CONSENSUS IN SYNCHRONOUS SYSTEMS:
A CONCISE GUIDED TOUR

MICHEL RAYNAL
Consensus in synchronous systems: 
a concise guided tour

Michel Raynal*

Thème 1 — Réseaux et systèmes
Projet ADP

Publication interne n°1467 — Juillet 2002 — 12 pages

Abstract: This paper is on consensus protocols for synchronous systems where processes can commit crash failures, omission failures or Byzantine failures. It presents and revisits consensus protocols coping with such failures in an increasing order of difficulty. The paper can be seen as a short tutorial whose aim is to make the reader familiar with synchrony assumptions, different definitions of the consensus problem, and a hierarchy of process failure models. An important concern of the paper lies in simplicity. In addition to the survey flavor of the paper, several results that are presented are new, among which the ones concerning the omission failure model.

Key-words: Byzantine Failure, Consensus, Crash Failure, Early Decision, Indistinguishability, Omission Failure, Message Passing System, Synchronous Distributed System, Uniform Agreement, Weak Validity.

(Résumé : tsup)
Consensus synchrone : une visite guidée

Résumé : Ce rapport présente revisite le problème du consensus dans système un distribué synchrone. Il peut être vu comme un tutoriel dont le mot-clé serait “simplicité”. La partie sur les fautes par omission est originale.

Mots clés : Accord uniforme, consensus, crash de processus, décision au plus tôt, faute byzantine, faute d’omission, système distribué synchrone, validité faible.
1 Introduction

Consensus is one of the most fundamental problems one has to solve when designing and implementing reliable applications on top of distributed systems that can exhibit unreliable behaviors. This problem, first identified and formalized by Lamport, Shostack and Pease [20, 25] in the context of synchronous systems prone to Byzantine process failures, can be informally stated as follows: each process proposes a value and the non-faulty processes have to decide on the same value, that has to be one of the proposed value [5, 13]. Basically, consensus aims at providing processes with a meaningful global consistency mechanism. Each process proposes a “view” encoded in a value, and the consensus ensures that exactly one of these views is adopted by (at least) the non-faulty processes, from which they can consistently progress. Consistency is ensured here by the unicity of the decided value, its meaningfulness by the fact it has been proposed by a process.

One of the most important distributed computing results concerns the impossibility to design a deterministic solution to the consensus problem in asynchronous distributed systems prone to process crash failures [13]. On the contrary, the consensus problem can be solved in synchronous systems, and several distributed computing textbooks (e.g., [2, 21]) devote chapters to synchronous consensus protocols. But these books concentrate mainly on lower bound results, and consider only the two extreme points of the process failure spectrum, namely crash failures and Byzantine failures. Moreover, very few papers addresses the case of omission failures.

This paper surveys consensus protocols for synchronous systems. Its aim is to emphasize their design principles, and provide a clear insight on the additional assumptions a consensus protocol requires to go from crash failure to omission failures, and then to Byzantine failures. In that sense, in addition to the investigation concerning the omission failure mode, the paper can also be considered as an introductory tutorial to synchronous consensus protocols. An important point of the paper lies in its concern for simplicity. It is made up of six sections. Section 2 presents the computation model and the consensus problem. Then, Sections 3, 4 and 5 consider the case of crash failures, omission failures, and Byzantine failures, respectively. Among other issues, it is shown in Section 4 that a majority of processes must be correct to get Uniform Consensus in presence of omission failures. Finally, Section 6 provides a few concluding remarks.

2 Computation Model and the Consensus Problem

2.1 Synchronous System

The system model consists of a finite set of processes, namely, \( \Pi = \{p_1, \ldots, p_n\} \), that communicate and synchronize by sending and receiving messages through channels. Every pair of processes \( p_i \) and \( p_j \) is connected by a channel denoted \( (p_i, p_j) \).

