Uniform Agreement Despite Process Omission Failures

Philippe RAIPIN PARVEDY      Michel RAYNAL
IRISA, Université de Rennes 1, Campus de Beaulieu, 35042 Rennes Cedex, France
{praipinp|raynal}@irisa.fr

Abstract

A process fails by omission if it “forgets” to send or receive messages. Considering omission failures is crucial for distributed systems, as such failures model both crash failures and incorrect behavior of process input/output buffers (such as buffer overflow). So, designing protocols that cope not only with crash failures but also with omission failures is a real challenge as soon as one is interested in obtaining real-time dependable distributed systems.

While the consensus problem has received a lot of attention in the crash failure model and in the Byzantine failure model, it has received less attention in the omission failure model. This paper presents a simple uniform consensus protocol for synchronous systems made up of $n$ processes where up to $t$ can commit crash or omission failures. This protocol requires $t + 1$ communication steps. Interestingly, as this bound is tight for crash failures and those are included in omission failures, this shows that $t + 1$ is a tight lower bound for protocols solving uniform consensus in synchronous systems prone to process omission failures. The protocol assumes $t < n/2$ that is a necessary requirement on the maximum number of faulty processes that can be tolerated by any uniform consensus protocol in presence of omission failures. The proposed protocol is then extended, at no additional cost, to solve the Interactive Consistency problem.

Keywords: Consensus, Crash Failure, Interactive Consistency, Message Passing System, Omission Failure, Synchronous Distributed System, Uniform Agreement.

1 Introduction

Context of the paper  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 [12, 16] in the context of synchronous systems prone to Byzantine process failures\(^1\), 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 [4, 6]. 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. (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). Consistency is ensured here by the fact there is a single decided value, its meaning 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 [6]. On the contrary, the consensus problem can be solved in synchronous systems, and several distributed computing textbooks (e.g., [2, 8, 13]) 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 [3, 7]. Moreover, only a few papers address the case of omission failures [17]. A process commits an omission failure when it forgets to send or receive a message. Omission failures are important as they include both process crashes and input/output buffer management failures.

Content of the paper  To illustrate the issues raised by process omission failures, this paper consider distributed agreement problems in synchronous systems prone to such failures, with a special emphasis on the uniform consensus problem. The paper first provides a direct proof that $n > 2t$ is a necessary requirement to solve uniform consensus in such a context (where $n$ is the total number of processes,\(^1\))

---
\(^1\)This problem originated in the development of on-board aircraft control systems (which have to be reliable!).
and \( t \) the maximum number of process that can fail. Then, it presents and proves correct an omission-tolerant uniform consensus protocol. It also shows that this protocol is optimal in the number of communication steps, namely \( t + 1 \). Interestingly, as this bound is tight for crash failures and those are included in omission failures, this shows that \( t + 1 \) is a tight lower bound when solving uniform consensus in synchronous systems prone to process omission failures.

Road map This paper is made up of six sections. Section 2 presents the computation model and the consensus problem. Then, Section 3 presents an omission-tolerant consensus protocol. Section 4 proves it correct. Section 5 shows that the proposed protocol solves also the interactive consistency problem. Finally, Section 6 concludes the paper.

2 Computation Model and Consensus

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.
- 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 \( t \) is an upper bound on the number of faulty processes.

A failure model defines how a faulty process can deviate from its algorithm [10]. We consider here the following 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 crash\(^2\).

It easy to see that these failure models are of increasing “severity” in the sense that any protocol that solves a problem in the Omission failure model, also solves it in the (less severe Crash) failure model [10]. This follows from the fact that if a process crashes, it permanently commits omission failures after it has crashed.

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 \([4, 6]\):

- **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:

---

\(^2\) 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.
3 Uniform Consensus with Omission Failures

Before presenting an omission-tolerant uniform consensus protocol, this section shows that \( n > 2t \) is a necessary requirement for such protocols. A more general result is presented in [15]. This result shows that there is no translation transforming any algorithm tolerant of crash failures into one tolerant of omission failures when \( n \leq 2t \). Although Theorem 1 could be deduced from this general result, it is proved directly here. This direct proof has two noteworthy features: (1) it is particularly simple; and (2) it relies on a fundamental characteristic of distributed executions in presence of failures, namely the fact that there are executions that cannot be distinguished.

3.1 An Impossibility Result

**Theorem 1** There is no protocol solving the Uniform Consensus problem in a synchronous system where up to \( t \geq n/2 \) processes can commit omission failures.

**Proof** The proof, based on 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 \( t \geq n/2 \) faulty processes.

Let us partition the processes in two disjoint subsets \( P_0 \) and \( P_1 \) such that \( P_0 \) includes \( t \) processes, and \( P_1 \) includes the other \( n-t \) 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.

