Consensus in Byzantine asynchronous systems

Roberto Baldoni a,*, Jean-Michel Hélary b, Michel Raynal b, Lenaik Tangui b

a DIS, Università di Roma “La Sapienza”, Via Salaria 113, I00198, Roma, Italy
b IRISA, Campus de Beaulieu, 35042 Rennes cedex, France

Abstract

This paper presents a consensus protocol resilient to Byzantine failures. It uses signed and certified messages and is based on two underlying failure detection modules. The first is a muteness failure detection module of the class ◇M. The second is a reliable Byzantine behaviour detection module. More precisely, the first module detects processes that stop sending messages, while processes experiencing other non-correct behaviours (i.e., Byzantine) are detected by the second module. The protocol is resilient to F faulty processes, \( F \leq \min\left(\lceil(n - 1)/2\rceil, C\right) \) (where C is the maximum number of faulty processes that can be tolerated by the underlying certification service).

The approach used to design the protocol is new. While usual Byzantine consensus protocols are based on failure detectors to detect processes that stop communicating, none of them use a module to detect their Byzantine behaviour (this detection is not isolated from the protocol and makes it difficult to understand and prove correct). In addition to this modular approach and to a consensus protocol for Byzantine systems, the paper presents a finite state automaton-based implementation of the Byzantine behaviour detection module. Finally, the modular approach followed in this paper can be used to solve other problems in asynchronous systems experiencing Byzantine failures.

© 2003 Elsevier B.V. All rights reserved.

Keywords: Asynchronous systems; Consensus problem; Byzantine failures; Fault-tolerance; Failure detectors

1. Introduction

Consensus is a fundamental paradigm for fault-tolerant asynchronous distributed systems. Each process proposes a value to the others. All correct processes have to agree
(Termination) on the same value (Agreement) which must be one of the initially proposed values (Validity). Solving consensus in asynchronous distributed systems where processes can crash is a well-known difficult task: Fischer, Lynch and Paterson have proved an impossibility result [7] stating that there is no deterministic solution to the consensus problem in the presence of even a single crash failure. A way to circumvent this impossibility result is to use the concept of unreliable failure detectors introduced by Chandra and Toueg [2]. A failure detector is composed of failure detection modules, one by process, that provides it with a list of processes currently suspected by the detector to have crashed. A failure detection module can make mistakes by not suspecting a crashed process or by erroneously suspecting a correct one. Formally, a failure detector is defined by two properties: completeness (a property on the actual detection of process crashes), and accuracy (a property that restricts the mistakes on erroneous suspicions). Among those classes, [3] proves that the weakest one denoted $\diamond S$ allows to solve the consensus problem in a crash failure model is based on strong completeness (eventually every crashed process is detected by every correct process) and eventual weak accuracy (eventually, some correct process is not suspected by any correct process).

Solving consensus in an environment where processes can exhibit a Byzantine behaviour (i.e., arbitrarily deviate from their program specification) is notably difficult. First steps in this direction have been carried out by Malkhi and Reiter [10], Kihlstrom et al. [9] and Doudou and Schiper [5]. Malkhi and Reiter [10] have been the first to propose an extension of failure detectors able to cope with other types of failures. They have introduced a failure detector class $\diamond S(bz)$ based on the notion of a quiet process. A process $p$ is quiet if there exists a time after which some correct process does not receive anymore messages from $p$. (Note that a crashed process is a quiet process, but the converse is not true.) Failure detectors of class $\diamond S(bz)$ satisfy strong completeness and eventual weak accuracy. Here, strong completeness means that each correct process eventually suspects each quiet process, and eventual weak accuracy has the same meaning as in the crash model [2].

More recently, Doudou et al. [6] have pointed out that, in presence of arbitrary failures, it is no longer possible to define failure detectors independently of the protocols that rely on them: unlike process crashes, other types of failures are not context-free. To cope with this, the authors have introduced the notion of muteness: a process $p$ is mute to a process $q$ with respect to a given protocol $\mathcal{A}$ if there is a time after which $p$ stops to send to $q$ messages it should send according to $\mathcal{A}$. Note that a process can be mute to some process with respect to a given protocol, without being quiet (in fact, a mute process can continue to send messages that are not part of $\mathcal{A}$). Consequently, these authors have introduced the notion of muteness failure detector, and particularly the class $\diamond M_{\mathcal{A}}$, where $\mathcal{A}$ denotes a particular protocol. This class satisfies Strong Completeness (with respect to mute processes) and Eventual Weak Accuracy. It can be seen as an instantiation of a generic class of omission failure detectors, introduced by Dolev et al. [4]. Moreover, the specification of their muteness failure detector makes sense only in the context of regular round-based distributed algorithms (a precise definition of this class of algorithms is provided in [6]). Kihlstrom et al. [9] noticed that messages need to be certified as well as signed. The signature allows a receiver to verify the sender while the certificate is a well-defined amount of redundant information carried by messages that allows a receiver to check if the content
of a message is valid and if the sending of the message was done “at the right time”. In
other words, a certificate allows a receiver to “look into the sender process” in order to see
if the actions that produced the sending were correct.

Both previous solutions [9,10] solve Strong Consensus, defined by the traditional agree-
ment and termination properties, but with another validity property, named Strong Validity.
It stipulates that, if all correct processes propose the same input value \( v \), then a correct
process that decides must decide on that value. Strong Consensus has a major drawback:
when correct processes propose different values, they are allowed to decide on a value
that is not necessarily connected to their inputs. More generally, the traditional validity
property is not adequate to define the consensus problem in a Byzantine setting. A Byzan-
tine process can initially propose an irrelevant value (i.e., a value different from the one it
should propose) and then behave correctly. There is no way for the other processes to detect
this failure. Consequently, the set of correct processes could agree on an irrelevant value \( v \)
proposed by a process. A significant advance has been done by Doudou and Schiper [5] to
circumvent this drawback: they have introduced a new validity property, namely the Vec-
tor Validity property. In this case, each process proposes a vector which contains a certain
number (at least one) of correct entries. An entry is correct if it is from a correct process.
So, processes have first to construct these vectors. Then, processes agree on one of these
vectors. This problem is called Vector Consensus. They have also shown that Vector Con-
sensus does not suffer from the same drawback as Strong Consensus, in the sense that other
agreement problems (e.g., Atomic Broadcast) can be reduced to Vector Consensus.

In the same papers [5,6] these authors have proposed a protocol that solves Vector Consen-
sus. Let \( F \) be the maximal number of processes that may be faulty (with \( F < n \) where \( n \)
is the total number of processes). Assuming \( F = \lfloor (n - 1)/3 \rfloor \), this protocol satisfies Agree-
ment, Termination and Vector Validity where at least \( F + 1 \) entries of the decided vector
are from correct processes. It is built as an extension of a consensus protocol designed
for the crash failure model [12]. To cope with other Byzantine failures, it uses a muteness
failure detector of class ◻M and properly signed and certified messages. The muteness
failure detector detects only mute processes, and the detection of other Byzantine failures
is imbricated into the protocol itself.

In this paper, we propose a protocol to solve Vector Consensus in a Byzantine asynchro-

ous distributed system. As in [6], this protocol is based on a muteness failure detector of
the class ◻M. However, the detection of other Byzantine failures is not imbricated in the
protocol, but is carried out by a Byzantine behaviour detector. The proposed implementa-
tion of such a detector uses signed and certified messages. Whereas all the aforementioned
protocols [2,5,9] require \( F = \lfloor (n - 1)/3 \rfloor \), ours assumes \( F \leq \min(\lfloor (n - 1)/2 \rfloor, C) \) where
\( C \) is the maximum number of faulty processes the underlying certification service used
by the protocol can cope with.\(^1\) This is due to our approach that clearly separates the con-

\(^1\) In an asynchronous distributed system prone to process crash failures and equipped with a failure
detector of the class ◻W, the consensus problem requires \( F \leq \lfloor (n - 1)/2 \rfloor \) to be solved [2]. In synchronous systems with
Byzantine process failures, the Byzantine general problem requires \( F \leq \lfloor (n - 1)/3 \rfloor \) if messages are not signed
(“oral messages”), and \( F \leq (n - 1) \) if messages are signed [11].

\(^2\) Previously used certification mechanisms require \( C = \lfloor (n - 1)/3 \rfloor \) [2,5,9]. This explains why these works
consider \( F = \lfloor (n - 1)/3 \rfloor \) in their consensus protocols and in their proofs.
The protocol uses as a skeleton an efficient consensus protocol designed by Hurfin and Raynal [8] for the crash failure model, and extends it to cope with the other Byzantine failures. The resulting protocol satisfies Agreement, Termination and Vector Validity with at least $\alpha = n - 2F$ entries from correct processes (note that, due to definition of $F$, we have $\alpha \geq 1$. So, the proposed protocol allows to solve other agreement problems, such as Atomic Broadcast [5]).

