made of three phases during which the processes exchange messages (namely, PHASE1(), PHASE2() and PHASE3() messages). More precisely:

- The aim of the first phase (lines 4-6) is to try to have the processes have the same estimate value. This is done with the help of the leader oracle. A process p_i waits for a message from the process it currently considers as the leader (line 5) and adopts as estimate the value carried by this message (line 6).

  As the leadership property of the leader oracle is only eventual, it is possible that the leader considered by p_i be different from the one considered by other processes (several processes having different leaders). To cope with these problems and prevent processes from deadlocking, the protocol first forces each process p_i to broadcast its current estimate (line 4); so, if other processes consider p_i as their leader, they will not wait forever. Second, the protocol allows a process p_i to find a leader (invocation of the leader function at line 5) that sent a PHASE1 message carrying an estimate value.

- The aim of the second phase (lines 7-10) is to allow the processes to know if there is a majority value v. This majority requirement is to guarantee that the agreement will not be violated by the processes that will decide during this round.

  To attain this goal, processes simply exchange the values of their current estimates. If any, the majority value (v) is kept in an auxiliary variable aux_i; otherwise aux_i is set to a default value ($\bot$). Notice that, when processes start this phase, there is a single or no majority value. So, at the end of this phase, $aux_i \neq \bot$ and $aux_j \neq \bot$ means that $aux_i = aux_j = v$ the majority value.

- The aim of the third phase (lines 11-15) is to allow a process to know if it can decide without compromising the agreement property in the case some processes decide during next rounds.

  To attain this aim, the processes exchange the value of their aux_i local variables. If a process p_i receives an $aux_i = v \neq \bot$ value, it adopts it as current estimate (line 13). If it receives enough (i.e., at least $f+1$) $aux_i = v \neq \bot$ values, it also decides on v (line 15).

  Let us note that if a process decides, it decides on a value that was a majority estimate value at the end of the first round. As a single value can be a majority value, a single value v ($\neq \bot$) can be decided during a round. Moreover, as any process p_i receives $(n-f)$ PHASE3 messages, we can conclude that if p_i does not decide while p_j does because it received $(f+1)$ PHASE3(r,v) messages, p_i adopts v as estimate (from $(n-f)+(f+1) > n$, we can conclude that p_i received at least one PHASE3(r,v) message). Hence, the agreement property is ensured despite the fact processes can decide at different rounds.

[1]As explained in the Introduction, although there are rounds, the protocol is not based on the rotating coordinator paradigm.

It is easy to see that when all the processes that have not initially crashed have the same initial estimate value, the decision is obtained during the first round, whatever the behavior of the leader oracle[2]. Interestingly, executions where a majority of processes propose the same initial value allow the processes to decide in a single round.

This presentation has implicitly assumed that a process decides at line 15 when it invokes *R_Broadcast* DECISION(est_i). Actually, this reliable broadcast primitive is used to disseminate the decided value. The subsection that follows motivates the use of this primitive.

Function Consensus(v_i)

Task $T1$:
(1) $r_i \leftarrow 0$; $est_i \leftarrow v_i$;
(2) **while** *true* **do** % Sequence of rounds %
(3) $r_i \leftarrow r_i + 1$;

——————— Phase 1 of round r_i ———————
(4) *broadcast* PHASE1(r_i, est_i);
(5) **wait until** ($\exists\ \ell$ s.t. leader=$\ell \wedge$ PHASE1(r_i, v) received from p_ℓ);
(6) $est_i \leftarrow v$;

——————— Phase 2 of round r_i ———————
(7) *broadcast* PHASE2(r_i, est_i);
(8) **wait until** (PHASE2(r_i, est) messages received from $(n-f)$ processes);
(9) **if** (the same value v has been received from a majority of processes)
(10) **then** $aux_i \leftarrow v$ **else** $aux_i \leftarrow \perp$ **endif**;
% $(aux_i = v \neq \perp) \Rightarrow$ (v is maj. in $\{est_k \mid 1 \leq k \leq n\}$ at the end of phase 1 %
% Hence: $((aux_i \neq \perp) \wedge (aux_j \neq \perp)) \Rightarrow (aux_i = aux_j = v)$ %

