Consensus in Asynchronous Systems Where Processes Can Crash and Recover

Michel HURFIN, Achour MOSTÉFAOUI, Michel RAYNAL
IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France
{hurfin, achour, raynal}@irisa.fr

Abstract

The Consensus problem is now well identified as being one of the most important problems encountered in the design and the construction of fault-tolerant distributed systems. This problem is defined as follows: processes have to reach a common decision, which depends on their inputs, despite failures.

We consider the Consensus problem in asynchronous distributed systems augmented with unreliable failure detectors. Several protocols have been proposed for these systems, when process crashes are assumed to be definitive. This paper addresses the Consensus problem in a more practical asynchronous system model, namely in a context where processes can crash and recover. As a process crash entails the loss of its volatile memory, each process is equipped with a stable storage. So, to be efficient a Consensus protocol has to log as few critical data as possible. The proposed protocol uses a new class of failure detectors suited to the crash/recovery model. It is particularly efficient when, whether there are crashes or not, the underlying failure detector makes few mistakes. Additionally, the proposed protocol tolerates message duplication and copes with some message losses.

1 Introduction

Consensus is now well identified as being a fundamental paradigm for fault-tolerant distributed systems. It abstracts an important family of problems known as agreement problems. Among those problems, atomic commitment and atomic broadcast are the most encountered. In both problems, processes have to agree on a common decision. In the first case, the common decision consists in a single (commit or abort) order each process has to apply to its local computation. In the second case, the common decision is a pair consisting of a set of messages plus a single delivery order for these messages. [4] and [8] describe consensus-based solutions to these problems ([4] addresses atomic broadcast, while [8] addresses atomic commitment). More generally, it appears that consensus is an underlying basic building block on top of which solutions to agreement problems can be designed.

The consensus problem is defined in the following way: each process proposes an initial value to the others, and, despite failures, all correct processes have to agree on a common value (called decision value), which has to be one of the proposed values. Unfortunately, this apparently simple problem has no deterministic solution in asynchronous distributed systems that are subject to even a single process crash failure: this is the so-called Fischer-Lynch-Paterson (FLP)'s impossibility result [7]. Intuitively, this negative result is due to the impossibility to safely distinguish (in an asynchronous setting) a crashed process from a slow process (or from a process with which communications are very slow). It is important to realize that this impossibility result also applies to agreement problems (e.g., to atomic commitment and to atomic broadcast), when they are considered in the context of asynchronous systems.

The FLP impossibility result has motivated researchers to find a set of minimal assumptions that, when satisfied by a distributed system, makes consensus solvable in this system. The concept of an unreliable failure detector introduced by Chandra and Toueg constitutes an answer to this challenge [4]. From a practical point of view, an unreliable failure detector can be seen as a set of oracles: each oracle is attached to a process and provides it with a list of processes it suspects to have failed. An oracle can make mistakes by not suspecting a failed process or by suspecting a not failed one. By restricting the domain of mistakes they can make, several classes of failure detectors can be defined. From a formal point of view, a failure detector class is defined by two properties: a completeness property which addresses detection of actual failures, and an accuracy property which restricts the mistakes a failure detector can make.

In [4], Chandra and Toueg have designed solutions to the consensus problem in the context of a crash/no recovery model. In this model process crashes are definitive (i.e., once crashed, a process never recovers), so, a failed process is a crashed process. In this context, they present a protocol that solves the consensus problem, assuming that a majority of processes do not crash and that the underlying failure detector belongs to the class $\mathcal{S}$. This class is defined by the two following properties:

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

It has been shown in [5] that these conditions are the weakest ones to solve the consensus problem. While the completeness property can always be realized by using timeouts, it is important to note that the accuracy property can only be approximated in purely asynchronous distributed systems (otherwise, this would contradict the FLP impossibility result!).