\( \square \) Theorem 1

3.2 Consensus vs Uniform Consensus

The crash-tolerant uniform consensus protocols described in [2, 13, 17] are based on a flood set strategy: the processes proceed by consecutive synchronous rounds, and during each round \( r \geq 1 \) they exchange the new values they have heard during the previous round \( r-1 \). It follows (assuming crashes are the only failures and at most \( t \) processes crash) that the processes that do not crash know the same vector of proposed values at the end of the round \( t+1 \), and can consequently decide the same value from it. Actually, it is possible to show that the previous protocols solve the non-uniform consensus problem even when any number of processes can commit omission failures. (This ensues from the fact that (1) a process forwards only once, during \( r+1 \), a value it has heard during \( r \), and (2) the correct processes neither omit to send or receive messages, from which we can conclude that the correct processes know the same vector of proposed values at the end of the round \( t+1 \).

Unfortunately, although these protocols solve uniform consensus 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 \( (t+1) \) 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.
3.3 A Uniform Consensus Protocol

Fortunately, the inability of the previous protocols to address the “uniform” 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 present here a protocol that solves uniform consensus despite up to \( t < n/2 \) processes that commit omission failures. As \( t < n/2 \) is a necessary requirement, this protocol is optimal with respect to the number of failures that can be tolerated.

The protocol for a process \( p_i \) is described in Figure 1. It underlying principles are simple: (1) they combine the flood set strategy with the pairwise elimination of the processes that are suspected to be faulty; (2) the processes are “self-trusting”, in the sense that a process first suspects the other processes before wondering if it is itself faulty.

\( V_i \) is the vector of proposed values know so far by \( p_i \); \( \bot \) denotes a default value that cannot be proposed; \( \text{suspected}_i \) is the set of processes that \( p_i \) suspects to be faulty. More precisely, \( p_i \) “suspects” \( p_j \) if, during some round \( r \), \( p_i \) does not receive a message from \( p_j \). Let us notice that, in that case, at least one of \( p_i \) or \( p_j \) is faulty but it is not possible to know if \( p_j \) committed a send omission fault or if \( p_i \) committed a receive omission fault. Suspicions are stable in the sense that any set \( \text{suspected}_i \) can only increase (line 12). As soon as a process \( p_i \) suspects \( p_j \), it stops communicating with it (lines 4 and 6). So, during the next round, if \( p_j \) has not crashed, it will not receive a message from \( p_i \) and consequently will suspect it. It follows that if at least one of \( p_i \) or \( p_j \) commits an omission fault during a round \( r \), each of them will include the other in its \( \text{suspected} \) set at the latest during \( r + 1 \).

Let us observe that if \( |\text{suspected}_i| > t \) (line 15), then \( p_i \) suspects at least \( t + 1 \) processes. But, as there are at most \( t \) faulty processes, \( p_i \) can conclude that it is faulty itself. In that case, \( p_i \) stops participating in the protocol (which is expressed by the statement \( \text{return} (\bot) \)). It follows that a process that gets \( \bot \) as a result, knows that it committed omission failures. Moreover, a faulty process that does not crash decides the value decided by the correct processes or \( \bot \).

4 Proof and Optimality of the Protocol

4.1 Preliminary Lemmas

**Lemma 1** Let \( \text{STOPPED}(r) \) be the set of processes \( p_i \) such that \( p_i \) has crashed or executed line 15 during a round \( r' \leq r \). Let \( \text{GOOD}(r) \) be the set of processes \( p_i \) such that (1) \( p_i \notin \text{STOPPED}(r) \) and (2) it exists at least one correct process \( p_j \) such that \( p_i \) has never committed omission failures with respect to \( p_j \) until \( r - 1 \) (included).

We have: (1): \( \forall r : \text{GOOD}(r) = \Pi - \text{STOPPED}(r) \), (2): \( \text{GOOD}(r) \supseteq \text{GOOD}(r + 1) \) and (3): \( \text{STOPPED}(r) \subseteq \text{STOPPED}(r + 1) \).

**Proof** Showing \( \text{GOOD}(r) = \Pi - \text{STOPPED}(r) \) amounts to show that if there is no correct process \( p_j \) with respect to which \( p_i \) never committed omission failures until \( r - 1 \) (included), then \( p_i \) belongs to \( \text{STOPPED}(r) \). If \( p_i \) committed omission failures with respect to each correct \( p_j \) during some round \( r(j) \leq r - 1 \), due to the management of the \( \text{suspected} \) sets we eventually get during some \( r' \leq r \): \( |\text{suspected}_i| \geq n - t \); as \( n > 2t \), this means \( |\text{suspected}_i| > t \). It follows that, at the latest during round \( r \), \( p_i \) executes line 15 and so belongs to \( \text{STOPPED}(r) \).