The system is synchronous. This means that each of its executions consists of a sequence of rounds. Those are identified by the successive integers 1, 2, etc. For the processes, the current round number appears as a global variable \( r \) that they can read, and whose progress is managed by the underlying system. A round is made up of three consecutive phases:

- A send phase in which each process sends messages.
- A receive phase in which each process receives messages.
  The fundamental property of the synchronous model lies in the fact that a message sent by a process \( p_i \) to a process \( p_j \) at round \( r \), is received by \( p_j \) at the same round \( r \).
- A computation phase during which each process processes the messages it received during that round and executes local computation.

The underlying communication system is assumed to be failure-free: there is no creation, alteration, loss or duplication of message.

2.2 Process Failure Model

A process is faulty during an execution if its behavior deviates from that prescribed by its algorithm, otherwise it is correct. We assume that \( f \) is an upper bound for the number of faulty processes.

\[ \text{This problem originated in the development of on-board aircraft control systems (which have to be reliable!).} \]

\[ \text{This “view” can be its current view of the global state, a proposal for an action that has to be done, etc. It is defined according to the problem that has to be solved.} \]
A failure model defines how a faulty process can deviate from its algorithm [17]. We consider here the following three failure models:

- **Crash** failure. A faulty process stops prematurely its execution. After it has crashed, a process does nothing.
  
  Let us observe that if a process crashes in the middle of a sending phase, only a subset of the messages it was supposed to send might actually be sent.

- **Omission** failure. A faulty process crashes or omits to send messages it was supposed to send (send omission) or omits to receive messages it was supposed to receive (receive omission).
  
  Let us observe that a faulty process can omit to send or receive messages during some rounds and later crash³.

- **Byzantine** failure. A faulty process can exhibit any behavior whatsoever (e.g., start in an arbitrary state, send arbitrary messages, perform arbitrary state transition, etc.).

  It easy to see that these three failure models are of increasing “severity”: Crash ∼ Omission < Byzantine in the sense that any protocol that solves a problem in a failure model A, solves it in a less severe failure model B, i.e., such that $B \prec A$ [17].

### 2.3 The Consensus Problem

The consensus problem has been informally stated in the Introduction: every process $p_i$ proposes a value $v_i$ and all correct processes have to decide on some value $v$, in relation to the set of proposed values. More precisely, the Consensus problem is defined by the following three properties [5, 13]:

- **Termination**: Every correct process eventually decides.
- **Validity**: If a process decides $v$, then $v$ was proposed by some process.
- **Agreement**: No two correct processes decide different values.

  Let us observe that the agreement property is only on correct processes: it allows a faulty process to decide differently from the correct processes. Such a property can be too weak for some applications that require a single decision whatever the deciding process be faulty or correct. **Uniform agreement** is a strengthened form of agreement that prevents such scenarios. More precisely, Uniform Consensus is defined by the previous Termination and Validity requirements plus the following Agreement property:

- **Uniform Agreement**: No two (correct or not) processes decide different values.

  Let us observe that a Byzantine process can decide any value. So it is impossible for a consensus protocol designed for the Byzantine failure model to guarantee Uniform Agreement or Validity. So, a reasonable specification of Byzantine Consensus includes Termination, Agreement, and the following Validity property:

- **Weak Validity**: If all the correct processes propose the same value $v$, then $v$ is the value decided by a correct process.

Table 1 (at the end of the paper) summarizes the previous definitions.

### 3 Consensus in Systems with Crash Failures

#### 3.1 A Consensus Protocol

**Principle** Figure 1 presents a variant of the well-known flood-set consensus protocol [2, 21]. Each process $p_i$ invokes the function $\text{Consensus}(v_i)$ where $v_i$ is the value it proposes. It terminates it with the invocation of the statement $\text{return}(\cdot)$ that provides it with the decided value. $\text{Consensus}(\cdot)$ is made up of $f + 1$ rounds whose aim is to fill in an array $V(j)$ in such a way that $V(j)$ contains the value proposed by $p_j$. The flood-set strategy used to attain this goal is particularly simple: it consists for a process to send during a round (say, $r$), all the
new information it got during the previous round \( (r - 1) \). The management of the other local variables of a process \( p_i \) is self-explanatory (\( \perp \) denotes a default value that cannot be proposed).