——————— Phase 3 of round r_i ———————
(11) *broadcast* PHASE3(r_i, aux_i);
(12) **wait until** (PHASE3(r_i, aux) messages received from $(n-f)$ processes);
(13) **if** ($\exists$ PHASE3(r_i, aux) with $aux = v \neq \perp$) **then** $est_i \leftarrow v$ **endif;**
(14) **if** ($(f+1)$ PHASE3(r_i, aux) messages are such that $aux \neq \perp$)
(15) **then** *R_Broadcast* DECISION(est_i); **stop** $T1$ **endif**
(16) **endwhile**

Task $T2$: **upon** *R_Delivery* **of** DECISION(v): *return*(v) % terminates the consensus %

Figure 1: Leader Oracle-Based Consensus ($f < n/2$)

3.3 Reliable Broadcast

The protocol uses a *Reliable Broadcast* primitive as a subroutine (line 15 and task $T2$). As a deciding process stops participating in the sequence of rounds, and all processes do not necessarily decide during the same round, it is possible that the termination of some processes blocks other processes that proceed to the next round. By disseminating the decided value, the reliable broadcast prevents such deadlock occurrences.

[2] Let us notice that the existing $\Diamond S$-based consensus protocols do not enjoy this nice property. Erroneous suspicions can force processes to progress to the next round even when the processes do propose the same initial value.

The *Reliable Broadcast* primitive allows a message to be reliably sent to processes[3]. "Reliably" means here that if the message is delivered by a process, then it is delivered by all correct processes. Formally, Reliable Broadcast is defined by two primitives [10]: *R_Broadcast* and *R_Delivery*. The semantics of these primitives is defined by three properties, namely, Validity, Integrity and Termination. When a process p executes *R_Broadcast*(m) (resp. *R_Delivery*(m)) we say that it R-broadcasts m (resp. R-delivers m). We assume that all the messages are different.

• Validity: If a process R-delivers m, then some process has R-broadcast m. (No spurious messages.)

• Integrity: A process R-delivers a message m at most once. (No duplication.)

• Termination: If (1) a correct process R-broadcasts m, or if (2) a process R-delivers m, then all correct processes R-deliver m. (No message R-broadcast by a correct process or R-delivered by a process is missed by a correct process.) This property defines the situations in which the reliable broadcast must terminate (i.e., when a message m must eventually be R-delivered).

Implementations of reliable broadcast can easily be designed for asynchronous systems. A very simple implementation that works in fully connected networks is the following one: when a process receives a message m for the first time, it first forwards m to the other processes, and only then delivers m [10]. According to the underlying network topology, more efficient implementations can be designed [23]. Interestingly, the reliable broadcast protocols described in [2, 22] works with fair lossy channels.

4 Proof of the Protocol

The proof of the validity and integrity properties are easy and left to the reader. As shown by the proof of the termination and agreement properties, the protocol requires $n > 2f$.

4.1 Termination

Lemma 1 *No correct process blocks forever in a round.*

Proof If a process decides, then due to the reliable broadcast of the DECISION message, all correct processes decide. Hence, they do not block forever during a round. Hence, let us assume that no process decides. The proof is by contradiction. Let r be the smallest round number in which a correct process p_i blocks forever. So, p_i blocks at line 5, 8 or 12. We show this is impossible.

Let us first consider line 5. Due to eventual leadership property of the leader oracle, the invocations of the leader function at line 5 eventually return a correct process name. Let p_j be this process. As p_j is correct, it sent a PHASE1$(r, -)$ message at line 4. It follows that p_i cannot block forever at line 5.

The fact that p_i blocks neither at line 8 nor at line 12 is a direct consequence of the assumption on the maximal number of faulty processes. There are at least $(n - f)$ correct processes that broadcast PHASE2$(r_i, -)$ and PHASE3$(r_i, -)$ messages. $\Box_{Lemma\ 1}$

Theorem 1 *Every correct process decides.*

Proof If a correct process decides a value v, due to the reliable broadcast primitive all the correct processes that have not yet decided deliver the DECISION(v) message and decide. So, let us assume (by contradiction) that no correct process decides.

[3]The primitive we present here is sometimes called *Uniform Reliable Broadcast* [10].