The muteness failure detector of class $\sM$, and the Byzantine behaviour detector are composed of detection modules, one for each process. So, each process is actually composed of five modules: (i) a consensus module, (ii) a muteness failure detection module, (iii) a Byzantine behaviour detection module, (iv) a certification module and (v) a signature module. The consensus module executes the protocol. The muteness failure detection module and the Byzantine behaviour detection module are used to reveal mute processes, and processes with other Byzantine behaviours, respectively. If the process is Byzantine, these three modules can behave in a Byzantine way. The other two modules do not behave maliciously. The signature module filters out messages in which the sender must be identified. The certification module manages certificates to be associated with messages. Note that the previous Byzantine consensus protocols leave to the protocol itself the detection of faulty processes not captured by the (unreliable) muteness failure detector. The introduction of a Byzantine behaviour detector, separated from the consensus protocol, is a new approach that provides a general, efficient, modular and simple way to extend distributed protocols to cope with Byzantine failures. In that sense, the contribution of this paper is not only algorithmic and practical (with the design of a new Byzantine consensus protocol), but also methodological.\footnote{A more general methodological approach has been presented by the authors in [1].}

Moreover, an implementation of the Byzantine behaviour detection modules is described. This implementation is based on a set of finite state automata, one for each process. At the operational level, the module associated with a process $p_i$ intercepts all messages sent to $p_i$ from the underlying network and checks if their sendings are done according to the program specification. In the affirmative, it relays the message to $p_i$’s consensus module. The Byzantine behaviour detection module maintains a set ($\text{Byzantine}_i$) of processes it detected to experience at least one Byzantine behaviour. Differently from the muteness failure detector of class $\sM$, the Byzantine behaviour detector is reliable (i.e., Byzantine behaviour detection modules associated with correct processes do not make mistake).

The paper is made of seven sections. Section 2 addresses the consensus problem in a crash failure model. Section 3 presents the Byzantine asynchronous distributed system model. Then Section 4 presents the Byzantine consensus protocol. Section 5 provides a finite state automaton-based implementation of the Byzantine behaviour detection module. Section 6 presents the proof of correctness of the protocol shown in Section 4. Finally, Section 7 concludes the paper.
2. Consensus in the crash failure model

This section considers the consensus problem in a distributed asynchronous system where processes can fail by crashing (i.e., a process behaves correctly until it possibly crashes). So, in this section, a correct process is a process that does not crash. Failure detectors suited to this model [2] and a consensus protocol [8] based on such failure detectors are presented.

2.1. Asynchronous systems

We consider a system consisting of \( n > 1 \) processes \( \Pi = \{p_1, p_2, \ldots, p_n\} \). Each process executes a sequence of statements defined by its program text. Any pair of processes communicate by exchanging messages through reliable\(^4\) and FIFO\(^5\) channels. As the system is asynchronous, there is no assumption about the relative speed of processes or the message transfer delays.

2.2. Consensus

Every correct 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 three following properties [2,7]:

- **Termination**: Every correct process eventually decides on some value.
- **Validity**: If a process decides \( v \), then \( v \) was the initial value proposed by some process.
- **Agreement**: No two correct processes decide differently.

2.3. Failure detectors

Informally, a failure detector consists of a set of modules, each one attached to a process: the module attached to \( p_i \) maintains a set (named suspected\(_i\)) of processes it currently suspects to have crashed. Any failure detection module is inherently unreliable: it can make mistakes by not suspecting a crashed process or by erroneously suspecting a correct one. As in [2], we say “process \( p_i \) suspects process \( p_j \)” at some time \( t \), if at time \( t \) we have \( p_j \in \text{suspected}_i \).

Failure detector classes have been defined by Chandra and Toueg [2] in terms of two properties, namely Completeness and Accuracy. They showed that the weakest failure detectors to solve consensus is \( \star S \) which is based on the following properties:

- **Strong Completeness**: Eventually, every crashed process is permanently suspected by every correct process.
- **Eventual Weak Accuracy**: Eventually, some correct process is never suspected by any correct process.

---

\(^4\) I.e., a message sent by a process \( p_i \) to a process \( p_j \) is eventually received by \( p_j \), if \( p_j \) is correct.

\(^5\) This assumption simplifies the solution when addressing Byzantine failures.
Several protocols solving consensus have been proposed using failure detectors of class $\Diamond S$. These protocols, in the setting proposed in this section, require the majority of processes to be correct (i.e., $F < n/2$, where $F$ is the maximum number of faulty processes).

2.4. Hurfin–Raynal’s Consensus protocol

This section provides a brief description of a version of Hurfin–Raynal’s protocol [8] that assumes FIFO channels (this constraint is not required by the original protocol). As other consensus protocols, this one proceeds in successive asynchronous rounds and uses the rotating coordinator paradigm. During a round, a predetermined process (the round coordinator) tries to impose a value as the decision value. To attain this goal, each process votes: either (vote CURRENT) in favor of the value proposed by the round coordinator (when it has received one), or (vote NEXT) to proceed to the next round and benefit from a new coordinator (when it suspects the current coordinator). Agreement is obtained by a majority principle: a value gets locked as soon as, during a round it has been forwarded by a majority of processes. Moreover, when a value is locked, it will be the decision value. The protocol ensures that all correct processes become aware of this decision value because, once a value $v$ has been locked during a round $r$, then any process entering a round $r' > r$ has $v$ as estimate value.

From an operational point of view, with each process $p_i$ is associated a finite state automaton, whose role is to manage the behaviour of $p_i$ with respect to its vote during the current round. Each of these automata has three states. Let $state_i$ denote the current state of the automaton associated with $p_i$. During a round, the signification of these three states is the following:

- $state_i = q_0$: $p_i$ has not yet voted ($q_0$ is the automaton initial state).
- $state_i = q_1$: $p_i$ has voted CURRENT and has not changed its mind ($p_i$ moves from $q_0$ to $q_1$).
- $state_i = q_2$: $p_i$ has voted NEXT.

Using such an automaton to describe the protocol is important, since the adaptation to the Byzantine failure model, and particularly the implementation of the Byzantine behaviour detection module, will be based on a similar automaton.

Local variables. In addition to the local variable $state_i$, process $p_i$ manages the following four local variables:

- $r_i$ defines the current round number.
- $est_i$ contains the current estimation by $p_i$ of the decision value.
- $nb\_current_i$ (respectively $nb\_next_i$) counts the number of CURRENT (respectively NEXT) votes received by $p_i$ during the current round.
- $rec\_from_i$ is a set composed of the process identities from which $p_i$ has received a (CURRENT or NEXT) vote during the current round.
Finally, suspected, is a set managed by the associated failure detection module (cf. Section 2.3); \( p_i \) can only read this set.

**Automaton transitions.** The protocol manages the progression of each process \( p_i \) within its automaton, according to the following rules. At the beginning of round \( r \), \( \text{state}_i = q_0 \). Then, during \( r \), the transitions are:

- **Transition** \( q_0 \rightarrow q_1 \) (\( p_i \) first votes CURRENT). This transition occurs when \( p_i \), while in the initial state \( q_0 \), receives a CURRENT vote (line 8). This means that \( p_i \) has not previously suspected the round coordinator (line 13). Moreover, when \( p_i \) moves to \( q_1 \) and it is not the current coordinator, it broadcasts a CURRENT vote (line 11).

- **Transition** \( q_0 \rightarrow q_2 \) (\( p_i \) first votes NEXT). This transition occurs when \( p_i \), while in the initial state \( q_0 \), suspects the current coordinator (line 13). This means that \( p_i \) has not previously received a CURRENT vote. Moreover, when \( p_i \) moves to \( q_2 \), it broadcasts a NEXT vote (line 14).

- **Transition** \( q_1 \rightarrow q_2 \) (\( p_i \) changes its mind). This transition (executed by statements at line 18) is used to prevent a possible deadlock. A process \( p_i \) that has issued a CURRENT vote is allowed to change its mind if \( p_i \) has received a (CURRENT or NEXT) vote from a majority of processes (i.e., \(|\text{rec}_i| > n/2|\) but has received neither a majority of CURRENT votes (so it cannot decide), nor a majority of NEXT votes (so it cannot progress to the next round). Then \( p_i \) changes its mind in order to make the protocol progress: it broadcasts a NEXT vote to favor the transition to the next round (line 18).

**Protocol description.** Function \texttt{consensus}() consists of two concurrent tasks. The first task handles the receipt of a DECIDE message (lines 2–3); it ensures that if a correct process \( p_i \) decides (line 3 or line 12), then all correct processes will also receive a DECIDE message. The second task (lines 4–21) describes a round: it consists of a loop that constitutes the core of the protocol. Each (CURRENT or NEXT) vote is labeled with its round number.\(^6\)

- At the beginning of a round \( r \), the current coordinator \( p_c \) proposes its estimate \( v_c \) to become the decision value by broadcasting a CURRENT vote carrying this value (line 6).

- Each time a process \( p_i \) receives a (CURRENT or NEXT) vote, it updates the corresponding counter and the set \texttt{rec}\textsuperscript{from}_i (lines 9 and 16).

- When a process receives a CURRENT vote for the first time, namely, CURRENT (\( p_k, r, est_k \)), it adopts \( est_k \) as its current estimate \( est_i \) (line 10). If, in addition, it is in state \( q_0 \), it moves to state \( q_1 \) (line 11).

- A process \( p_i \) decides on an estimate proposed by the current coordinator as soon as it has received a majority of CURRENT votes, i.e., a majority of votes that agree to conclude during the current round (line 12).

\(^6\) In any round \( r_i \), only votes related to round \( r_i \) can be received. A vote from \( p_k \) related to a past round is discarded and a vote related to a future round \( r_k \) (with \( r_k > r_i \)) is buffered and delivered when \( r_i = r_k \).
• When a process progresses from round $r$ to round $r + 1$ it issues a NEXT (line 20) if it did not do it in the while loop. These NEXT votes are used to prevent other processes from remaining blocked in round $r$ (line 7).