**Theorem 1** The protocol described in Figure 1 solves the Uniform Consensus problem in a synchronous system where up to \( f \) processes can crash.

**Proof** The proof that this protocol satisfies the Termination and Validity properties is easy. Termination follows from the bounded number of rounds. Validity follows from the fact that \( \forall (i,k) \): \( V_i[k] \) is \( v_k \) or \( \perp \).

Proving that the protocol satisfies Uniform Agreement consists in showing that any two processes \( p_i \) and \( p_j \) that decide, have the same vector \( V_i = V_j \) just before deciding at the end of the round \( f + 1 \). Let \( V_i[k] = v_k \neq \perp \) at the end of the last round. The process \( p_i \) received \( v_k \) for the first time at some round \( r \) (consider \( r = 0 \) if \( k = i \)). Let us first notice that, as \( p_i \) and \( p_j \) decide, they execute all the rounds until \( f + 1 \). There are two cases:

- \( r < f + 1 \). In that case, \( p_i \) put \( (v_i,k) \) in \( \text{New}_i \) at round \( r \) and sent it to \( p_j \) at round \( r + 1 \). Hence, \( p_j \) got \( (v_i,k) \) during the round \( r + 1 \leq f + 1 \) (at the latest).

- \( r = f + 1 \). In that case, \( (v_i,k) \) was forwarded through a chain of processes from \( p_k \) to \( p_i \) such that no process appears more than once in this chain (this follows from the fact that only new information is forwarded by a process). As at most \( f \) processes crash, it follows than this chain includes a correct process \( p_k \) that got the pair \( (v_i,k) \) at some round \( r' < f + 1 \) and forwarded it during the round \( r' + 1 \leq f + 1 \). Hence, \( p_j \) got \( (v_i,k) \) during \( r' + 1 \leq f + 1 \) (at the latest).

**Cost** The time complexity is trivially \( f + 1 \) rounds. Let \( b \) be the number of bits required to encode a proposed value. So, a pair \( (v_i,k) \) requires \( b + \log_2 n \) bits. Let us observe that a process \( p_i \) sends a given pair \( (v_i,k) \) at most once to each of the \( n - 1 \) other processes. Consequently, the bit complexity is upper bounded by \#Processes \( \times \) #Pairs \( \times \) size of a pair \( \times \) #Dest of a pair, i.e., \( n^2(n - 1)(b + \log_2 n) \).

**A Simple Improvement** It is often the case in practice that the set \( V \) of values that can be proposed is totally ordered by some relation (that we denote \( < \)). In that case, the array \( V_i \) can be reduced to a single variable containing the minimum value that has so far been seen by \( p_i \). That value will be the value decided by \( p_i \) at the end of the last round. The protocol simplifies as described in Figure 2. Its bit communication complexity is upper bounded by \( n(n - 1)b \times \min(f + 1, |V|) \). In the case of binary consensus (i.e., when \( V \) has only two elements) this reduces to \( 2n(n - 1) \). (An easy additional improvement consists for \( p_i \) not to send a given value to the processes from which it received that value.)

### 3.2 Early Decision

It has been shown that \( f + 1 \) is a lower on the number of rounds (e.g., [1, 12, 21, 22]), which means that, whatever the protocol, it is always possible to have an execution that requires \( f + 1 \) rounds in presence of up to \( f \) crashes. However, it might be supposed that in executions where the actual number of failures \( (t) \) is small compared to the number of allowable failures \( (f) \), the number of rounds could be correspondingly small. Such

---

\(^3\) A send (receive) omission failure actually models a failure of the output (input) buffer of a process. A buffer overflow is a typical example of such a failure.

\(^4\) This variant combines the flood-set strategy with “wave propagation/extinction” principles developed in [4, 19] for distributed graph algorithms.
an early deciding property [10] is not provided by the previous protocols that always require \( f + 1 \) rounds (even when there is no failure!).