In this paper, we address the consensus problem in a more realistic system, namely, the crash/recovery model. In this model,
processes can crash and later recover. Moreover, we assume that
when a process crashes (1) it loses the content of its volatile mem-
ory, and (2) the set of messages it has been delivered while it was
crashed is also lost. To cope with crashes, each process has a stable
storage in which it can log critical data; but in order to be efficient,
a consensus protocol must not consider all its data as critical and
must not log a critical data every time it is updated. (The protocol
presented in this paper addresses these efficiency issues.) At
a given time, a process can be up (working) or down (crashed).
When it is up, it behaves according to its specification. Moreover,
when considering its lifetime, a process can be sound or flawed.
A process is sound if, despite possible crashes, it will eventually
remain permanently up. It is flawed otherwise. From a practical
point of view, "permanently up" means that the process is up long
enough to terminate the protocol it executes. So, in this model a
failed process is a flawed process.

To cope with process crashes and recoveries, we define a fail-
ure detector class, we call $\Diamond S_e$, based on a completeness property
and an accuracy property that take into account sound and flawed
processes. Then, we propose a protocol that solves the consen-
sus problem in a crash-recovery model assuming the underlying
failure detector belongs to the class $\Diamond S_e$. Similar to consensus
protocols suited to the crash/no recovery model [1, 4, 10, 13], the
proposed protocol is based on the rotating coordinator paradigm:
processes proceed in asynchronous consecutive rounds, and each
round is managed by a predetermined process (the round coordina-
tor). When the underlying failure detector makes no mistake, the
proposed protocol is as efficient as [10]: in such a no-mistake sce-
nario, during a round, the decision is obtained in 2 communication
steps if the current coordinator has not crashed; if it has crashed,
only one communication step is required to proceed to the next
round.

The problem of solving consensus in a crash/recovery model
equipped with failure detectors has been first addressed in [6] and
then in [3, 12]. In [6, 12] the entire state of a process is recorded
into stable storage at every state transition. [3] determines, in the
context of asynchronous distributed systems equipped with fail-
ures detectors, a condition $C$ under which the problem cannot be
solved without using a stable storage. This condition depends on
two values, namely, $n_c$, the number of processes that never crash,
and $n_0$, the maximal number of processes that forever oscillate
between up and down periods or that eventually remain crashed:
$C = (n_c \leq n_0)$. More, [3] also proposes two protocols. The first
one assumes $C$ is satisfied (i.e., it considers the case where a stable
storage is necessary); this protocol requires to save state informa-
tion twice during each round. The second protocol assumes $C$ is
false (i.e., it addresses the case where no stable storage is not nec-
necessary). Protocols proposed in [6, 12] and in this paper use similar
failure detectors that outputs list of "suspects"; so, their outputs are
bounded. [3] uses failure detectors whose outputs are unbounded
(in addition to lists of suspects, the outputs include counters). The
proposed protocol assumes $C$ and consequently uses a stable stor-
age. It requires a process to log some state data at most once during
a round.

This paper is composed of six sections. Section 2 introduces
the crash/recovery model. Then, Section 3 provides a definition
of consensus suited to this model, and introduces the $\Diamond S_e$ class
of failure detectors. Section 4 presents the protocol: its under-
lying principles and its description. Due to space limitation the
proof that the protocol satisfies the properties stated in Section 3.2
is omitted. The reader interested in this proof may consult [11].

Section 5 discusses channels reliability.

2 Crash/Recovery System Model

2.1 Processes

We consider a system consisting of a finite set of processes
$\Pi = \{p_1, p_2, \ldots, p_n\}$. At a given time, a process is either up
or down. When it is up, a process progresses at its own speed
behaving according to its specification (i.e., it correctly executes
its program text). While being up, a process can fail by crashing:
it then stops working and becomes down. A down process can later
recover: it then becomes up again and restarts by invoking a
recovery procedure. So, the occurrence of the local event crash
(resp. recover) generated by the local environment of a process,
makes this process transit from up to down (resp. from down to
up).

A process is equipped with two local memories: a volatile
memory and a stable storage. The primitives log and retrieve
allow an up process to access its stable storage. When it crashes,
a process definitely loses the content of its volatile memory; the
content of a stable storage is not affected by crashes.