3. Consensus in a Byzantine model

This section first defines what is meant by “Byzantine” behaviour. Then it defines a version of the consensus problem suited to the Byzantine system model. Finally the modules that are part of the system model are described.

3.1. Byzantine processes

A correct process is a process that does not exhibit a Byzantine behaviour. A process is Byzantine if, during its execution, one of the following faults occurs:

- **Crash** The process stops executing statements of its program and halts.
- **Corruption** The process changes arbitrarily the value of a local variable (i.e., arbitrary assignment) with respect to its program specification. This fault could be propagated to other processes by including incorrect values in the content of a message sent by the process.
- **Omission** The process omits to execute a statement of its program (e.g., it omits to send a message, it omits to update a variable, etc.). If a process omits to execute an assignment, this could lead to a corruption fault.
- **Duplication** The process executes more than one time a statement of its program. Note that if a process executes an assignment more than one time, this could lead to a corruption fault.
- **Misevaluation** The process misevaluates an expression included in its program. This fault is different from a corruption fault: misevaluating an expression does not imply the update of the variables involved in the expression and, in some cases (e.g., conditions used in if and loop statements) the result of an evaluation is not assigned to a variable.

Distinction between misevaluation and corruption is extremely important, particularly in the case of conditions. Conditions involving local variables can be misleading because of a corruption fault, even though no misevaluation occurred. For example, in a test like $\text{if } (\text{state}_i = q_0)$ (line 14 of Fig. 1), the value $\text{state}_i$ could have been previously corrupted.

But failures experienced by a Byzantine process cannot be revealed to other processes (and have no consequence on their behaviour) unless the Byzantine process send messages. Thus, the important point is the effect of Byzantine failures and the consequence they can have on the safety and liveness properties of the computation. These effects are of three types:

- **Message corruption**, i.e., inclusion of corrupted value in a message.
- **Message sending omission**, i.e., omission to send a message at the right time.
- **Message sending duplication**.
function consensus(vi)
(1) ri ← 0; esti ← vi;
cobegin
(2) | upon receipt of DECIDE(pk, estk) send DECIDE(p, esti) to Π; return(est)
(3) | loop % on a sequence of asynchronous rounds %
(4) c ← (ri mod n) + 1; ri ← ri + 1; statei ← q0; rec_fromi ← ∅;

(5) if (i = c) then send CURRENT(p, ri, esti) to Π; statei ← q1 endif;

(6) while (nb_nexti ≤ n/2) do % wait until a branch can be selected, and then execute it %
(7) upon receipt of CURRENT(pk, ri, estk) nb_currenti ← nb_currenti + 1; rec_fromi ← rec_fromi ∪ {pk};
(8) if (nb_currenti = 1) then esti ← estk endif;
(9) if (statei = q0) then send CURRENT(p, ri, esti) to Π; statei ← q1 endif;
(10) if (nb_currenti > n/2) then send DECIDE(p, esti) to Π; return(esti) endif

(11) upon (pc ∈ suspectedi) if (statei = q0) then statei ← q2; send NEXT(p, ri) to Π endif
(12) upon receipt of NEXT(pk, ri)
(13) nb_nexti ← nb_nexti + 1; rec_fromi ← rec_fromi ∪ {pk}
(14) if (statei = q0) then statei ← q2; send NEXT(p, ri) to Π endif
(15) if (statei = q1 ∧ |rec_fromi| > n/2) statei ← q2; send NEXT(p, ri) to Π
(16) endwhile
(17) if (statei = q2) then statei ← q2; send NEXT(p, ri) to Π endif
(18) endloop
(19) endcoend

Fig. 1. Hurfin–Raynal’s ⊗S-based Consensus protocol (adapted to FIFO channels).

The role of a Byzantine behaviour detector will be to detect these effects.

3.2. From Consensus to Vector Consensus

Doudou and Schiper have pointed out [5] that in the presence of Byzantine processes the validity property of consensus described in Section 2.2 is not adequate. Hence, they have defined the Vector Consensus problem by using the Agreement and Termination property of Section 2.2 and the following Vector Validity property [5]:

Vector Validity. Every process decides on a vector vect of size n (let v denote the value proposed by p):

- For every process p: if p is correct, then either vect[i] = vi or vect[i] = null, and
- At least α ⩾ 1 elements of vect are initial values of correct processes. 7

So, to solve Vector Consensus, there is a preliminary phase where each process has to build its initial vector, containing at least $\alpha$ initial values of correct processes. Then, processes have to solve the classical consensus problem where each one proposes the vector built in the preliminary phase.

3.3. The class $\Diamond M_A$ of failure detectors

Let us consider the concept of mute process introduced by Doudou et al. in [6]. A process $p_i$ is mute if there exists a time $t$ after which a correct process stops receiving messages from $p_i$ with respect to a particular protocol $A$, i.e., $p_i$ stops sending messages it is supposed to send according to its protocol structure. This includes, but is not limited to, the process crash, where the process stops executing statements of its program and halts.

Considering a mute process as a faulty process, the Completeness property of Section 2.3 is redefined as follows [5]:

**Strong Completeness:** Eventually, every mute process is permanently suspected by every correct process.

A failure detector that satisfies Eventual Weak Accuracy and the definition of Strong Completeness based on the notion of mute process belongs to the $\Diamond M_A$ class. It is interesting to remark that the mute notion captures only some Byzantine behaviours (permanent message omissions). All the other Byzantine behaviours are captured by the Byzantine behaviour detector that will be introduced in the next section. Finally, the specification of a muteness failure detector makes sense only in the context of regular round-based distributed algorithms (a precise definition of this class of algorithms is provided in [6]).

3.4. Additional assumptions on the model

We denote by $F$ the maximum number of Byzantine processes. $F$ is a known bound, and we assume $F \leq \min([\frac{(n-1)}{2}]; C)$ (where $C$ is the maximum number of faulty processes allowed by the certification mechanism). So, at least $n - F \geq \max([\frac{(n+1)}{2}]; n - C)$ processes are correct.

**Key cryptosystem.** Each process $p_i$ possesses a private key and a public key. The private key is used by $p_i$ to sign, in an unforgeable way, outgoing messages. A message $m$ signed by $p_i$ is denoted $\langle m \rangle_i$. Upon the arrival of a signed message at $p_i$, the sender’s public key allows the receiver $p_i$ to verify the identity of the assumed message sender.

**Certificates.** Messages exchanged during the execution of a protocol must be consistent with the protocol specification, i.e., they have to be sent at the right time and have to carry the correct values. The usage of a key cryptosystem is not sufficient to achieve this goal. That is why another tool, called certification, must be used. With each message of the protocol is attached a certificate. A certificate is a well-defined amount of redundant information, including a part of the message sender’s history. Let us denote by $m.\text{cert}$ the certificate attached to a message $m$. We will say that $m.\text{cert}$ is well-formed with respect to a value $v$ contained in $m$ if the value of $v$ can be checked by the certifi-
cate (i.e., by matching \( v \) against the information constituting \( m.cert \)). Similarly, we will say that \( m.cert \) is well-formed if the decision to send \( m \) can be checked by the certificate.

A correct certification service must satisfy the following properties. For each message \( m \) of the protocol:

- **Accuracy**: if the information included in \( m.cert \) is corrupted, then the sender is detected as faulty by a correct receiver.
- **Consistency**: \( m.cert \) is properly formed and well-formed with respect to all values contained in \( m \).

Technically, a correct certification service can be implemented in the following way. Each certificate is a set of signed messages. Accuracy results from the fact that no process can falsify the content of a signed message without being detected as faulty by a correct receiver. Consistency is achieved by selecting appropriate signed messages among those whose receipt is the cause of the sending of \( m \), or whose content has influenced the values included in \( m \). Then, “majority” arguments are used: proper and well-formed certificates are required to contain at least \( n - C \) messages, sufficient to “witness” the content of the message and the occurrence of events enabling the sending of the message (recall that as we assume \( F \leq \min(\lfloor \frac{n-1}{2} \rfloor, C) \), at least \( n - F \geq n - C \) processes are correct). Known certification techniques assume \( n - C = \lceil \frac{2n+1}{3} \rceil \).

We denote as \( \langle m, m.cert \rangle \) a certified and signed message \( m \) sent by \( p_i \). \( \langle m, m.cert \rangle \) is properly formed if its certificate is properly formed.

3.5. Structure of a process

A process consists of five modules whose role has been presented in the introduction: (i) a consensus module, (ii) a Byzantine behaviour detection module, (iii) a muteness failure detection module, (iv) a certification module, and (v) a signature module. More precisely, the structure of a process \( p_i \) is given in Fig. 2. The same figure also shows the path followed by a message \( m \) (respectively \( m' \)) received (respectively sent) by \( p_i \).

**Signature module.** Each message arriving at \( p_i \) is first processed by this module which verifies the signature of the sender (by using its public key). If the signature of the message is inconsistent with the identity field contained in the message, the message is discarded.

![Fig. 2. Structure of a process \( p_i \).](image-url)
and its sender identity (known thanks to the unforgeable signature), is passed to the Byzantine behaviour detection module to be added to the set $\text{Byzantine}_i$. Otherwise, the message is passed to the muteness failure detection module. Also, each message sent by $p_i$ is signed by the signature module just before leaving the process. So, if the sender identity contained in the message is corrupted, this will be discovered by the receiver signature module.