The assertions \( \text{GOOD}(r) \supseteq \text{GOOD}(r + 1) \) and \( \text{STOPPED}(r) \subseteq \text{STOPPED}(r + 1) \) follow directly from the definitions of \( \text{STOPPED}(r) \) and \( \text{GOOD}(r) \) sets. \( \square \)

**Lemma 2** Assuming that at least one correct process receives \( v_k \) (the initial value of \( p_k \)), let \( r \) be the first round during which a correct process \( p_i \) receives \( v_k \). We have \( r \leq t \).

**Proof** Let \( r \) be the first round during which a correct process \( p_i \) receives \( v_k \). The proof is by contradiction. Let us assume that \( r > t \).

As a process forwards \( (v_k, k) \) at most once, we conclude that from \( p_k \) to \( p_i \), the pair \( (v_k, k) \) has been forwarded by \( r \) distinct processes. So, if \( r > t \), the pair \( (v_k, k) \) has been seen by at least \( t + 1 \) distinct processes. As there are at most \( t \) faulty processes, we conclude that before arriving at \( p_i \) during \( r \), the pair \( (v_k, k) \) has been received by at least one correct process during a round \( r' \) such that \( r' \leq t < r \). This contradicts the fact that \( r \) is the first round during which a correct process \( p_i \) receives \( v_k \). \( \square \)

4.2 Proof of the Protocol

**Theorem 2** The protocol described in Figure 1 solves the Uniform Consensus problem in synchronous systems where up to \( t < n/2 \) processes can commit omission failures. It requires \( t + 1 \) rounds.

**Proof** In this proof, the sentence “process \( p_i \) decides” means “\( p_i \) executes line 17”.

**Termination.** It follows from Lemma 1 that all the processes in \( \text{STOPPED}(t + 1) \) do not decide and all processes in \( \text{GOOD}(t + 1) \) decide. So, we have to show that \( \text{GOOD}(t + 1) \) includes all correct processes.

Let \( p_i \) be a correct process, so it neither crashes nor commits send or receive omission failure. It follows that
when \( p_i \) suspects a process \( p_j \) and includes it in \( \text{suspected}_i \) (line 12), necessarily \( p_j \) is faulty. As there are at most \( t \) faulty processes, it follows that \( \text{suspected}_i \) contains at most \( t \) different processes. Hence the test of line 15 is always false for a correct process \( p_i \). Moreover, due to the synchrony assumption, the rounds progress systematically, and each correct process executes them. It follows that \( \text{GOOD}(t + 1) \) includes all the correct processes.

Validity. This property follows directly from the following observations:

- (1) The entries of the array \( V_i \) of a process \( p_i \) that decides contains only proposed values or \( \bot \) (lines 1, 4 and 9),
- (2) The array \( V_i \) of a process \( p_i \) that decides contains at least one proposed value (line 1),
- (3) Only non-\( \bot \) values can be decided (line 17).

Uniform Agreement. This proof uses Lemmas 1 and 2. We show that all the processes \( p_i \) that belong to \( \text{GOOD}(t + 1) \) have the same array \( V_i \) at the end of \( t + 1 \). Let \( v_k \) be the value proposed by process \( p_k \). We consider two cases.

- \( p_k \) is correct. In that case, all the correct processes receive \( v_k \) during the first round. If \( t = 0 \), we trivially have \( V_i[k] = V_j[k] \) at the end of the first round. If \( t > 0 \), all processes in \( \text{GOOD}(2) \) have received \( v_k \) by the end of the second round, and consequently, as \( \text{GOOD}(t + 1) \subseteq \text{GOOD}(2) \), all the processes in \( \text{GOOD}(t + 1) \) have received \( v_k \).

- \( p_k \) is faulty. We consider three subcases.

  - No correct process receives \( v_k \) by the end of round \( t \). It follows from Lemma 2 that no correct process ever receives \( v_k \). Let us consider a process \( p_j \) that received \( v_k \). As no correct process receives \( v_k \) by the end of round \( t \), we can conclude that, between round 1 and round \( t \), \( p_j \) committed omission failures with respect to every correct process, from which it follows that \( p_j \in \text{STOPPED}(t + 1) \). Consequently, no process in \( \text{GOOD}(t + 1) \) has received \( v_k \). Hence, agreement follows.

    - A correct process receives \( v_k \) by the end of round \( \ell < t \). In that case, all correct processes receive \( v_k \) at the latest by the end of \( \ell + 1 \leq t \). Consequently, all the processes in \( \text{GOOD}(\ell + 2) \) (they communicate without omission with at least one correct process), received \( v_k \) by the end of \( \ell + 2 \). As \( \text{GOOD}(t + 1) \subseteq \text{GOOD}(\ell + 2) \), agreement follows.