2.2 Communications

Processes communicate and synchronize by sending and re-
ceiving messages through channels. We assume there is a bidi-
rectional channel between each pair of processes. Channels are not
required to be FIFO. Moreover, they can duplicate messages.
Message transfer delays are arbitrary but finite. Lossy channels
are considered in Section 5.

A process sends a message by invoking a send primitive. When
a message arrives at a process it is deposited in its input buffer that
is a part of its volatile memory. The process will consume it by
invoking a reception primitive. If the input buffer is empty, this
primitive blocks its caller until a message arrives.

The combination of crashes, recoveries and arbitrary message
transfer delays can entail message losses; the set of messages that
arrive at a process while it is down are lost. This is illustrated in
Figure 1: messages $m_1, m_4$, and $m_3$ are received by $p_i$, while $m_2$
and $m_4$ arrived during a down period are lost. Notice that if a
given process never crashes, it will receive all the messages sent
to it. In other words, a recovering process knows it has possibly
missed some messages sent to it.

The multiplicity of processes and the message-passing com-
munication makes the system distributed. The absence of timing
assumptions makes it asynchronous. It is the role of upper layer
protocols to make it reliable.
3 The Consensus Problem

The definition of the consensus problem requires a definition of a "correct process". This is done in Section 3.1. As the words "correct" and "faulty" are used with a precise meaning in the crash/no-recovery model [4], and as, for clarity purpose, we do not want to overload them semantically, we define their equivalents in the crash/recovery model, namely, "sound" and "flawed" processes. If crashed processes never recover, "sound" and "flawed" processes are equivalent with "correct" and "faulty" processes, respectively. Section 3.2 specifies the three properties defining the consensus problem in this model.

3.1 Sound and Flawed Processes

A sound process is a process that eventually remains permanently up. A flawed process is a process that is not sound. So, after some time, a sound process never crashes. On the other hand, after some time, a flawed process either permanently remains crashed or permanently oscillates between crashes (down periods) and recoveries (up periods). From a practical point of view, a sound process is a process that, after some time, remains up long enough to complete the upper layer protocol.

It is important to note that, when considering a process, the words "up" and "down" refer to its current state (as seen by an external observer), while the words "sound" and "flawed" refer to its whole execution.

3.2 The Consensus Problem

The definition of the consensus problem in the crash/recovery model is obtained from the one given in the crash/no-recovery model by replacing “correct process” by “sound process”.

Each process $p_i$ has an initial value $v_i$ that it proposes to the others, and all sound processes have to decide on a single value that has to be one of the proposed values. More precisely, the consensus problem is defined by the following three properties:

- Termination: Every sound process eventually decides some value.
- Uniform Validity: If a process decides $v$, then $v$ was proposed by some process.
- Uniform Agreement: no two processes (sound or flawed) decide differently.

3.3 Enriching the Model to Solve Consensus

As noted previously, the consensus problem has no deterministic solution in the simple crash/no-recovery model. This model has to be enriched with a failure detector that, albeit unreliable, satisfies some minimal conditions in order that the consensus be solvable.

In the same way, the crash/recovery model has to be augmented with a failure detector in order that the consensus can be solved. We consider in this paper the class $\Diamond S_{cr}$ of failure detectors defined by the following two properties:

- Strong Completeness: Eventually, every flawed process is permanently suspected by every sound process.
- Eventual Weak Accuracy: Eventually, there is a sound process that is never suspected. (The reader can verify that $\Diamond S_{cr}$ is equivalent to $\Diamond S$ [4] when processes that crash never recover.)

As we can see, failure detectors belonging to $\Diamond S_{cr}$ may be unreliable. For example, a sound process may be suspected by all processes that are currently up.