Muteness detection module. This module manages the set $\text{suspected}_i$. It is devoted to the detection of mute processes [5]. The set of modules constitute a muteness failure detector of class $\diamond \mathcal{M}$. It can be implemented by a set of time-outs. Upon the receipt of a message sent by process $p_k$, the signature module of $p_i$ resets the local timer associated with $p_k$ and, if $p_k \in \text{suspected}_i$, removes $p_k$ from that set. Then, the message is passed to $p_i$'s Byzantine behaviour detection module. When the timer associated with $p_k$ expires, $p_k$ is appended to $\text{suspected}_i$. It is important to note that, due to asynchrony, the implementation of the Eventual Weak Accuracy property of $\diamond \mathcal{M}$ can at best be approximate.

Byzantine behaviour detection module. This module receives messages from the muteness detection module and checks if they are properly formed and follow the program specification of the sender. In the affirmative, it passes the message to $p_i$'s certification module. The Byzantine behaviour detection module maintains a set ($\text{Byzantine}_i$) of processes it detected to experience at least one Byzantine behaviour such as duplication, corruption or misevaluation. We say “process $p_i$ declares $p_j$ to be Byzantine” at some time, if, at that time, $p_j \in \text{Byzantine}_i$. Note that differently from the muteness failure detection module, the Byzantine behaviour detection module associated with a correct process $p_i$ is reliable (i.e., if $p_j \in \text{Byzantine}_i$, then $p_j$ has experienced an incorrect behaviour detected by the Byzantine behaviour detection module of $p_j$). Finally, as for the set $\text{suspected}_i$, $p_i$'s consensus module can only read $\text{Byzantine}_i$. Let us note that $\text{Byzantine}_i$ cannot be corrupted by process $p_i$’s consensus module, but, this does not prevent $p_i$’s consensus module to misevaluate an expression involving $\text{Byzantine}_i$ (e.g., $p_j \in \text{Byzantine}_i$ is evaluated to false by $p_i$ even though it was actually true).

Section 5 is devoted to the implementation of the Byzantine behaviour detection module.

Certification module. This module is responsible, upon the receipt of a message from the Byzantine behaviour detection module, for updating the corresponding certificate local variable. It is also in charge to append properly formed certificates to the messages that are sent by $p_i$.

4. The Vector Consensus protocol

Each of the previously proposed protocols solving consensus in Byzantine systems is based on a skeleton protocol solving consensus in a process crash model. [2] and [9] use [2], and [5] uses [12]. We have chosen Hurfin–Raynal’s protocol [8] as a skeleton for our Byzantine consensus protocol because, in addition to its conceptual simplicity, this protocol is particularly efficient when the underlying failure detector makes no mistakes, whether there are failures or not.
4.1. Local variables

Each local variable is a way that a Byzantine process can use to attack correct processes by corrupting its value. Hence, local variables should be used very rarely and their values should be carefully certified.

- $nb_{\textcurrent}_i$ (respectively $nb_{\textnext}_i$) can be replaced by using the cardinality of the certificate $\textcurrent_{\textcert}_i$ (respectively $\textnext_{\textcert}_i$) which contains properly formed CURRENT (respectively NEXT) votes received in the current round.
- $state_i$ can assume three values ($q_0$, $q_1$, $q_2$). Each state can be identified (when necessary) by using certificates in the following way:
  - $state_i = q_0$: no CURRENT vote has been received by $p_i$ and $p_i$ has not sent a NEXT vote. I.e., $|\textcurrent_{\textcert}_i| = 0 \land \langle next(p_i, r_i), cert \rangle_i \notin \textnext_{\textcert}_i$.
  - $state_i = q_1$: a CURRENT vote has been received by $p_i$ and $p_i$ has not sent a NEXT vote: $|\textcurrent_{\textcert}_i| \geq 1 \land \langle next(p_i, r_i), cert \rangle_i \notin \textnext_{\textcert}_i$.
  - $state_i = q_2$: $p_i$ has sent a NEXT vote: $\langle next(p_i, r_i), cert \rangle_i \in \textnext_{\textcert}_i$.
- $\textrec_{\textfrom}_i$ can be replaced by the variable $\textREC_{\textFROM}_i$ using certificates in the following way:

$$\textREC_{\textFROM}_i \equiv \{ p_{\ell} | \langle next(p_{\ell}, r_{\ell}), cert \rangle_{\ell} \in \textnext_{\textcert}_i \lor \langle current(p_{\ell}, est_{\textvect}_{\ell}, r_{\ell}), cert \rangle_{\ell} \in \textcurrent_{\textcert}_i \}.$$  

The predicate change_mind. For the sake of brevity, let $\textchange_mind$ denote the following predicate:

$$|\textcurrent_{\textcert}_i| \geq 1 \land \langle next(p_i, r_i), cert \rangle_i \notin \textnext_{\textcert}_i \land |\textREC_{\textFROM}_i| \geq (n - F).$$

This predicate corresponds to the predicate $(state_i = q_1) \land (|\textrec_{\textfrom}_i| > n/2)$ used in the line 17 of the protocol shown in Fig. 1. Note that the only variables that cannot be replaced by the use of certificates are the round number ($r$), the coordinator ($c$, which directly depends on $r$) and the current estimates ($est_{\textvect}$). Then, their values must be authenticated by certificates as explained below.

4.2. Certificates attached to messages

Four types of messages are exchanged, namely, INIT, CURRENT, NEXT and DECIDE. Each time a message $m$ (INIT, CURRENT, NEXT) is received by the certification module, $m$ is appended to a local variable keeping the corresponding certificates ($est_{\textcert}$, $\textcurrent_{\textcert}$ or $\textnext_{\textcert}$, respectively). These statements are depicted inside a box in the protocol described in Fig. 3.

Each time a process $p_i$ sends a message $m$, according to the type of $m$, an appropriate certificate is associated with $m$ by the certification module. This certificate depends
function consensus(vi)

1. \( \text{next\_cert}_i \leftarrow \emptyset \); \( \text{est\_cert}_i \leftarrow \emptyset \); \( r_i \leftarrow 0 \);
2. \( \text{next\_cert}_i \leftarrow \emptyset \); \( \text{est\_cert}_i \leftarrow \emptyset \); \( r_i \leftarrow 0 \);
3. begin
4. if \( \text{est\_cert}_i \neq (n - F) \) do
5. wait receipt of \( (\text{init}(p_k, v_k), \emptyset)_i \); \( \text{est\_cert}_i \leftarrow \emptyset \);
6. while \( \text{est\_cert}_i \neq (n - F) \) do
7. if \( (i = c) \) then send \( (\text{current}(p_i, r_i, \text{est\_vect}_i), \text{est\_cert}_i \cup \text{next\_cert}_i)_i \); to \( \Pi \); endif;
8. if \( \text{est\_cert}_i = 1 \) then \( \text{est\_cert}_i \leftarrow \text{est\_cert}_i \); \( \text{est\_vect}_i \leftarrow \text{est\_vect}_i \); endif;
9. if \( \text{est\_cert}_i = 1 \) then send \( (\text{current}(p_i, r_i), \text{est\_cert}_i)_i \); to \( \Pi \); endif;
10. if \( \text{est\_cert}_i = 1 \) then send \( (\text{current}(p_i, r_i, \text{est\_vect}_i), \text{est\_cert}_i)_i \); to \( \Pi \); endif;
11. upon receipt of a properly signed and formed \( \langle \text{decide}(p_k, \text{est\_vect}_k), \text{cert}_k \rangle_k \)
12. \( \text{current\_cert}_i \leftarrow \text{current\_cert}_i \cup \text{current}(p_k, r_i, \text{est\_vect}_k, \text{cert}_k)_k \).
13. if \( \text{est\_cert}_i = 1 \) then \( \text{est\_cert}_i \leftarrow \text{est\_cert}_i \); \( \text{est\_vect}_i \leftarrow \text{est\_vect}_i \); endif;
14. if \( \text{est\_cert}_i = 1 \) then send \( (\text{current}(p_i, r_i), \text{est\_cert}_i)_i \); to \( \Pi \); endif;
15. if \( \text{est\_cert}_i = 1 \) then send \( (\text{current}(p_i, r_i, \text{est\_vect}_i), \text{est\_cert}_i)_i \); to \( \Pi \); endif;
16. upon receipt of a properly signed and formed \( \langle \text{current}(p_i, r_i, \text{est\_vect}_i), \text{cert}_k \rangle_k \)
17. if \( \text{current\_cert}_i = 0 \) then \( \text{est\_cert}_i \leftarrow \text{est\_cert}_i \); \( \text{est\_vect}_i \leftarrow \text{est\_vect}_i \); endif;
18. if \( \text{current\_cert}_i = 0 \) then send \( (\text{current}(p_i, r_i), \text{est\_cert}_i)_i \); to \( \Pi \); endif;
19. if \( \text{current\_cert}_i = 0 \) then send \( (\text{current}(p_i, r_i, \text{est\_vect}_i), \text{est\_cert}_i)_i \); to \( \Pi \); endif;
20. upon \( \langle \text{change\_mind} \rangle \) % \( q_1 \rightarrow q_2 \%
21. send \( (\text{next}(p_i, r_i), \text{est\_cert}_i \cup \text{next\_cert}_i)_i \); to \( \Pi \)
22. endwhile
23. if \( \text{next}(p_i, r_i, \text{cert}_i)_i \neq \text{next\_cert}_i \) then send \( (\text{next}(p_i, r_i), \text{next\_cert}_i)_i \); to \( \Pi \) endif % \( q_0/q_1 \rightarrow q_2 \%
24. endwhile
25. coend

Fig. 3. The Consensus module of process \( p_i \).
on the protocol statement that issues the sending of \( m \). This certificate will be used by the Byzantine behaviour detection module of the receiver to check the proper sending of \( m \).