Due to Lemma 1 (no correct process is blocked forever in a round), and the eventual leadership property of the leader oracle, there is a time t and a correct process p_x such that, there is a round r during which there are only correct processes and each correct process p_i has leader$=x$. This means that during r the correct processes have the same leader and trust it. It follows that: (1) during the first phase of r, they get its estimate value $v = est_x$. (2) During the second phase of r, they exchange this value, and consequently their aux_i variables are set to v. (3) During the third phase of r, they again exchange only v. From $n > 2f$, we conclude $(n - f) \geq (f + 1)$, which means that each p_i receives $(f + 1)$ PHASE3(r, v) messages and consequently decides. $\Box_{Theorem\ 1}$

4.2 Uniform Agreement

Theorem 2 *No two processes decide different values.*

Proof Let r be the first round during which a process decides ("decide v during r" means "execute line 15 with $est_i = v$ during r"). Let v be the value it decides. We show that (1) the processes that decide during r decide v, and (2) all estimates are equal to v at the end of r (hence, no other value can be decided in a next round).

First of all, let us observe that at the end of the second phase of r, any aux_i variable is equal either to $\perp$ or to the value v that was the majority estimate value (if any) at the end of the first phase of r. This means that $((aux_i \neq \perp) \wedge (aux_j \neq \perp)) \Rightarrow (aux_i = aux_j = v)$. As $\perp$ cannot be decided, it follows that, if two processes decide during r, they decide the same non-$\perp$ value.

Assuming that processes decide v during r, we now prove that the estimate values of the processes that progress to $(r + 1)$ are equal to v at the end of r. Let p_i be any process that decides v, and let p_j be any process that proceeds to $r + 1$. As $(f + 1)$ (number of PHASE3 messages that allowed p_i to decide during r) + $(n - f)$ (number of PHASE3 messages received by p_j during r) $> n$, it follows that at least one PHASE3(r, v) message received by p_i has also been received by p_j. Consequently p_j executed line 13, and updated est_j to v. $\Box_{Theorem\ 2}$

5 Discussion

5.1 Time and Message Cost

A round costs $O(n^2)$ messages. The time performance of the protocol depends on the behavior of the leader oracle. In the best case, the oracle provides the processes with the same correct leader from the beginning. In that case, a single round allows the processes to decide.

If the leader oracle behaves "perfectly" during a long period of time, it allows each execution of the consensus protocol that occurs during that period to be expedited in one round. As shown by the proof, the simultaneous presence of several leaders can impede the protocol progress, but cannot cause inconsistencies. In that sense, the proposed leader oracle-based protocol has some similarities with the $\Pi\alpha\xi o\varsigma$ algorithm [13, 14].

5.2 Case $f < n/3$

Although failures do occur, they are rare in practice. This observation shows that the assumption "less than a third of the processes can crash" is not really constraining. This section shows that the number of phases per round can be reduced from 3 to 2 when $f < n/3$. This stronger assumption actually allows to merge the second and the third phase into a single one.

The resulting protocol is shown in Figure 2. As $(n-f) > n/2$, let us first observe that, during a round r, no two processes can decide different values at lines 10-11. Let us now consider the case where, during a round r, a process p_i decides a value v at lines 10-11. We conclude that no more than f PHASE2_3 messages carry a value different from v. As any process p_j that executes the second phase receives at least $(n-f)$ PHASE2_3 messages, it follows that at least $(n-f)-f$ of those PHASE2_3 messages carry the value v, forcing p_j to adopt v as estimate at line 9. Hence, as soon as a process decides v, all estimate values are set to v. Agreement cannot be violated. The proof of the termination property is the same as before.

Function Consensus(v_i)

Task $T1$:
(1) $r_i \leftarrow 0$; $est_i \leftarrow v_i$;
(2) **while** *true* **do** % Sequence of rounds %
(3) $r_i \leftarrow r_i + 1$;
——————— Phase 1 of round r_i ———————
(4) *broadcast* PHASE1(r_i, est_i);
(5) **wait until** ($\exists$ ℓ s.t. leader=ℓ $\wedge$ PHASE1(r_i, v) received from p_ℓ);
(6) $est_i \leftarrow v$;
——————— Phase 2_3 of round r_i ———————
(7) *broadcast* PHASE2_3(r_i, est_i);
(8) **wait until** (PHASE2_3(r_i, est) messages received from $(n-f)$ processes);
(9) **if** (at least $(n-2f)$ PHASE2_3(r_i, est) carry the same $est = v$) **then** $est_i \leftarrow v$ **endif**;
(10) **if** (the $(n-f)$ PHASE2_3(r_i, est) carry the same $est = v$)
(11) **then** *R_Broadcast* DECISION(est_i); **stop** $T1$ **endif**
(12) **endwhile**