4 Description of the Protocol

The proposed protocol borrows some principles from protocols solving the consensus problem in the crash/no recovery model [4, 10, 13]. Each process $p_i$ is endowed with a local variable $est_i$ containing its local estimate of the decision value; initially, $est_i = v_i$. Moreover, the protocol proceeds in consecutive asynchronous rounds. On each round $r$, a predetermined process $p_c \equiv (r \mod n) + 1$ is assigned to be the current initiator and the protocol attempts to impose $p_c$’s estimate as the decision value; $p_c$ is called the current initiator. So, at the beginning of $r$, the current initiator $p_c$ sends its value $est_c$ to all; then, it behaves as the other processes.

To ease explanations, we first (Section 4.1) assume crashes are definitive. In that case, the proposed protocol can be simplified and becomes close to [10]. Then (Section 4.2), we address recoveries.

4.1 Basic Principles: the No Recovery Case

View of the current round by a process

Let us consider a round $r$. During this round, the protocol directs each process either to decide a value (in that case the decided value will be $est_c$, or to proceed to the next round. To attain this goal, the protocol requires that processes vote either for deciding $est_c$ at the current round, or for proceeding to the next round.

Each process $p_i$ manages three variables that give it a local perception of the current state of the round:

- An integer variable $r_i$ (initialized to 1). Its value represents the round number to which $p_i$ is currently participating.
- A set $endorsing\ group_i$ (initially empty). This set contains all processes that, to $p_i$’s knowledge, have voted for deciding in the current round (as we will see, these processes necessarily know the value $est_c$ of the current initiator $p_c$).
- A set $next\ round\ group_i$ (initially empty). This set contains the set of processes that have voted for proceeding to the next round.

When a process has received a majority of identical votes, it reacts accordingly. More precisely, during the current round, $p_i$ considers $est_c$ as the decision value as soon as $\|endorsing\ group_i\| > \frac{n}{2}$. It proceeds to the next round if $\|next\ round\ group_i\| > \frac{n}{2}$.

How does a process vote during a round

While the previous variables describe the local view $p_i$ has about the current round, its participation in this round is described by a finite state automaton: the local variable $state_i$ defines $p_i$’s state with respect to the votes it has issued. More precisely (state $q_3$ will be introduced in the next paragraph):

$$state_i = \begin{cases} q_0 : p_i \text{ has not yet voted during the current round (i.e., it has not sent any message indicating whether it is in favor of} \\ q_3 : p_i \text{ has voted during the current round and never received any message indicating} \\ \text{whether another process has voted for the same decision value.} \\ q_1 : p_i \text{ has voted during the current round and received} \\ \text{a majority of messages indicating a decision value different from} \\ \text{the one decided during this round.} \\ q_2 : p_i \text{ has voted during the current round and received} \\ \text{a majority of messages indicating that another process has} \\ \text{voted for the same decision value.} \end{cases}$$
deciding during the current round or in favor of proceeding to the next round).

\[ \text{state}_1 = q_1 : p_i \text{ has voted for deciding during the current round} \] (so, \( p_i \) belongs to endorsing group).

\[ \text{state}_1 = q_2 : p_i \text{ has voted for proceeding to the next round} \] (so, \( p_i \) belongs to next round group).

Votes are implemented by STATE messages. Each STATE message carries the identity, the round number, the current estimate and the current state of its sender \( p_k \). Hence, they have the following form: STATE\((p_k, r_k, \text{est}_k, \text{state}_k)\). When \( p_k \) receives such a message, the field \( \text{state}_k \) allows it to know the vote issued by \( p_k \). Whatever is the value of \( \text{state}_k \), \( p_i \) updates the set endorsing group or the set next round group accordingly: \( \text{state}_k = q_1 \) means \( p_k \) voted for deciding during the current round (consequently, \( p_k \) is added to endorsing group); \( \text{state}_k = q_2 \) means \( p_k \) voted for proceeding to the next round (consequently, \( p_k \) is added to next round group).