**Certifying initial values for Vector Consensus.** For each process \( p_i \), the first problem lies in obtaining a vector of proposed values (certified vector) that verifies the Vector Validity property (Section 3). This certified vector will then be used as the value proposed by \( p_i \) to the consensus protocol.

This certification procedure is similar to the one proposed by Doudou and Schiper in [5]. It is described in the protocol of Fig. 3 (line 4 to line 9). Each process initially broadcasts its value \( v_i \) and then waits for \( (n - F) \) values from other processes. Each time \( p_i \) receives a value, the message is added to \( \text{est}_i \).

Exiting from the initial while loop (lines 6–9) we say that "\( \text{est}_i \) is well-formed with respect to a value \( \text{est}_i \)" if the following conditions are satisfied:

- \( |\text{est}_i| = (n - F) \) (otherwise process \( p_k \) has either omitted to execute the receipt of line 7 or misevaluated the condition of line 6), and
- the value \( \text{est}_i \) is correct with respect to the \( (n - F) \) INIT messages contained in \( \text{est}_i \). Otherwise process \( p_i \) either has omitted to execute the update of \( \text{est}_i \) (line 8), or has corrupted its value (intuitively, this means that those \( (n - F) \) messages "witness" that \( \text{est}_i \) is a correct value, because \( n - F \geq n - C \)).

**Certifying estimate values.** The initial value of \( \text{est}_i \) (obtained when exiting from lines 4–9) is certified by \( \text{est}_i \), as explained above. This variable can then take successive values: it can be updated at most once per round (line 17) due to the delivery of the first CURRENT message received during this round (line 15). When this occurs, the certificate \( \text{est}_i \) is also updated. Since this message is properly formed, its certificate \( \text{cert}_k \) contains a correct certificate \( \text{est}_i \) (i.e., a certificate well-formed with respect to the value \( \text{est}_i \) contained in the CURRENT message). During a round, \( \text{est}_i \) is said to be "well-formed with respect to \( \text{est}_i \)" if the value \( \text{est}_i \) is the value included in the \( (n - F) \) messages contained in \( \text{est}_i \). Otherwise, it means that process \( p_i \) has either omitted to execute the update of \( \text{est}_i \) (line 17), or corrupted its value.

**From round \( r - 1 \) to round \( r \).** A process progresses from round \( r - 1 \) to round \( r \) when the predicate of line 14 is false. When a new round \( r \) starts, we say that \( \text{next}_i \) is well-formed with respect to \( r - 1 \) if the following two conditions are satisfied:

- \( |\text{next}_i| = (n - F) \) and \( r > 1 \) (otherwise process \( p_i \) has misevaluated the condition of line 14).
- The value \( r - 1 \) is consistent with respect to the information in the \( (n - F) \) NEXT messages contained in \( \text{est}_i \) (i.e., all messages refer to round \( r - 1 \). Otherwise process \( p_k \) has corrupted the value of \( r \) at line 11).
- If \( r = 1 \), then \( \text{next}_i = \emptyset \) (otherwise either \( p_i \) has corrupted \( r \) or \( c \) (line 11) or has misevaluated the condition of line 12).
4.3. Protocol description

The text of the protocol is presented in Fig. 3. In the following, a vote means a message CURRENT or NEXT, and a valid vote means a properly signed and formed vote.

- At the beginning of a round \( r \), the current coordinator \( p_c \) proposes its estimate \( \text{est}_v \) to become the decision value by broadcasting a CURRENT vote carrying this value (line 12). This vote is certified by \( \text{est}_\text{cert}_c \cup \text{next}_\text{cert}_c \). \( \text{est}_\text{cert}_c \) is used to certify the value proposed by the coordinator \( \text{est}_v \). I.e., \( \text{est}_\text{cert}_c \) must be well formed with respect to \( \text{est}_v \). \( \text{next}_\text{cert}_c \) is used to certify the value of the current round \( r \) (i.e., \( \text{next}_\text{cert}_c \) must be well formed with respect to \( r - 1 \)).
- When \( p_i \) receives the first CURRENT valid vote while in state \( q_0 \) (line 17). It relays a CURRENT vote (line 19) by using the valid vote CURRENT just received as a certificate. This certificate contains \( \text{est}_\text{cert}_c \cup \text{next}_\text{cert}_c \) used to certify \( r \) and \( \text{est}_v \) (as above).
- If, while it is in the initial state \( q_0 \), \( p_i \) suspects the current coordinator, it broadcasts a NEXT vote (line 24) and moves to \( q_2 \). This vote is certified by \( \text{est}_\text{cert}_i \cup \text{current}_\text{cert}_i \cup \text{next}_\text{cert}_i \). Those certificates (\( \text{current}_\text{cert}_i \) and \( \text{next}_\text{cert}_i \)) will be used by the Byzantine behaviour detection module of the receiver to decide whether \( p_i \) has misevaluated or not the sending condition (at line 23). Moreover, as NEXT votes can also be sent at lines 29 and 31, \( \text{est}_\text{cert}_i \) is used to allow the receiver to determine the condition that has triggered the NEXT vote it receives.
- When the predicate at line 28 becomes true, in order to avoid a deadlock, process \( p_i \) broadcasts a NEXT vote to favor the transition to the next round (line 29). This vote is certified by \( \text{current}_\text{cert}_i \cup \text{next}_\text{cert}_i \). Those certificates are used to certify the non-misevaluation of the predicate \( \text{change_mind} \).
- When a process progresses from round \( r \) to round \( r + 1 \) it issues a NEXT vote if it did not do it in the while loop. These NEXT votes are used to prevent other processes from remaining blocked in round \( r \) (line 31). This vote is certified by \( \text{next}_\text{cert}_i \) which will allow a receiver to check the correct evaluation of the condition at line 14 by verifying if \( \text{next}_\text{cert}_i \) is well formed with respect to \( r \).

4.4. Taking a decision

A process decides a value \( \text{est}_v \) at round \( r \) either when it has received \( (n - F) \) valid CURRENT votes (lines 20–21) or when it receives a properly signed and formed DECIDE message from another process (lines 2–3). In the first case the process authenticates its decision by using a well-formed \( \text{current}_\text{cert}_i \) as a certificate (line 21), in the second case the message (with the same certificate) is relayed to the other processes (line 3).

We say that \( \text{current}_\text{cert}_i \) is well formed with respect to \( r \) and \( \text{est}_v \) if the following conditions are satisfied:

- \( |\text{current}_\text{cert}_i| = (n - F) \) (otherwise process \( p_i \) misevaluated the condition of line 20);
- the certificate of each message in \( \text{current}_\text{cert}_i \) contains a \( \text{est}_\text{cert}_c \) well formed with respect to \( \text{est}_v \);
the certificate of each message in current_cert_k contains a next_cert well formed with respect to r.

4.5. Summary

Thanks to the verifications made by the Signature verification module and the Byzantine behaviour detection module, we have, for every correct process:

- Every accepted INIT(pk, v_k) message has a correct field pk (signature module).
- Every accepted CURRENT(pk, r, est_vect_k) message has correct fields pk (Signature verification module), r (certificate next_cert_k) and est_vect_k (certificate est_cert_k contained in current_cert_k).
- Every accepted NEXT(pk, r) message has correct fields pk (signature module) and r (certificate next_cert_k).
- Every accepted DECIDE(pk, est_vect_k) message has correct fields pk (Signature verification module), and est_vect_k (certificates contained in current_cert_k).

5. The Byzantine behaviour detection module

The Byzantine behaviour detection module of process p_i is composed of a set of finite state automata, one for each process. The module associated with p_i is depicted in Fig. 4. This module detects if a given process p_k has shown a Byzantine behaviour. This detection is carried out in two steps. (i) It is first checked if a certificate of a message is well formed with respect to the value carried by this message. It is then (ii) checked if the type of the message follows the program specification (i.e., if it has been sent at the right time). If one of these steps detects something wrong, the automaton falls in the state Byz which means

Fig. 4. Finite state automaton of the p_i’s Byzantine behaviour detection module to monitor process p_j.
$p_k$ is Byzantine, furthermore, the message is discarded. Otherwise, the message is passed to the consensus module of process $p_i$. So, the detection of the three types of Byzantine behaviour effects, introduced Section 3.1, is ensured:

- Message corruption is detected by the certification mechanism (the sender’s identity has been checked by the signature module);
- Message sending omission or duplication are detected by the automaton, designed accordingly to the protocol specification (the FIFO channel communication assumption facilitates this detection).

The automaton of $p_i$ related to $p_k$ represents the view $p_i$ has, during the current round $r_i$, on the behaviour of $p_k$ with respect to its automaton (made of states $q_0, q_1$ and $q_2$, see Sections 2 and 4) during the same round. It evaluates a condition each time a message has arrived from $p_k$. According to the result of this evaluation, it moves from its current state $a$ to another state $b$. The condition is composed of the following predicates.