We present here a simple observation that allows to design an early deciding protocol from the protocol described in Figure 1. Its number of rounds is upper bounded by \( \min(t + 2, f + 1) \) which has been proved to be optimal [10]. Let us first replace line (a) in Figure 1 by the following two lines (a1) and (a2):

(a1) \( \text{foreach } j \text{ send } (New_i) \text{ to } p_j \);

(a2) \( \text{let } R_i(r) = \{ \text{proc. from which msg have been received during } r \}\)

So, now each process \( p_i \) broadcasts a \( New \) message at each round, even if its content is the empty set. Let \( UP^r \) be the set of processes that have not crashed at the beginning of the round \( r \). We trivially have \( R_i(r) \subseteq UP^r \). Moreover, for generality purpose, let us define \( R_i(0) = \Pi \). As an immediate consequence of the synchrony assumption and the fact that crashes are definitive, we have the following monotone decreasing sequence:

\[
\Pi = R_i(0) \supseteq \cdots \supseteq R_i(r - 1) \supseteq UP^r \supseteq R_i(r) \supseteq \cdots
\]

Let us consider a process \( p_k \) such that, during a round \( r \), we have \( R_i(r - 1) = R_i(r) \). From the previous set containments, we conclude that \( R_i(r - 1) = UP^r = R_i(r) \), i.e., the processes of \( R_i(r) \) are exactly the processes that were not crashed when the system progressed from \( r - 1 \) to \( r \). It then follows that, at the end of \( r \), \( p_i \) knows all the information that can be known, i.e., all the pairs \( (v, k) \) that were known by the non-crashed processes when the round \( r \) started. Consequently, \( p_i \) cannot learn new pairs in the future [8]. So, if \( R_i(r - 1) = R_i(r) \) at the end of \( r \), (1) \( p_i \) knows all the pairs that can be known, but (2) it does not know if the other processes \( p_j \) know the same. So, (due to 1) \( p_i \) can decide, but (due to 2) it has first to send the new pairs it got during \( r \) to the other processes. Hence, \( p_k \) proceeds to round \( r + 1 \) (to send the new pairs), but before sets a flag to decide at \( r + 1 \). This observation brings us to add the following statement just after line \( \beta \) of Figure 1 to get an early deciding protocol:

\[
\text{if } ( R_i(r - 1) = R_i(r) \land r < f + 1 ) \text{ then set a flag to direct } p_k \text{ to decide at the end of } r + 1 \text{ endif}
\]

The resulting protocol solves the Uniform Consensus problem. The reader can check that it has the following nice “early deciding” properties. (1) It terminates in two rounds when no process crashes. (2) It terminates in three rounds when the processes that are faulty have initially crashed\(^5\). (3) More generally, it executes at most \( \min(t + 2, f + 1) \) rounds.

**Other improvements** Let us refine the protocol of Figure 2 with the previous improvement and the suppression of the variable \( prev_i \). Let \( V_i(r) \) be the value of the local variable \( V_i \) after it has been updated during the round \( r \). Let us observe that, when \( R_i(r - 1) = R_i(r) \land V_i(r - 1) = V_i(r) \), \( p_k \) can conclude that all the other processes know the value \( V_i = v \). So, in that case, it is not necessary for \( p_k \) to transmit \( v \) to the other processes as they know that value. So, the round behavior of the previous protocol based on ordered values can be refined as follows:

\[^5\text{Such processes are sometimes called initially dead processes. The situation of item (2) is when some processes have crashed, but no process crashes during the execution of the protocol. Furthermore, let us also notice that if the set of initially crashed processes is initially known by each process (i.e., } R_i(0) \text{ contains exactly those processes) the protocol terminates in two rounds if no process crashes during its execution.} \]
from the correct processes. That idea defines what is done in the additional round (namely, \( f + 2 \)) then set a flag to direct \( p_k \) to decide at \( r + 1 \)

endcase
endif

Further improvements can be obtained by considering the case \( |R_i(r)| = n \) or the case \( |R_i(1)| = (n - f) \). In the former case, \( p_k \) knows that it knows all the proposed values [18]. In the latter case, \( p_k \) knows that, from now on, no process will crash [7].