The votes of \( p_i \), reflected locally by its progress from \( q_0 \) to \( q_1 \) or from \( q_0 \) to \( q_2 \), are controlled by the following rules:

- If, while \( \text{state}_i = q_0 \), \( p_i \) receives a vote for deciding in the current round \( r_i \), it moves to \( q_1 \), updates est\(_k\) to est\(_k\) and votes also in the same way (by broadcasting STATE\((p_i, r_i, \text{est}_k, q_1)\)). Note that the first vote for deciding during the current round is necessarily issued by the current round initiator \( p_i \) that broadcast STATE\((p_i, r_i, \text{est}_i, q_1)\). So, we have \( \text{est}_k = \text{est}_i \).
- If, while \( \text{state}_i = q_0 \), \( p_i \) suspects the current initiator, it moves to \( q_2 \) and votes for proceeding to the next round.

Preventing deadlock situations

Let us consider Figure 2: there are three processes \( p_1, p_2 \) and \( p_3 \) (with \( p_1 \) being the initiator of the current round). The described scenario is the following one:
- \( p_1 \) sent a vote for deciding during this round and moved from \( q_0 \) to \( q_1 \).
- \( p_2 \) suspected \( p_1 \) before receiving its vote. Consequently, it moved from \( q_0 \) to \( q_2 \) and sent a vote for proceeding to the next round.
- \( p_3 \) crashed before sending any vote.

So, after both votes have been received we have:

- endorsing group\(_1\) = endorsing group\(_2\) = \{\( p_1 \)\}
- next round group\(_1\) = next round group\(_2\) = \{\( p_3 \)\}

Neither set contains a majority: so, \( p_1 \) and \( p_2 \) can neither decide nor proceed to the next round. This is a typical deadlock situation. To prevent it, a process is allowed to change its mind by voting for proceeding to the next round after it has voted for deciding during the current round. In the automaton, this change of mind is expressed by a new transition from \( q_1 \) to \( q_2 \). It occurs if the following conditions are satisfied:
- \( \text{state}_1 = q_1 \) (i.e., \( p_i \) has previously sent a vote in favor of deciding in this round).
- \( p_i \) has received a vote from a majority of processes (i.e., \(|\text{endorsing group}\_i \cup \text{next round group}\_i| > n/2\)).

Hence, the meaning of \( q_3 \) is the following:

\[ \text{state}_1 = q_3 : p_i \text{ has first voted for deciding in the current round, and then has voted for proceeding to the next round.} \]

When this “change of mind” transition occurs, \( p_i \) votes in favor of proceeding to the next round by broadcasting a message STATE\((p_i, r, \text{est}_i, q_3)\). When a process \( p_k \) receives such a message, it learns its sender \( p_i \) has voted both for deciding in the current round and for proceeding to the next round (consequently, \( p_k \) updates endorsing group and next round group).

![Figure 2. A Deadlock Situation](image)

Remark. In order to prevent infinite blocking situations, the protocol introduces an asymmetry in the automaton. A process can first vote for a decision to be taken during the current round (transition \( q_0 \rightarrow q_1 \)), and then vote for proceeding to the next round (transition \( q_1 \rightarrow q_2 \)). But, if a process has first voted for proceeding to the next round (transition from \( q_0 \rightarrow q_2 \)), this vote is definitive. This asymmetry is not arbitrary. Making definitive the vote for deciding in the current round, and allowing a process to change its mind by voting for deciding in the current round after having voted for proceeding to the next round, would compromise the safety property of the protocol: in some scenarios this would allow processes to decide different values, and consequently does not guarantee uniform agreement. End of remark.

A value has been decided

When a process \( p_i \) has decided a value, it proceeds to a final state \( q_f \) of its automaton, broadcasts its state to all processes and remains in this state forever. Then, it does not participate in any round.

\[ \text{state}_1 = q_f : p_i \text{ has decided. The decided value is kept in est}_i \]

4.2 Basic Principles: Recovery

Logging critical data

Until it (possibly) crashes, a process \( p_i \) participates in the protocol by progressing from round to round, and in each round, by sending votes as described previously. These votes are processed by the other processes when they receive them. The crash of \( p_i \) during round \( r_i \) must not contradict the behavior it had during previous rounds: in other words, because they affected other processes, the votes it has previously sent cannot be cancelled and replayed differently during a recovery.