Automaton states. The automaton of process $p_i$ related to $p_k$ is composed of 6 states. Three states are related to a single round (as the ones described in the previous section: $q_0, q_1, q_2$), one is the initial state (start), one is the final state (final) and one is the state declaring $p_k$ is Byzantine ($Byz$). Note that the predicate $p_k \in Byzantine$ is true if the automata related to $p_k$ of process $p_i$ is in state $Byz$. It is false otherwise.

Automaton transitions.

- **Transition start $\rightarrow q_0$.** It occurs as soon as an INIT message is received from $p_k$, i.e., $init_k$ becomes true.
- **Transition start $\rightarrow Byz$.** It occurs when $p_i$ receives any other message from $p_k$ (e.g., CURRENT, NEXT or DECIDE) before INIT. Due to the FIFO property of channels, $p_i$ concludes that $p_k$ is Byzantine as it omitted to execute line 5 and moves to state $Byz$.
- **Transition $q_0 \rightarrow q_1$.** The Byzantine behaviour detection module of process $p_i$ receives $\langle current(p_k, r_i, est_vect_k), cert_k \rangle_k$ from $p_k$. We have two cases:
  - **Case 1.** $p_k$ is the coordinator (the message was sent by $p_k$ from line 12). I.e., $cert_k = est_cert_k \cup next_cert_k$. The predicate $PF_{0,1}(type_of_message_k)$ returns true if the message is properly formed. The predicate $type_of_message_k$ is true if a message of that type has been received from $p_k$.
  - **Case 2.** $p_k$ is not the coordinator (the message was sent by $p_k$ from line 19). I.e., $cert_k = current_cert_k$. According to the protocol, the certificate must include the message $\langle current(p_c, est_vect_c, r_i), cert \rangle_c$ in order to be properly formed (line 16), then, once extracted this message from the certificate, all tests of case 1 can be executed on $est_cert_c \cup next_cert_c$.
- **Transition $q_0 \rightarrow q_2$.** Upon the arrival of a $\langle next(p_k, r_i), cert_k \rangle_k$ at the Byzantine behaviour detection module, we have two cases:
  - **Case 1.** If $cert_k$ does not contain any INIT message, it was sent by $p_k$ from line 31 where $cert_k = next_cert_k$. In such a case $PF_{0,2}(next)$ is true if $cert_k$ is well formed with respect to $r_i - 1$. Otherwise there was a misevaluation of the condition at line 14.
Case 2. If \( \text{cert}_k \) contains at least one \textsc{init} message, the \textsc{next} message was sent by \( p_k \) from line 24 with \( \text{cert}_k = \text{current\_cert}_k \cup \text{next\_cert}_k \cup \text{est\_cert}_k \). This sending is guarded by the conditions at line 23. \( PF_{0,2}(\text{next}) \) is true, if by using the information contained in \( \text{cert}_k \), \( \text{est\_cert}_k \) is well-formed with respect to a value \( \text{est\_vect}_k \) and the conditions at line 23 (i.e., \( \langle |\text{current\_cert}_k| = 0 \rangle \land \langle \text{next}(p_k, r_i), \text{cert}_k \notin \text{next\_cert}_k \rangle \)) can be evaluated to true.

Remark. Note that even though \( PF_{0,2}(\text{next}) \) is true, the process \( p_k \) could be Byzantine as it could misevaluate the predicate \( p_c \in (\text{suspected}, \lor \text{Byzantine}) \) used at line 22 and no information about that fault can be found in the certificate. One solution to this problem is to say that, as at most \( F \) processes are Byzantine, then even though all of them misevaluate that predicate and generate false \textsc{next} messages, this does not prevent other processes to get a decision due to conditions of line 14.

- Transition \( q_0 \rightarrow \text{Byz} \). Upon the arrival of a message, \( p_i \) declares \( p_k \) Byzantine, if one of the following four conditions is true: (i) \( PF_{0,1}(\text{current}) \) is false (in the case of the receipt of a \textsc{current} message); or (ii) \( PF_{0,2}(\text{next}) \) is false (in the case of the receipt of a \textsc{next} message); or (iii) the message is of type \textsc{decide}; or (iv) the message is of type \textsc{init}. In the case (iii), as channels are FIFO, a \textsc{next} or \textsc{decide} message has been omitted by \( p_k \). In the case (iv) process \( p_k \) executed twice the statement at line 5.

- Transition \( q_1 \rightarrow q_2 \). Upon the arrival of a \( \langle \text{next}(p_k, r_i), \text{cert}_k \rangle \) from \( p_k \) at the Byzantine behaviour detection module, it executes the following tests on the certificate \( \text{cert}_k \). The \textsc{next} message was sent by \( p_k \) from line 24 with \( \text{cert}_k = \text{current\_cert}_k \cup \text{next\_cert}_k \cup \text{est\_cert}_k \). That sending is guarded by the condition at line 23. \( PF_{1,2}(\text{next}) \) is true if, using the information contained in \( \text{cert}_k \), the predicate \text{change\_mind} can be evaluated to true. As above remark the fact that \( PF_{1,2}(\text{next}) \) is true does not imply that the sender process is correct, as the condition at line 28 includes the predicate \( p_c \in (\text{suspected}, \lor \text{Byzantine}) \) that can be misevaluated by \( p_k \) without being detected.

- Transition \( q_1 \rightarrow \text{final and transition } q_2 \rightarrow \text{final} \). These transitions occur upon the arrival of a message \( \langle \text{decide}(p_k, \text{est\_vect}_k), \text{cert}_k \rangle \) at the Byzantine behaviour detection module. \( PF_{1,2}(\text{decide}) \) (respectively \( PF_{1,\text{final}}(\text{decide}) \)) is true if \( \text{cert}_k \) is well formed with respect to \( \text{est\_vect}_k \) and \( r_i \).

- Transition \( q_1 \rightarrow \text{Byz} \). Upon the receipt of a message, \( p_i \) declares \( p_k \) Byzantine, if one of the following three conditions is true: (i) \( PF_{1,2}(\text{next}_k) \) is false (in the case of the receipt of a \textsc{next} message); or (ii) \( PF_{1,\text{final}}(\text{decide}_k) \) is false (in the case of the receipt of a \textsc{decide} message); or (iii) the message is of type \textsc{current}. In the latter case, as channels are FIFO, a \textsc{next} or \textsc{decide} message should be received by \( p_i \) from \( p_k \), then \( p_k \) had a Byzantine behaviour.

- Transition \( q_2 \rightarrow \text{Byz} \). Upon the receipt of a message, \( p_i \) declares \( p_k \) Byzantine, if one of the following three conditions is true: (i) \( PF_{2,\text{final}}(\text{decide}_k) \) is false (in the case of the receipt of a \textsc{decide} message); or (ii) the message is of type \textsc{current}; or (iv) the message is of type \textsc{next}. In the latter two cases, as channels are FIFO only a \textsc{decide} message should be received by \( p_i \) from \( p_k \), if a then message of type \textsc{current} or \textsc{next} is received from \( p_k \), it had a Byzantine behaviour.
• Transition \( \text{Byz} \rightarrow \text{Byz} \). Once \( p_k \) is declared Byzantine it remains in the state \( \text{Byz} \) whatever type of message is received.

• Transition \( q_2 \rightarrow q_0 \) (denoted by a dotted line). A process \( p_k \) in \( p_2 \) either decides a value, and then moves to final or moves to \( q_0 \) (and then to the successive round) in a finite time. The fact that \( p_k \) moved to \( q_0 \) can be detected if the successive message received by the process from \( p_k \) is either a \text{NEXT} or a \text{CURRENT} message related to round \( r + 1 \).

6. Correctness proof

This section proves that the previous protocol satisfies the Termination, Vector Validity and Agreement properties. This proof assumes that:

• H1: There are at most \( F \) processes that do not behave correctly, with \( F \leq \min\left(\left\lfloor \frac{n-1}{2} \right\rfloor, C\right) \).

• H2: The underlying muteness failure detector belongs to the class \( \Diamond \mathcal{M} \), i.e., it satisfies:
  – H2.1: Strong Completeness (eventually, every mute process is permanently suspected by every correct process). And,
  – H2.2: Eventual Weak Accuracy (eventually, some correct process is never suspected by any correct process).

• H3: The underlying Byzantine behaviour detector is reliable (if \( p_i \) is correct and if \( p_j \in \text{Byzantine}_i \) then \( p_j \) has experienced at least one Byzantine behaviour).

• H4: Communication channels are \text{FIFO}.

6.1. Vector Validity

Lemma 6.1. Eventually, every correct process builds a vector \( \text{est\_vect}_i \) such that

\[
\forall k \neq i : \text{est\_vect}_i[k] = v_k \text{ or } \text{est\_vect}_i[k] = \text{null}.
\]

Proof. From assumptions H1 and H4, the while loop eventually terminates (line 9) and the values contained in \( \text{est\_vect}_i \) are either \text{null} (line 4) or set at line 8. \( \square \)

Lemma 6.2. During any round \( r \), for any process \( p_i \), we have: \( |\text{current\_cert}_i| > 0 \Rightarrow \text{est\_vect}_i = \text{est\_vect}_c \), where \( p_c \) is the current coordinator. In particular, \( \text{est\_vect}_i[i] = v_i \text{ or } \text{est\_vect}_i[i] = \text{null} \).

Proof. case \( i = c \). \( \text{est\_vect}_c \) is updated with the value \( \text{est\_vect}_c \) at most once in the current round, at line 17.

case \( i \neq c \). \( \text{est\_vect}_i \) is updated at most once per round, at line 17, when \( p_i \) receives for the first time a valid vote \text{CURRENT}. Let \( p_k \) be the sender of this vote.