Task $T2$: **upon** *R_Delivery* **of** DECISION(v): *return*(v) % terminates the consensus %

Figure 2: A Leader-Based Consensus Protocol for $f < n/3$

5.3 A Hybrid Protocol

When the eventual leadership property is not satisfied by the current leader oracle implementation, it is possible that correct processes obtain oracle outputs such that several processes have different leaders. This situation can hinder the correct processes from deciding. We show that this bad behavior can be prevented by enriching the system with an additional oracle.

Without loss of generality, this section assumes that the consensus is *binary*, i.e., a process can propose one out of two values, namely 0 or 1. (The result of this section can be extended to the case of multivalued consensus by using the techniques developed in [8, 19].)

Let us consider that, in addition to the leader oracle, the underlying asynchronous distributed system is equipped with a *random oracle*. Such an oracle consists of a module per process p_i that provides p_i with a function called random. When it works correctly, the module attached to a process behaves as a random number generator: it outputs a random bit each time random is queried. Moreover, for simplicity, we assume a uniform distribution: each value has probability 1/2 to be returned when p_i invokes random.

The aim of a hybrid protocol is to benefit from the best of both worlds. It terminates deterministically if the underlying leader oracle satisfies the eventual leadership property (whatever the behavior of the the random oracle). When the leader oracle does not satisfy the eventual leadership property, the protocol terminates with probability 1 if the random oracle works correctly, and eventually each call to the function leader outputs a correct process name (let us note that, in that case, processes can have different leaders).

The motivation that underlies the design of a hybrid protocol is the following one. Let the "good" periods be the periods when the leadership property is satisfied by the modules implementing the leader oracle, and the "bad" periods be the other periods. ("Good/bad" periods are sometimes called "stable/unstable" periods.) A consensus execution that occurs in a good period always terminates. A bad period does not prevent a consensus execution to terminate, but the termination is only probabilistically guaranteed. Hence, a hybrid protocol (1) guarantees termination of consensus executions when the period is good, and (2) does its best effort (with the help of the random oracle) to ensure termination when the period is bad.

It appears that the addition of a single **if ... then ... endif** statement to the protocol described in Figure 1 makes it hybrid. More specifically, the hybrid protocol is obtained by adding the following line after line 15:

(15′) **if** (all the PHASE3(r_i, aux) msgs rec. are such that $aux = \bot$) **then** $est_i \leftarrow$ random **endif**

Let us remark that if a process executes $est_i \leftarrow$ random, then both values (0 and 1) have been proposed. Moreover (due to the relation $(n-f)+(f+1)>n$ used at the end of Theorem 2) it is easy to see that, during a round r, it is impossible for a process p_i to execute line 15, while another process p_j would execute line 15′. The proof of the hybrid protocol is left to the reader (it is close to the proof described in [20])[4].

Interestingly, the structure of the hybrid protocol is the same as the structure of the original protocol: the number of phases per round, and the message exchange pattern are left unchanged. Let us note that, as described in [21], the sharing of a long sequence of random bits can help expedite the decision.

5.4 Crash/Recovery Model

The protocol could be adapted to work in the crash/recovery model. Let us note that such a model also assumes message losses. In that case, the leader oracle definition has to appropriately redefined to take into account recoveries. The modification of the proposed protocol would be close to the one we followed in [11] to adapt to the crash/recovery model the $\diamond\mathcal{S}$-based consensus protocol described in [12].

6 Conclusion

This paper presented a new consensus protocol based on a leader oracle (Ω). Although it uses asynchronous rounds, this protocol does not suffer from drawbacks inherent to $\diamond S$-based consensus protocols that explicitly use the rotating coordinator paradigm.

The proposed protocol has several advantages. First it has a surprising design simplicity. Moreover, it allows the processes to agree in a single round (which is made up of three communication steps) when the oracle provides the processes with the same leader (a common case in practice).