It follows that \( p_i \) has to consider \( r_i, \text{est}_i \) and \( \text{state}_i \) as critical data. After starting a round \( r_i \), \( p_i \) logs them when it votes for the first time, i.e., when it moves from \( q_0 \rightarrow q_1 \) or from \( q_0 \rightarrow q_2 \). More precisely:

- When \( p_i \) moves from \( q_0 \rightarrow q_1 \): it has to log the round number \( r_i \) and the value \( \text{est}_i \) it has endorsed (by updating \( \text{est}_i \) and by voting to decide this value during this round).
- When \( p_i \) moves from \( q_0 \rightarrow q_2 \): it has to log the round number \( r_i \) and the fact it has voted for proceeding to the next round, i.e., the state \( q_2 \).
The protocol resulting from the principles introduced in Sections 4.1 and 4.2 is described in Figure 3. It is composed of three procedures. For each of them, we indicate why and when it is called, and describe its operational behavior.

Procedure transitio_to

The procedure transitio_to(q, action) is called by a process p_i when it moves to a new state q, during round r_i. The value of the parameter action is to_log or not_to_log. It is to_log if p_i moves from q_0 to q_1 at line 11, or from q_0 to q_2 at line 29 or from any state to q_f (lines 12, 23). It is not_to_log in all other cases (lines 16, 17, 20, 30).

This procedure first sets state_f to q (line 1). Then, if p_i is recovering (then, recovery_f = true), or if it is starting a new round, its round variables are initialized (line 2). If logging is required, it is done (line 3). Finally, the statements associated with the progression to q are executed:

- If the new state is q_0 and p_i is the initiator for this round, it moves to q_1 (and consequently logs the set of critical data) and broadcasts a vote for deciding during the current round (the sending to itself is realized by updating endorsing_groups_i (line 4) and the sending to others is realized at line 9).
- If the new state is q_1, q_2 or q_f, p_i sends the corresponding vote to itself (line 5 or 6) and to all the other processes (line 9).
- In all other cases p_i sends its new state to all processes (line 9).

Procedure endorsement

This procedure is called by p_i when it learns a process p_k has voted for deciding in the current round, i.e., when it receives a message STATE(p_k, r_k, end, state_k, *false*) with state_k ∈ {q_1, q_2} (lines 25, 27).

p_i first updates est_i to est_k (which is equal to est_k) and adds p_k to endorsing_groups_i (line 10). Moreover, if p_i is in q_0, it moves to q_1 (line 11). Finally, if p_i knows a majority of processes are for deciding in the current round, it decides by moving to q_f (line 12).

Procedure Consensus

This procedure is initially called by p_i to launch its participation in the consensus. It is also called by p_i when it recovers after a crash. We assume that the log contains the initial values of the critical data r_i, est_i, and state_i (namely: 1, v_i and q_0). In this way, the initial call can be processed as a call due to a recovery.

First, p_i retrieves its critical data and sets recovery_i to true (line 13). Then, if state_i = q_f, it has already decided and consequently enters a loop (lines 33-34). In this loop, p_i waits for STATE(p_i, r_i, *true*, *true*, *true*, state_i) messages. When it receives such a message, p_i returns to p_i the decided value.

If p_i has not yet decided, it enters a sequence of asynchronous rounds (lines 15-52). A round starts by calling the procedure transitio_to with appropriate parameters (i.e., according to the fact it is recovering, or it is normally proceeding to the next round, lines 16-17). Then, p_i participates in the current round (lines 18-31) until this round terminates (either because p_i proceeds to the next round, or because it has decided a value, line 18). This participation is composed of three sets of statements:

If, while it is in q_0, p_i suspects the current initiator p_i, then it moves to q_2 (line 29).
procedure transition_to(q, action)
(1) state_e i := q;
(2) if (recovery q) \lor (state e i := q_0) then e := (r_i \mod n) + 1; endorsing group \leftarrow \{\}; next round group \leftarrow \{\} endif;
(3) if (action = log q) then log r_i, e, state_e i; endif;
(4) case state_e i of q_0:
if (a = e) then state_e i := q; endorsing group \leftarrow endorsing group \cup \{p_i\}; log r_i, e, state_e i; endif
q_i:
endorse group \leftarrow endorsing group \cup \{p_i\}
q_o, q_n: next round group \leftarrow next round group \cup \{p_i\}
(6) else skip endif;
(7) endcase;
(8) send state(p_i, r_i, e, state_e i, recovery_q) to \Pi \leftarrow \{p_i\}:
(9) procedure endorsement(p_k, est_k):
(10) est_k := est_k; endorsing group \leftarrow endorsing group \cup \{p_k\};
(11) if (state_e i = q_k) then transition_to(q_f, log q) endif;
(12) if (\| endorsing group \| > n/2) then transition_to(q_f, log q) endif;
(13) retrieve(r_i, est_i, state_e i) from the log; recovery q := true;
(14) if (state_e i = q_f) then send state(p_i, r_i, e, state_q, true) to \Pi \leftarrow \{p_i\} endif;
(15) while (state_e i \neq q_f) do % sequence of asynchronous rounds %
(16) if (recovery_q) then transition_to(state_e i, not log q); recovery_q := false
(17) else est_i := est_i + 1; transition_to(state_e i, not log q) endif;
(18) while (\| next round group \| \leq n/2) and (state_e i \neq q_f) do % automation transitions during a round %
(19) upon reception of state(p_k, r_k, est_k, state_k, recovery_k) do
(20) if (r_k > r_i) then state_e i := q_k; r_k := r_k; est_i := est_k; transition_to(q_0, not log q)
(21) else if (recovery_q) then send state(p_i, r_i, est_i, state_e i, false) to p_k endif endif;
(22) case state_e i of
(23) q_f: state_e i := q_f; est_i := est_k; transition_to(q_f, log q)
(24) q_o: skip
(25) q_i: if (r_i = r_k) then endorsement(p_k, est_k) endif
(26) q_o: if (r_i = r_k) then next round group := next round group \cup \{p_k\} endif
(27) q_o: if (r_i = r_k) then endorsement(p_k, est_k); next round group := next round group \cup \{p_k\} endif
(28) endcase
(29) upon (p_k \in suspected_e i) \land (state_e i = q_0) do transition_to(p_k, log q) endupon
(30) upon (state_e i = q_i) \land (\| endorsing group \| > n/2) do transition_to(q_0, not log q) endupon
(31) endwhile
(32) endwhile;
(33) while (true) do upon reception of state(p_k, r_k, est_k, state_k, recovery_k) such that (state_k \neq q_f)
(34) do send state(p_i, r_i, est_i, q_f, false) to p_k endupon
(35) endwhile

Figure 3. Consensus Protocol Based on $\diamond S_r$. 

If, while it is in q_1 (so it has not yet decided), p_k has received a vote from a majority of processes, then it changes its mind and moves to q_2 (line 30).

When p_i receives a state(p_k, r_k, e, state_k, state_k, recovery_k) message, it processes it in the following way:

- If r_k > r_i, p_i joins round r_k (line 20).
- If r_k \leq r_i and p_k is recovering, then p_i sends back its state to p_k (line 21) (this allows p_k to resynchronize at round r_i).
- Then, p_i processes the message according to the value of state_e i:
  - If it learns a value has been decided, it moves to q_f (line 23).
  - If the message carries a vote for deciding during this round, p_i endorses this vote by calling the procedure endorsement (lines 25, 27).
  - If the message carries a vote for proceeding to the next round, p_i accordingly updates the set next round group (lines 26, 27).

5 Unreliable channels

We have seen that the protocol tolerates message duplication (Section 4.2). Actually, it remains correct despite message losses, if the following property is satisfied by any channel connecting sound processes [6, 9]:

Property $LM$: the last message sent by a sound process is eventually received by its receiver (provided the receiver is sound).