• If \( k = c \) then \( \text{est\_vect}_i \leftarrow \text{est\_vect}_c = \text{est\_vect}_c \)
If \( k \neq c \) let us examine the sequence of votes CURRENT leading to the update of line 17:
- Every process \( p_j (j \neq c) \) sending a vote CURRENT (at line 19) has received a valid vote \( \langle \text{current}, \text{vect}, - \rangle \) and has performed line 17: \( \text{est}_\text{vect}_j \leftarrow \text{vect} \)
- Only \( p_c \) can initiate such a sequence of CURRENT messages.

From this follows that \( \text{est}_\text{vect}_i = \text{est}_\text{vect}_j = \cdots = \text{est}_\text{vect}_j = \cdots = \text{est}_\text{vect}_c \). Thus, in particular, \( \text{est}_\text{vect}_i[i] = \text{est}_\text{vect}_c[i] \) and this value is either \( v_i \) or null, from Lemma 6.1.

**Lemma 6.3.** No process can build two different initial certified vectors.

**Proof.** During the initial phase (lines 6–9) a process \( p_i \) can receive at most one INIT message from each process \( p_k \). In fact, INIT messages are signed and the Byzantine behaviour detection module filters out duplicate messages. Suppose that \( p_i \) builds two different initial certified vectors \( \text{est}_\text{vect}_1 \) and \( \text{est}_\text{vect}_2 \) (with \( \text{est}_\text{vect}_1 \neq \text{est}_\text{vect}_2 \)). In that case, \( p_i \) has two certificates \( \text{est}_\text{cert}_1 \) and \( \text{est}_\text{cert}_2 \) well-formed respectively with respect to \( \text{est}_\text{vect}_1 \) and \( \text{est}_\text{vect}_2 \). In particular, \( |\text{est}_\text{cert}_1| = |\text{est}_\text{cert}_2| = (n-F) \). Each of these certificates contains \((n-F)\) identities of processes, namely the senders of the \((n-F)\) signed INIT messages forming these certificates. Let \( X \) (respectively \( X' \)) denote the set of process identities belonging to \( \text{cert}_1 \) (respectively \( \text{cert}_2 \)). By assumption, \(|X| = |X'| = (n-F)\). On the other hand, \(|X \cap X'| = |X| + |X'| - |X \cup X'|\), and thus \(|X \cap X'| \geq 2(n-F) - n = n - 2F\). From assumption \( H1 \), we get \(n - 2F \geq 1\), which means \(|X \cap X'| \neq \emptyset\). Thus, \( p_i \) has received two INIT messages from a same sender. A contradiction. \( \Box \)

**Lemma 6.4.** A correct process sending a message DECIDE (whose initial sending has been initiated at round \( r \), line 21) decides \( \text{est}_\text{vect}_c \) (where \( p_c \) is the coordinator of round \( r \)).

**Proof.** A process decides at line 3 or at line 21.

- If \( p_i \) decides at line 21, then \(|\text{current}_c| > 0\) and thus, from Lemma 6.2, \( p_i \) decides \( \text{est}_\text{vect}_c = \text{est}_\text{vect}_c \). Before deciding, it sends a message DECIDE with the value \( \text{est}_\text{vect}_c \) to all other processes.
- If \( p_i \) decides at line 3, it decides the value contained in the properly signed and formed message DECIDE. This message contains \( \text{est}_\text{vect}_c \). \( \Box \)

**Theorem 6.5.** If a correct process decides, it is on a vector \( v \) of size \( n \), satisfying Vector Validity with at least \( \alpha = n - 2F \geq 1 \) entries from correct processes.

**Proof.** Every process builds a vector (Lemma 6.1), containing \((n-F)\) values from different processes (lines 6–9) and \( F \) null values (line 4). If a process decides at round \( r \), then it decides \( \text{est}_\text{vect}_c \) (Lemma 6.4). If \( p_c \) has sent a CURRENT vote, \( \text{est}_\text{cert}_c \) is well-formed with respect to \( \text{est}_\text{vect}_c \). In particular, \( \text{est}_\text{vect}_c \) is correct with respect to the \((n-F)\) INIT messages contained in \( \text{cert}_\text{init}_c \). So, \( \text{est}_\text{vect}_c \) contains \((n-F)\) initial values of different processes. Moreover, from Lemma 6.1, \( \text{est}_\text{vect}_c[i] = v_i \) or \( \text{est}_\text{vect}_c[i] = \text{null} \).
Let $nb_{\text{correct}}_c$ (respectively $nb_{\text{faulty}}_c$) denote the number of non-null entries of $est_{\text{vect}}_c$ that are initial values from correct (respectively non correct) processes. We have $nb_{\text{correct}}_c + nb_{\text{faulty}}_c = (n - F)$. But $nb_{\text{faulty}}_c \leq F$ and thus $nb_{\text{correct}}_c \geq n - 2F$.

From assumption H1, we have $n - 2F \geq n - 2\left\lceil \frac{n-1}{2} \right\rceil \geq 1$. \hfill \Box

6.2. Termination

Lemma 6.6. If a correct process decides, then eventually every correct process will decide.

Proof. Let $p_i$ deciding at line 3 or at line 21. In both cases, $p_i$ sends a properly signed and formed message $\text{DECIDE}$ to every other process. From assumption H4, every correct process that did not yet decide eventually receive this message (line 2) and decides (line 3). \Box

Lemma 6.7. If no process decides during any round $r' \leq r$, then all correct processes start round $r + 1$.

Proof. The proof is by contradiction. Suppose no process has decided in any round $r' \leq r$, where $r$ is the smallest round number in which some correct process blocks forever in the while loop (lines 14–32). Let us note that no correct process has received a $\text{DECIDE}$ message (otherwise, it would execute lines 2–3 and decide). In the following proof, we use the following notations, recalling the “hidden” states $q_0, q_1, q_2$ of the processes (cf. Section 4.1):

\begin{align*}
\text{state } q_0 &\equiv (|current_{cert}_i| = 0) \wedge \langle next(p_i, r_i), cert \rangle_i \notin next_{cert}_i, \\
\text{state } q_1 &\equiv (|current_{cert}_i| \geq 1) \wedge \langle next(p_i, r_i), cert \rangle_i \notin next_{cert}_i, \\
\text{state } q_2 &\equiv \langle next(p_i, r_i), cert \rangle_i \in next_{cert}_i.
\end{align*}

(1) Note first that any process starting round $r$ is in state $q_0$ before entering the while loop (lines 11 and 13). Firstly, it is shown that a correct process $p_i$ cannot remain in state $q_0$ during round $r$. This follows from:

(i) Either $p_i$ suspects $p_c$ and moves to $q_2$ (lines 22–25). This is due either to the underlying muteness failure detection module (possibly mistaken), or because $p_c$ is detected as Byzantine by the underlying Byzantine behaviour detection module.

(ii) Or $p_i$ never suspects $p_c$. Due to assumption H2.1 (Strong Completeness), this means that $p_c$ is not quiet for $p_i$: eventually it broadcasts a $\text{CURRENT}$ vote. Due to assumption H4, $p_i$ receives at least one $\text{CURRENT}$ valid vote (from $p_c$ or from another process) and moves to $q_1$ (line 15).

(2) A correct process cannot remain in state $q_1$. By contradiction: suppose that a correct process $p_i$ remains in state $q_1$. From the previous point, any correct process either sends a $\text{CURRENT}$ vote and moves to $q_1$ (case ii), or sends a $\text{NEXT}$ vote and moves to $q_2$ (case i). It follows from assumption H1 that any correct process will receive at least $(n - F)$ valid votes and thus eventually $|REC_{FROM}| \geq (n - F)$. Thus $p_i$ eventually satisfies $\text{change\_mind}$. Hence, the condition stated at line 28 eventually becomes true: $p_i$ issues a $\text{NEXT}$ vote and moves to $q_2$. 


(3) It follows from the two previous points that all correct processes move to $q_2$ and send a 
NEXT vote. Consequently, any correct process $p_i$ receives at least $(n - F)$ NEXT valid 
votes, and thus proceeds to round $r + 1$. A contradiction. □

**Lemma 6.8.** During a round $r$, if no process suspects the coordinator (or, equivalently, 
no process moves from state $q_0$ directly to state $q_2$) within the while loop, then no process 
sends a NEXT vote (or equivalently, no process moves to $q_2$).

**Proof.** From the lemma assumption, no process executes lines 22–25 (going to state $q_2$). 
Suppose that, during round $r$, a process sends a NEXT vote. So, there exists a process 
that has sent a NEXT vote before receiving a NEXT valid vote. Such a process $p_i$ executes 
line 29 or line 31.

- $p_i$ executes line 29. This is possible only because $change\_mind$ has become true 
(line 28), i.e., $p_i$ has received a (CURRENT or NEXT) valid vote from at least $(n - F)$ 
processes. If one of these votes is a NEXT, this contradicts the fact that $p_i$ has sent 
a NEXT vote before receiving a NEXT valid vote. So, all these valid votes are CURRENT. Thus, $p_i$ has decided at line 21 when it received the last valid vote making true 
the condition $|current\_cert| = (n - F)$. Consequently, as $p_i$ has executed a return 
statement it will never execute line 29. Contradiction.