4 Consensus in Systems with Omission Failures

4.1 Consensus vs Uniform Consensus

The protocols described in Figures 1 and 2 solve the Consensus problem even when processes commit failures more severe than crashes, namely, omission failures. This follows from the fact that, as the correct processes neither crash nor omit to send or receive messages, they have the same \( V_i \) at the end of the round \( f + 1 \).

Unfortunately, although these protocols solve the Uniform Consensus problem in presence of any number of crash failures, they do not solve it in presence of even a single process that fails by omission. This can be easily deduced from the following observation. Let \( p_k \) be a faulty process that permanently omits to receive messages. When, \( p_k \) executes the last round \( (f + 1) \) of any of the previous protocols, it knows only the value it has proposed, and consequently decides it, whatever the value decided by the correct processes.

4.2 A Uniform Consensus Protocol

Fortunately, the inability of the previous protocols to address the Uniformity attribute of the Agreement property (namely, be it faulty or not, a deciding process cannot decide differently from the correct processes), is not inherent to the Uniform Consensus problem. We show here that the simple addition of an extra round to the protocol described in Figure 2 allows it to ensure Uniformity of the Agreement property despite omission failures. This protocol is described in Figure 3.

The idea is rather simple: it consists in preventing a faulty process to decide without knowing the value decided by the correct processes. That idea defines what is done in the additional round (namely, \( f + 2 \)):
- Each process \( p_k \) first sends to all the processes (including itself) the value it is about to decide (namely \( V_i \)),
- Then it waits for messages, collecting all the values received during the round \( f + 2 \) in a bag \( rec_i \) (differently from a set, a bag can contain several copies of the same value),
- And finally, \( p_k \) decides on a value it has received more than \( f \) times.

As we can see, what does this additional round is to prevent from terminating a faulty process that would decide a different value. But, as a process that commits an omission failure is faulty (and consequently not required to decide), this strategy is correct. So, a faulty process either decides correctly or does not decide. Let us also notice that the protocol enjoys the following property: a process that fails only by send omission decides.

**Theorem 2** The protocol described in Figure 3 solves the Uniform Consensus problem in synchronous systems where up to \( f < n/2 \) processes can commit omission failures. It requires \( f + 2 \) rounds.

**Proof** The proof of the Validity property is left to the reader. For the Uniform Agreement property, let us first observe that all the correct processes \( p_k \) have the same value \( v \) in their variables \( V_i \) at the end of the round \( f + 1 \). (This follows from the fact that the protocol described in Figure 2 satisfies the Agreement property despite up to \( f \) processes failing by crash or omission.) So, at least \( n - f \) processes have \( V_i = v \) at the end of \( f + 1 \). It follows that if a process receives at least \( f + 1 \) copies of the same value during the round \( f + 2 \), that value is necessarily \( v \). Hence, Uniformity of the Agreement property.

For the proof of the Termination property, we have to show that the correct processes decide. Let us first observe that all the processes that do not crash (this set includes at least the correct processes) execute the round \( f + 2 \). Due to Agreement on the value of the variables \( V_i \) of the correct processes at the end of the round \( f + 1 \), we conclude that at least \( n - f \) processes broadcast \( v \) during the sending phase of the round \( f + 2 \). Due
4.3 An Impossibility Result

This section shows that, when one is interested in Uniform Consensus in presence of omission failures, the constraint on the number of processes that can be faulty is not a limitation of the proposed protocol. It is actually a necessary requirement to solve the problem.

Theorem 3 There is no protocol solving the Uniform Consensus problem in a synchronous system where up to \( f \geq n/2 \) processes can commit omission failures.

Proof The proof, based a very classical "partitioning" argument, is by contradiction. Let us assume that there is a protocol \( A \) that solves the uniform binary consensus problem (i.e., only the values 0 and 1 can be proposed) in presence of up to \( f \geq n/2 \) faulty processes.