Remark. It is important to note that, since a long time, property $LM$ is used to cope with message losses in communication protocols. In those protocols, an ack(x) control message is used to
acknowledge the receipt of all the messages whose sequence number is \( \leq x \). So, when several \( \text{ack} \) messages are transmitted, only the last one has to be received for the \( \text{ack} \) protocol to work. *End of remark.*

At the implementation level, this means that only the last message sent by a process has to be buffered and re-sent. When \( \mathcal{LM} \) is satisfied, the protocol works and remains correct for the following reason. To progress from round \( r \) to \( r + 1 \), a process \( p_i \) has to cross a synchronization barrier, namely, it must have received a \( \text{STATE}(p_k, r, *, \text{state}_k, *) \) message from a majority of processes \( p_k \) (these receptions make true the condition \( \text{next \_round \_group}[p_i] > n/2 \)). More precisely:

- As the automaton is cycle-free, the last \( \text{STATE}(p_k, r, *, \text{state}_k, *) \) message sent by a process \( p_k \) informs unambiguously the receiver about \( p_k \)'s progress during round \( r \).
- The progress of a process \( p_i \) from round \( r \) to \( r + 1 \) is synchronized by the exchange of \( \text{STATE}(p_k, r, *, \text{state}_k, *) \) messages where \( \text{state}_k = q_2 \) or \( q_3 \). This exchange of \( \text{STATE} \) messages realizes the synchronization barrier that, as long as \( p_i \) has not received a majority of such messages (or a message related to a round \( r \)), prevents it to send any further message \( M' \). This synchronization ensures that if a majority of such last \( \text{STATE} \) messages has been received, then at least one process will progress to round \( r + 1 \).

Actually, a channel connecting \( p_i \) to \( p_j \) and satisfying \( \mathcal{LM} \), can be modeled as a variable shared by \( p_i \) and \( p_j \), a send by \( p_i \) being a write on this variable, and a receive by \( p_j \) being a read. The last value (\( \text{STATE} \) message) written by \( p_i \) remains accessible to its receiver \( p_j \) until \( p_i \) overwrites it. By ensuring some values cannot be overwritten before being read, the synchronization barrier ensures the correct progress of the protocol.

As indicated previously, message omission failures have been considered in [6, 9]. To model channels satisfying the property \( \mathcal{LM} \), the channel stubbornness notion has been introduced in [9]. Expressed in the crash/recovery model, this notion is the following one. Let \( p_i \) and \( p_j \) be two sound processes. The channel connecting \( p_i \) to \( p_j \) is stubborn if, when \( p_i \) sends a message \( M \) to \( p_j \) and indefinitely delays sending any further message \( M' \) to \( p_j \), then \( p_j \) eventually receives \( M \). Stubbornness does not mean that, in order to ensure \( M \) is received, \( p_j \) is not allowed to send any new message \( M' \) to \( p_j \); it only means that, as indicated in [9], “there is no fixed delay after which \( M' \) can be sent without compromising the reception of \( M' \).” \( M \) being the last \( \text{STATE} \) message sent by \( p_i \) during round \( r \). The synchronization barrier of our protocol actually implements this “no fixed delay after which \( M' \) can be sent without compromising the reception of \( M' \).”

Finally, let us also note that if (1) all flawed processes eventually remain down forever, and (2) all channels connecting sound processes satisfy \( \mathcal{LM} \), then the proposed protocol has still the quiescence property [2] (but note that the underlying protocol implementing \( \mathcal{LM} \) property on top of fair lossy channels cannot be quiescent).

6 Conclusion

Several consensus protocols have been proposed for asynchronous distributed systems (augmented with unreliable failure detectors) in which process crashes are definitive. This paper has addressed the consensus problem in a more practical asynchronous system model, namely in a model where processes can arbitrarily crash and recover. As a process crash entails the loss of its volatile memory, each process has been equipped with a stable storage. A consensus protocol has been designed for such a realistic context. The protocol is based on a new class of failure detectors suited to the crash/recovery model. From a recovery point of view, the protocol is particularly efficient as it logs very few critical data and logs at most once during a round. Furthermore, the protocol tolerates message duplication and copes with the loss of some messages.

References