    - The first correct process that receives \( v_k \), receives it during round \( t \). In that case, \( v_k \) has been forwarded along a path including \( t \) faulty processes (as a process sends a pair \( (v_k, k) \) at most once), i.e., this path includes all the faulty processes. It follows that all the faulty processes have \( v_k \). Moreover, as a correct knows \( v_k \) during \( t \), all the correct processes have it by the end of \( t + 1 \). The agreement follows.

\[ \square \text{Theorem 2} \]
4.3 Remark on Termination

As we have seen, the processes that decide are the processes of GOOD($t + 1$). This set includes the correct processes plus faulty processes. So, interestingly, the protocol does not systematically direct a faulty process that does not crash to terminate with $\bot$. More precisely, let $p_i$ be a process that does not crash, and let $P \text{Jive wrt } i$ be the set of processes $p_j$ such that $p_i$ committed an omission wrt $p_j$ or $p_j$ committed an omission wrt $p_i$. We have

$$([ P \text{Jive wrt } i ] \leq t) \Rightarrow p_i \text{ decides.}$$

From a practical point of view, this means that when there are few omission failures, all processes decide.

4.4 Optimality of the Protocol

Theorem 3 $t + 1$ is a tight lower bound on the number of rounds (communication steps) required to solve uniform consensus in presence of crash or omission failures.

Proof It has been proved that $t + 1$ is a lower bound on the number of rounds (communication steps) required by any protocol to solve uniform consensus in presence of crash failures [1, 5]. Moreover, as indicated in Section 2.2, crash failures are also omission failures, from which we conclude that $t + 1$ is a lower bound on the number of rounds required to solve uniform consensus in presence of omission failures. Finally, as the protocol described in Figure 1 solves uniform consensus despite omission failures in $t + 1$ rounds, this lower bound is tight. □Theorem 3

The following corollary is a direct consequence of the previous theorem.

Corollary 1 The protocol described in Figure 1 is optimal with respect to the number of rounds (communication steps).

5 The Interactive Consistency Problem

As noticed in the Introduction, the interactive consistency (IC) problem has been defined in the context of synchronous systems prone to Byzantine failures [16]. We consider here its uniform version in the context of synchronous systems prone to process omission failures. Each process $p_i$ proposes a value, and the processes have to decide a vector $D$ such that the following properties are satisfied:

- IC-Uniform Agreement. No two different vectors are decided.
- IC-Termination. Each correct process decides.
- IC-Validity. The decided vector $D$ is such that $D[i] \in \{v_i, \bot\}$, and is $v_i$ if $p_i$ is not faulty.

Applications of interactive consistency can be found in [11]. Interestingly, we have the following result:

Theorem 4 The protocol described in Figure 1 (where line 17 is replaced by the statement $\text{return}(V_i)$) solves the interactive consistency problem in synchronous systems where up to $t < n/2$ processes can commit omission failures. It requires $t + 1$ rounds.

Proof The theorem follows from the following observations. Theorem 2 has shown that the protocol described in Figure 1 guarantees that:

- At the end of the first round, each array $V_i$ of a correct process includes the values proposed by all correct processes,
- Any two processes $p_i$ and $p_j$ that execute line 17 have the same array, namely $V_i = V_j$,
- If a process exits the protocol at line 15 then it is faulty, and all correct processes execute line 17.

It follows from these observations that, when replacing line 17 with the statement “$\text{return}(V_i)$”, the protocol solves the uniform interactive consistency problem. □Theorem 4

6 Conclusion

The focus of this paper was on omission failures in the context of synchronous distributed systems. (Let us notice that, as real-time systems are usually synchronous, this protocol is particularly well-suited for real-time distributed systems.) Such a type of failure is worth considering as, in addition to crash failures, it also includes input/output buffer failures. To illustrate the design of protocols in such a context, the paper has considered the uniform consensus problem. It has shown that $n > 2t$ is a necessary requirement for solving uniform consensus despite up to $t$ process omission failures in a system of $n$ processes. Such a protocol has been presented. Interestingly, it has been shown that this protocol is optimal in the number of rounds (namely, $t + 1$ communication steps).

For readers interested in the consensus problem, let us notice that solving consensus in asynchronous distributed systems requires additional assumptions (e.g., equipping the underlying system with failure detectors [4], or restricting the set of input vectors proposed by processes [14]). An introductory survey can be found in [9]. Furthermore, it is shown in [11] that, in asynchronous distributed systems

3This means that, whatever the protocol, it has executions that require at least $t + 1$ communication steps.
prone to process crash failures, the interactive consistency problem and the design of a perfect failure detector are equivalent problem.

References