Let us partition the processes in two disjoint subsets \( P_0 \) and \( P_1 \) such that \( P_0 \) includes \( f \) processes, and \( P_1 \) includes the other \( n - f \) processes. Let us consider the three following executions:

- **Execution \( E_0 \).** All the processes propose the value 0, all the processes in \( P_0 \) are correct, and all the processes in \( P_1 \) have initially crashed. As by assumption the protocol \( A \) is correct, we conclude that the processes in \( P_0 \) decide (Termination), they decide the same value (Uniform Agreement) which is 0 (Validity) as that is the only proposed value.

- **Execution \( E_1 \).** All the processes propose the value 1, all the processes in \( P_1 \) are correct, and all the processes in \( P_0 \) have initially crashed. Similarly to the previous item, as by assumption the protocol \( A \) is correct, we conclude that the processes in \( P_1 \) decide (Termination), they decide the same value (Uniform Agreement) which is 1 (Validity) as that is the only proposed value.

- **Execution \( E_{01} \).** All the processes in \( P_0 \) propose 0, all the processes in \( P_1 \) propose 1, all the processes in \( P_0 \) are correct, and all the processes in \( P_1 \) are faulty. More precisely, the processes in \( P_1 \) commit the following faults: they commit send omission with respect to the messages they have to send to the processes of \( P_0 \), and receive omission with respect to the messages they have to receive from the processes of \( P_0 \). Moreover, there are no omissions inside \( P_1 \).

  - The executions \( E_0 \) and \( E_{01} \) are indistinguishable for any process \( p_i \in P_0 \). So, the processes of \( P_0 \) have to decide 0 in \( E_{01} \).
  
  - Similarly, the executions \( E_1 \) and \( E_{01} \) are indistinguishable for any process \( p_j \in P_1 \). So, the processes of \( P_1 \) have to decide 1 in the execution \( E_{01} \).

So, in execution \( E_{01} \), some processes decide 0, while others decide 1. Hence, the protocol \( A \) does not guarantee the Uniform Agreement property. A contradiction.

\[ \square \text{Theorem 3} \]
5 Consensus in Systems with Byzantine Failures

For completeness of the tutorial concern of the paper we present a consensus protocol that allows processes to exhibit an arbitrary behavior. It has been shown that \( f < n/3 \) is a necessary requirement for such protocols [25]. The protocol we present here (due to Berman and Garay [3], hence denoted BG) requires a stronger assumption, namely, \( f < n/4 \), but is particularly simple and elegant. It has also the noteworthy property to use messages of constant size \( (b \text{ bits}) \), as every message carries a single value\(^6\). Moreover, this protocol is based on the rotating-coordinator paradigm which has proved to be a valuable paradigm in the design of a lot of distributed algorithms.

Principles In addition to Termination and Agreement, a protocol coping with Byzantine failures must force the decided value to be \( v \) when all the correct processes propose \( v \) (Weak Validity). To attain this goal, The BG protocol (Figure 4) is based on the following principle. (1) If the occurrence number of the most current value bypasses some threshold, that value will be the decided value. (2) Otherwise, the coordinator paradigm is used to force a value to be adopted by enough processes in order the previous property become satisfied.

To implement this idea, the protocol naturally uses a sequence of stages, each made up of two rounds - conceptually related to the items (1) and (2). During each stage, each process computes an estimate of the decision value (kept in the local variable \( V_i \), initialized to \( v_i \)), and the aim of the sequence of stages is to guarantee that a value eventually becomes “present enough” to bypass the threshold. More precisely, we have:

- The first round of stage \( k \) (i.e., the round whose number is \( r = 2k - 1 \)) is an estimate determination. The processes exchange their current estimate values \( V_i \), and each process \( p_i \) determines the one it sees the most often\(^7\) and keeps it in \( \text{most}_\text{freq}_i \).