- $p_i$ executes line 31. To send a NEXT vote at line 31, $p_i$ has terminated its while loop. 
So, it has $|next\_cert| > F$. It follows that $p_i$ has received NEXT valid votes before 
nexecuting line 31. Contradiction. □

**Theorem 6.9.** Every correct process eventually decides some value.

**Proof.** Let us consider the two following cases.

(1) A correct process $p_j$ decides (at line 21). From Lemma 6.6, every correct process 
eventually decides.

(2) No process decides. We will show there is a contradiction. There is a time $t$ after which 
(due to assumption H2.2, Eventual Weak Accuracy) there is a correct process that is 
no longer suspected (let $p_j$ be this process). Let $r$ be the first round that occurs after $t$ 
and which is coordinated by $p_j$ (due to Lemma 6.7, such a round does exist since no 
process decides).

- The coordinator $p_j$ sends a CURRENT vote to all processes.

- As, by assumption, the current coordinator $p_j$ is not suspected, no process $p_i$ executes 
lines 22–25. Consequently, no process moves directly from $q_0$ to $q_2$ within 
the while loop.

- From Lemma 6.8, we conclude that no process sends a NEXT vote, i.e., no process 
moves to $q_2$.

- A process entering the loop while (line 14) cannot exit this loop, because 
$|next\_cert|$ remains equal to 0.

- From assumptions H1 and H4, each correct process $p_i$ will receive at least $(n - F)$ CURRENT valid votes. As no process sends a NEXT vote, it follows that
any correct process \( p_i \) will necessarily decide at line 21. This contradicts the assumption that no process decides. \( \square \)

6.3. Agreement

**Lemma 6.10.** If a process decides \( v \) and broadcasts a properly signed and formed \( \text{DECIDE} \) message at line 21 of round \( r \), then all correct processes \( p_j \) that start round \( r+1 \) do so with \( \text{estvect}_j = v \).

**Proof.**

1. As \( p_i \) broadcasts a message \( \text{DECIDE} \) labeled with the round number \( r \), some process \( p_k \) (possibly \( p_i \)) has processed the line 21 during round \( r \) and, consequently, \( |\text{currentcert}_k| = (n - F) \) (R1). Moreover, from Lemma 6.4, the decision value \( v \) is equal to \( \text{estvect}_c \).

2. Let us consider the three following sets of processes (related to round \( r \)):
   \[
   \begin{align*}
   X_{1}^r &= \{ \text{processes that moved from } q_0 \text{ to } q_1 \text{ and did not move to } q_2 \}, \\
   X_{2}^r &= \{ \text{processes that moved from } q_0 \text{ directly to } q_2 \}, \\
   X_{3}^r &= \{ \text{processes that moved from } q_0 \text{ to } q_1 \text{ and then to } q_2 \}.
   \end{align*}
   \]

   Note that these sets are disjoint. They include processes that have possibly behaved incorrectly after moving from \( q_0 \) to another state. Moreover, we have \( |X_{1}^r| + |X_{2}^r| + |X_{3}^r| \leq n \) (R2).

3. Let \( \text{sent}_p \) be the number of processes that sent a \( \text{CURRENT} \) vote to \( p_k \). As the number of \( \text{CURRENT} \) votes sent to a \( p_k \) is greater than or equal to the number of \( \text{CURRENT} \) valid votes received by \( p_k \), we have \( \text{sent}_p \geq |\text{currentcert}_k| \) (R3).

4. All processes belonging to \( X_{3}^r \) have sent a \( \text{CURRENT} \) vote to \( p_k \) at line 19 (when they moved from \( q_0 \) to \( q_1 \)). Moreover, all processes belonging to \( X_{1}^r \) and which executed all of line 19 have sent a \( \text{CURRENT} \) vote to \( p_k \). Some processes belonging to \( X_{1}^r \) and which have partially executed line 19 have also sent a \( \text{CURRENT} \) vote to \( p_k \). Finally, processes in \( X_{2}^r \) have not sent a \( \text{CURRENT} \) vote. It follows that \( \text{sent}_p \leq |X_{1}^r| + |X_{3}^r| \) (R4).

5. From (R1) and (R3), we conclude \( \text{sent}_p \geq (n - F) \) (R5).

6. From (R5) and (R4), we conclude \( |X_{1}^r| + |X_{3}^r| \geq (n - F) \) (R6).

7. From (R6) and (R2), we conclude \( |X_{2}^r| \leq F \) (R7).

The proof is now by contradiction. Suppose that \( p_i \) decides (and consequently (R7) holds), and that there is a process \( p_j \) which enters round \( r+1 \) with \( \text{estvect}_j \neq v \) (i.e., \( \text{estvect}_j \neq \text{estvect}_c \)). Let us consider the value \( |\text{currentcert}_j| \) just before \( p_j \) leaves round \( r \). There are two cases.

1. \( |\text{currentcert}_j| > 0 \). From Lemma 6.2, we have \( \text{estvect}_j = \text{estvect}_c \), a contradiction.

2. \( |\text{currentcert}_j| = 0 \). In this case, since, according to the lemma assumption, \( p_j \) proceeds to the next round, we have \( |\text{nextcert}_j| > F \) (R8), at the end of round \( r \). Combining (R7) and (R8), we get \( |\text{nextcert}_j| > |X_{2}^r| \) (R9).
Since NEXT votes are only sent by processes belonging to \( X_r^2 \cup X_r^3 \), and as (by definition) \( X_r^2 \cap X_r^3 = \emptyset \), from (R9) we conclude that \( p_j \) received at least one NEXT valid vote from a process \( p_\ell \) belonging to \( X_r^3 \). As \( p_\ell \) belongs to \( X_r^3 \):

- \( \ell \neq c \) and \( p_\ell \) first passed through state \( q_1 \). So, it processed lines 17–19, from which we conclude \(|current\_cert_\ell| > 0\) and, from Lemma 6.2, \( est\_vect_\ell = est\_vect_c \). In particular, \( p_\ell \) has sent a CURRENT vote to \( p_j \).
- \( p_\ell \) then moved from \( q_1 \) to \( q_2 \). So, it necessarily sent the NEXT vote (received by \( p_j \)) at line 29.

From H4, \( p_j \) has received from \( p_\ell \) the \( \langle current(p_\ell, r_\ell, est\_vect_\ell), current\_cert_\ell \rangle_\ell \) valid vote first (line 15), then the \( \langle next(p_\ell, r_\ell), current\_cert_\ell \cup next\_cert_\ell \rangle_\ell \) (line 26). Upon the receipt of the CURRENT valid vote, the condition of line 17 holds and \( p_j \) has updated \( est\_vect_j \) to \( est\_vect_\ell = est\_vect_c \). Upon the receipt of the NEXT valid vote, no update of \( est\_vect_j \) occurred. So, just before \( p_j \) leaves round \( r \), \( est\_vect_j = est\_vect_c \). A contradiction. \( \square \)

**Theorem 6.11.** No two correct processes decide different values.

**Proof.** Let \( p_i \) and \( p_j \) two correct processes that decide. There are two cases:

1. Both \( p_i \) and \( p_j \) decide at the same round \( r \) (both execute the line 21 during the round \( r \)). Let \( est\_vect_c \) denote the certified initial vector built by the coordinator \( p_c \) of round \( r \) (Lemma 6.1). From Lemma 6.4, a process that decides at round \( r \) decides the value \( est\_vect_c \). From Lemma 6.3, the coordinator cannot build two different certified initial vectors. So, \( p_i \) and \( p_j \) decide the same value \( est\_vect_c \).
2. \( p_i \) decides \( v \) at round \( r \) and \( p_j \) decides at round \( r' > r \). From Lemma 6.10, every correct process \( p_k \) that starts round \( r + 1 \) does so with \( est\_vect_k = v \). In other words, the only possible estimate value for a correct process participating in any round \( r' \geq r + 1 \), is now \( v = est\_vect_c \). So, whatever the coordinator of round \( r' \), due to Lemma 6.4, the value decided by \( p_j \) will be \( v = est\_vect_c \). \( \square \)

7. Conclusion

The paper has presented a protocol that solves the (vector) consensus problem in a Byzantine system. The proposed protocol is resilient to \( F \) faulty processes, \( F \leq \min([\lfloor (n - 1)/2 \rfloor], C) \) (where \( C \) is the maximum number of faulty processes the underlying certification mechanism can tolerate). As it ensures the Vector Validity property, it can be used as a building block to solve other agreement problems, such as Atomic Broadcast [5]. The paper has also presented an implementation of the Byzantine behaviour detection module. This implementation is based on a set of finite state automata. An additional interest of the proposed protocol lies in its systematic use of certificates associated with messages. This facilitates the implementation of the Byzantine behaviour detection module. Moreover, re-
ducing the number of local variables, also reduces their incorrect management by faulty processes.

References

[1] R. Baldoni, J.M. Helary, M. Raynal, From crash-fault tolerance to arbitrary fault tolerance: towards a mod-
267.
685–722.
[4] D. Dolev, R. Friedman, I. Keidar, D. Malkhi, Failure detectors in omission failure environments. Brief an-
[5] A. Doudou, A. Schiper, Muteness failure detectors for consensus with Byzantine processes. Brief an-
[7] M.J. Fischer, N. Lynch, M.S. Paterson, Impossibility of distributed consensus with one faulty process,
[8] M. Hurfin, M. Raynal, A simple and fast asynchronous consensus protocol based on a weak failure detector,