[4]When the eventual leadership property is not satisfied, the property "Eventually any call to leader returns a correct process" is necessary to ensure that a process does not block forever at line 5, the correctness of the random oracle ensuring the probabilistic termination.

Last but not least, it has been shown that, when the system is equipped with a leader oracle and a random oracle, the proposed protocol can very easily be extended to provide a hybrid consensus protocol, thereby benefiting from the best of both worlds. It has also been shown that it can be made more time and message efficient when less than a third of the processes may crash.

References

[1] Aguilera M.K. and Toueg S., Failure Detection and Randomization: a Hybrid Approach to Solve Consensus. *SIAM Journal of Computing*, 28(3):890-903, 1998.

[2] Aguilera M.K., Toueg S. and Deianov B., Revisiting the Weakest Failure Detector for Uniform Reliable Broadcast. *Proc. 13th Int. Symposium on DIStributed Computing (DISC'99)*, Springer -Verlag LNCS #1693, pp. 21-34, Bratislava (Slovaquia), 1999.

[3] Ben-Or M., Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols. *2nd ACM Symposium on Principles of Distributed Computing, (PODC'83)*, Montréal (CA), pp. 27-30, 1983.

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

[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.

[6] Chu F., Reducing Ω to $\diamond W$. *Information Processing Letters*, 67(6):289-293, 1998.

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

[8] Ezhilchelvan P., Mostefaoui A. and Raynal M., Randomized Multivalued Consensus. *Research Report* #1320, IRISA, Université de Rennes (France), July 2000.

[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] Hadzilacos V. and Toueg S., Reliable Broadcast and Related Problems. In *Distributed Systems*, ACM Press (S. Mullender Ed.), New-York, pp. 97-145, 1993.

[11] Hurfin M., Mostefaoui A. and Raynal M., Consensus in Asynchronous Systems Where Processes Can Crash and Recover. *Proc. 17th IEEE Symposium on Reliable Distributed Systems*, Purdue University (IN), pp. 280-286, October 1998.

[12] 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.

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

[14] Lampson B.W., How to Build a Highly Available System Using Consensus. *Proc. 10th Int. Workshop on Distributed Algorithms*, Springer-Verlag LNCS #1051, pp. 1-17, (Ö. Babaoğlu and K. Marzullo Eds), Bologna (Italy), October 1996.

[15] Larrea M., Efficient Algorithms to Implement Failure Detector and Solve Consensus in Distributed Systems. *Ph. D. Thesis*, Universidad del País Vasco, San Sebastián (Spain), 98 pages, October 2000.

[16] Larrea M., Fernández A. and Arévalo S., Eventually Consistent Failure Detectors. *Brief announcement, 14th Int. Symposium on Distributed Computing*, Toledo (Spain), October 2000. (Tech Report FIM/110.1/DLSIIS/2000, Technical University of Madrid.)

[17] Lynch N., Distributed Algorithms. *Morgan Kaufmann Pub.*, San Francisco (CA), 872 pages, 1996.

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

[19] Mostefaoui A., Raynal M. and Tronel F., From Binary Consensus to Multivalued Consensus in Asynchronous Message Passing Systems. *Information Processing Letters*, 73:207-212, 2000.

[20] Mostefaoui A., Raynal M. and Tronel F., The Best of Both Worlds: A Hybrid Approach to Solve Consensus. *Proc. Int. Conference on Dependable Systems and Networks (DSN'00, formerly FTCS)*, IEEE Computer Society Press, pp. 513-5222, New-York City, June 2000.

[21] Rabin M., Randomized Byzantine Generals. *Proc. 24th IEEE Symposium on Foundations of Computer Science (FOCS'83)*, pp. 116-124, Los Alamitos (CA), 1983.

[22] Raynal M., Quiescent Uniform Reliable Broadcast as an Introductory Survey to Failure Detector Oracles. *Research Report #1356*, IRISA, Université de Rennes, 13 pages, October 2000.

[23] Rodrigues L. and Verìssimo P., Topology-Aware Algorithms for Large Scale Communication. *Advances in Dist. Systems*, Springer-Verlag LNCS #1752, pp.1217-1256, 2000.

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