- The second round of stage \( k \) (i.e., the round whose number is \( r = 2k \)) is an estimate adoption. For each process \( p_i \), as indicated previously, if the occurrence number of the estimate \( v \) it has seen the most often bypasses the threshold, \( p_i \) adopts it as new estimate. The other case is solved by the rotating coordinator paradigm as follows. During round \( r = 2k \), process \( p_k \) acts a coordinator role: it broadcasts its \( \text{most}_\text{freq}_k \) value to all the processes \( p_i \) in order they adopt it in case they cannot adopt their \( \text{most}_\text{freq}_i \) value.

Let us notice that, as at most \( f \) processes are faulty, \( f + 1 \) stages necessarily include a stage whose coordinator is correct. So, this coordinator will impose the same estimate value to the correct processes if, up to this stage, no estimate value was “present enough” to bypass the threshold.

<table>
<thead>
<tr>
<th>Function Consensus((v_i))</th>
</tr>
</thead>
<tbody>
<tr>
<td>( V_i \leftarrow v_i );</td>
</tr>
<tr>
<td>( \text{when } r = 1,3,\ldots;2f-1,2f+1 \text{ do } % r: \text{round number } % )</td>
</tr>
<tr>
<td>\text{begin round}</td>
</tr>
<tr>
<td>\text{foreach } j: \text{send } (V_i) \text{ to } p_i;</td>
</tr>
<tr>
<td>let \text{rec}_j \text{ be the bag of values received during } r;</td>
</tr>
<tr>
<td>\text{most}_\text{freq}_j \leftarrow \text{most frequent value in } \text{rec}_j;</td>
</tr>
<tr>
<td>\text{occ}r\text{e}<em>j \leftarrow \text{occurrence number of } \text{most}</em>\text{freq}_j</td>
</tr>
<tr>
<td>\text{end round; }</td>
</tr>
<tr>
<td>\text{when } r = 2,4,\ldots;2f;2f+1 \text{ do } % r: \text{round number } % )</td>
</tr>
<tr>
<td>\text{begin round}</td>
</tr>
<tr>
<td>if ( (i = r/2) ) then \text{foreach } j: \text{send } \text{most}_\text{freq}_j \text{ to } p_i \text{ endif};</td>
</tr>
<tr>
<td>if ( v \text{ received from } p_i/2 ) then \text{coord}_i\text{val} \leftarrow v \text{ else } \text{coord}_i\text{val} \leftarrow v_i \text{ endif};</td>
</tr>
<tr>
<td>if ( \text{occ}r\text{e}<em>i \geq n/2 + f ) then ( V_i \leftarrow \text{most}</em>\text{freq}_i \text{ else } V_i \leftarrow \text{coord}_i\text{val} \text{ endif};</td>
</tr>
<tr>
<td>\text{end round; return } (V_i)</td>
</tr>
</tbody>
</table>

Figure 4: The BG Protocol for Byzantine Failures \((f < n/4)\)

The threshold value is \( n/2 + f \). As shown by the following lemma this threshold value is required to guarantee the Agreement property cannot be violated despite up to \( f \) processes exhibiting a Byzantine behavior. The lemma shows that the protocol maintains Agreement as soon as the correct processes have converged to the same estimate (\textit{persistence property}).

\(^6\)The interested reader can consult [15] for more efficient (but also much more complicated and sophisticated) protocols.

\(^7\)If several values are “most common”, one is deterministically selected.
Lemma 1 Let $f < n/4$, and consider the situation where, at the beginning of stage $k$, all the correct processes have the same estimate value $v$. They will never change their estimate value, thereafter. 

Proof. It follows from the lemma assumption that the bag $\text{rec}_i$ of any correct process contains at least $n - f$ copies of $v$ at the end of the first round of stage $k$ (round $r = 2k - 1$), and (as $n - f > n/2$) its variable $\text{most}_\text{freq}_k$ contains $v$. From $f < n/4$, we get $n - f > n/2 + f$, from which we conclude that during the second round of $k$ (round $r = 2k$) the estimate $V_i$ of a correct process is set to $\text{most}_\text{freq}_k$, i.e., keeps the value $v$. $\Box$

Theorem 4 Let $f < n/4$. The protocol described in Figure 4 satisfies the Termination, Agreement and Weak Validity properties in presence of up to $f$ Byzantine processes. It requires $2(f + 1)$ rounds.

Proof. Weak Validity is an immediate consequence of Lemma 1: if all the correct processes start with the same value $v$, they keep it until they decide. Termination follows from synchrony: a correct process decides at the end of round $2(f + 1)$.

Let us consider the agreement property. Since there are $f + 1$ stages and at most $f$ Byzantine processes, there is at least one stage coordinated by a correct process. Let $k$ be the first stage coordinated by a correct process (say $p_k$), and $p_k$ be a correct process. At the end of stage $k$, $p_k$ has some value $v$ in $V_i$:

- If it has executed $V_i \leftarrow \text{most}_\text{freq}_k$, then we conclude that at least $n/2 + f + 1$ processes had $v$ as estimate (in their $V_j$ variables) at the beginning of stage $k$. Therefore, the coordinator $p_k$ of stage $k$ (which is correct) received at least $n/2 + 1$ copies of $v$; so, it has seen a single most frequent value, namely a majority value, and consequently the value it sent during the second round of stage $k$ is $\text{most}_\text{freq}_k = v$. Hence, whatever the assignment executed by another correct process $p_j$ at the end of stage $k$ (i.e., $V_j \leftarrow \text{most}_\text{freq}_k$ or $V_j \leftarrow \text{coordval}_j$), we have $V_j = v$. It follows that all the correct processes have the same estimate value at the end of stage $k$.

- If no correct process $p_k$ has executed $V_i \leftarrow \text{most}_\text{freq}_k$, then they all executed $V_i \leftarrow \text{coordval}_i$, and consequently, all the correct processes have the same estimate value at the end of stage $k$ (remind that, as $p_k$ is correct, it sent the same value to all the processes).

In both cases, due to Lemma 1, the correct processes will not modify these estimates in the future. Hence, the Agreement property. $\Box$

6 Concluding Remarks

When considering the properties defining the consensus variants, let $T$ stands for Termination, $A$ for Agreement, $\text{UA}$ for Uniform Agreement, $V$ for Validity, and $\text{WV}$ for Weak Validity. Table 1 summarizes the different versions of the consensus problem, the failure model they are associated with, and their necessary requirement on the value of $f$.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Specification</th>
<th>Failure Model</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consensus</td>
<td>$T + A + V$</td>
<td>Crash/Omission</td>
<td>$f &lt; n$</td>
</tr>
<tr>
<td>Uniform Consensus</td>
<td>$T + \text{UA} + V$</td>
<td>Crash</td>
<td>$f &lt; n$</td>
</tr>
<tr>
<td>Uniform Consensus</td>
<td>$T + \text{UA} + V$</td>
<td>Omission</td>
<td>$f &lt; n/2$</td>
</tr>
<tr>
<td>Weak Consensus</td>
<td>$T + A + \text{WV}$</td>
<td>Byzantine</td>
<td>$f &lt; n/3$</td>
</tr>
</tbody>
</table>

Table 1: Consensus Problems and Failure Models.

This paper has presented consensus protocols for the synchronous computation model, and revisited some of them. It has shown that different consensus specifications are required by the different failure models. In that sense, the paper provided a guided tour to the consensus problem in synchronous systems prone to process failures (and thus complements [16]).

As noticed in the Introduction, consensus has no deterministic solution in asynchronous systems where even a single process may crash [13]. This means that the underlying system must be augmented with additional synchrony assumptions in order to make consensus solvable by a deterministic protocol [9, 11]. Such assumptions have been abstracted in the notion of unreliable failure detectors [5] ([6] presents the necessary and sufficient requirements on failure detection for solving consensus). Recently, a new approach has been proposed to address
the consensus problem: it consists in identifying sets of input vectors (vectors of proposed values) for which consensus is solvable despite up to \( f \) faulty process \([23]\) (this approach is related to error correcting codes \([14, 24]\)).

References


P1 n`1467


